THESIS ON INFORMATICS AND SYSTEM ENGINEERING C115

Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems

AIVO ANIER

TUT PRESS

TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Computer Science

This dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer Science on July 27th, 2016.

Supervisor:	Prof. Jüri Vain Department of Computer Science Tallinn University of Technology, Estonia
Opponents:	Prof. Juha Röning, Ph.D. Infotech Oulu and Department of Electrical and Information Engineering University of Oulu, Finland
	Prof. Ivan Porres, Tk.D. Computer Engineering at the Faculty of Natural Sciences and Technology Abo Akademi University, Finland

Defence of the thesis: August 30th, 2016

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.

Aivo Anier

Copyright: Aivo Anier, 2016
ISSN: 1406-4731
ISBN: 978-9949-23-995-5 (publication)
ISBN: 978-9949-23-996-2 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C115

Mudelipõhine raamistik küber-füüsikaliste süsteemide hajusjuhtimiseks ja -testimiseks

AIVO ANIER

TTÜ
KIRJASTUS

Contents

List of Publications

The work of this thesis is based on the following publications included in the Appendices, with author's contributions:

1. Anier, Aivo; Vain, Jüri (2010). **Timed automata based provably correct robot control.** BEC 2010 : 12th Biennial Baltic Electronics Conference, [Proceedings : Tallinn University of Technology, October 4-6, 2010, Tallinn, Estonia]. [S. l.]: IEEE, 201–204.

   The author designed and implemented the Scrub Nurse Robot control software. The robot general concept and hardware platform was developed earlier as a result of cooperation between TTU and Tokyo Denki University (TDU). The author spent two months in internship at TDU for better understanding of the project and the hardware used. The author prepared and presented the paper at 12th Biennial Baltic Electronics Conference (BEC) summarizing a 3-year work.

2. Vain, J., Miyawaki, F., Nõmm, S., Totskaya, T., & Anier, A. (2009, August). **Human-robot interaction learning using timed automata.** In ICCAS-SICE, 2009 (pp. 2037-2042). IEEE.

   The author participated in the development of the learning algorithm and laboratory setup for human-robot interaction learning. He designed and implemented an integration software for automated passive infrared camera based 3D measurement system (3DMS) to record and replay the motion capture inputs for the learning algorithm. This resulted in a unified platform for the TDU capture system, Tallinn University of Technology 3DMS and offline use without the expensive laboratory setup.

3. Vain, J; Miyawaki, F.; Nõmm, S.; Totskaya, T.; Anier, A. (2009). **Human-robot interaction learning using timed automata.** ICCAS-SICE 2009 : ICROS-SICE International Joint Conference 2009, Fukuoka City, Japan, August 18-21, 2009, Proceedings. Tokyo: IEEE/SICE, 2037–2042.

   The author participated in the development of voting automata training by contributing with the upgrading of integration software described for the previous paper (Human-robot interaction learning using timed automata).

4. Anier, A.; Vain, J. (2012). **Model based continual planning and control for assistive robots.** HEALTHINF 2012 : Proceedings of the International Conference on Health Informatics, Vilamoura, Algarve, Portugal, 1 - 4 February, 2012. Ed. Conchon, Emmanuel; Correia, Carlos Manuel B.A.; Fred, Ana L.N.; Gamboa, Hugo. SciTePress, 382–385.

The author re-designed and re-implemented previous work to generalize the software and apply it for distributed testing (later named DTRON). The results were a part of Competence Centre in Electronics, Info and Communication Technologies (ELIKO) sub-project. The author prepared and presented the paper at International Conference on Health Informatics.

5. Vain, Jüri; Anier, Aivo; Halling, Evelin (2014). **Provably correct test development for timed systems.** Databases and Information Systems VIII : Selected Papers from the Eleventh International Baltic Conference, Baltic DB&IS 2014. Ed. Haav, Hele-Mai; Kalja, Ahto; Robal, Tarmo. Amsterdam: IOS Press, 289—302. (Frontiers in Artificial Intelligence and Applications; 270).

   The author participated in preparing the paper by adding the test development and related to that latency monitoring features to DTRON, also by deriving the test deployment correctness verification criteria

Publications not included

1. Kirt, T., & Anier, A. (2006, October). **Self-organization in Ad Hoc Wireless Networks.** In Electronics Conference, 2006 International Baltic (pp. 1-4). IEEE.

2. Jüri Vain, Evelin Halling and Gert Kanter, Aivo Anier, Deepak Pal. **Model-based testing of real-time distributed systems.** Databases and Information Systems. International Baltic Conference, Baltic DB&IS 2016, Riga. (pp. 1-14). (accepted for publication)

List of Abbreviations

CPS	Cyber-Physical Systems
DI	Dependency Injection (programming pattern and -frameworks)
DOF	Degree Of Freedom
DTRON	Distributed Testing Realtime systems Online
EFSM	Extended Finite State Machine
ELIKO	Competence Centre in Electronics , Info and Communication Technologies
FIFO	First-In-First-Out
HRI	Human-Robot Interaction learning
IEEE1394	aka. FireWire interface standard
IOCO	Input-Output Conformance relation
IOTS	Input-Output Transition System
IUT	Implementation Under Test
JSNR	Java Scrub Nurse Robot (a predecessor of DTRON, built for SNR)
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MB	Model-Based
MBC	Model-Based Control
MBD	Model-Based Design
MBT	Model-Based Testing
NTA	Network of Timed Automata
PCD	Provably Correct Development
PCO	Point of Control and Observation
PTA	Probabilistic Timed Automata
RPT	Reactive Planning Tester

RPT	Reactive Planning Tester
RT-IOCO	Real-time Input-Ouput Conformance relation
SCM	Source Code Management
SNR	Scrub Nurse Robot
SR	Strong Responsiveness
SUT	System Under Test
TA	Timed Automata
TAIO	Timed Input-Output Automata
TCTL	Timed Computation Tree Logic
TDU	Tokyo Denki University
TIOTS	Timed Input-Output Transition System
TRON	Uppaal TRON (Testing Realtime systems Online)
TS	Transition System
TUT	Tallinn University of Technology
UPTA	Uppaal Timed Automata

List of Figures

List of Algorithms

1 Introduction

This thesis presents results on developing the DTRON framework to automatize model based control and testing of time critical applications. This section presents the background and motivation of thesis, defines the scope of research, postulates the research goals and finally presents the main contributions and the thesis structure.

DTRON relies on Uppaal model checking tool [7] and on-line test execution tool TRON [3], extending these tools by enabling coordination and synchronization of distributed components and providing a consistent API based on standard Java technology. The research focus of the thesis is verifiable modelling, and execution of models to enable provably correct on-line testing, model-based control and monitoring in robotic applications.

This work was originally motivated by the Scrub Nurse Robot project [8] where human adaptive control scenarios and on-line safety monitoring were main design concerns. The same design principles and technical solutions later appeared to be relevant also for remote and distributed testing of web-based applications as demonstrated in a street light control software testing case-study.

The novel ideas of DTRON framework design revealed also extension opportunities for its application in broader class of Cyber-Physical Systems capitalizing on dynamic reconfiguration, self-calibration depending on the dynamic delay estimates of the deployment configuration, and on-the-fly feasibility checks of models used in control and testing applications.

The author of thesis believes that the ideas presented in thesis contribute also to improve the practical development processes of broader class of industrial scale time critical Cyber-physical systems.

1.1 Cyber-physical systems

A cyber-physical system (CPS) is a system of collaborating computational elements controlling physical entities [9]. Contemporary CPS integrate computation with physical processes. CPS combines a cyber side (computing and networking) with a physical side (e.g., mechanical, electrical, and chemical processes). Such systems present the biggest challenges and biggest opportunities in several critical industrial segments such as electronics, energy, automotive, defence and aerospace, telecommunications, instrumentation, and industrial automation [10].

Applications of CPS include high confidence medical devices and systems, traffic control and safety, autonomous automotive systems, process control, environmental control, avionics, (smart energy grids and communications systems for example), tele-medicine, defence systems, manufacturing, and smart structures and other. The positive economic impact of any one of these applications areas is hard to overestimate. On the other hand, the development costs of CPS and the risk of having quality deficiencies in the software increase at least proportionally with the complexity growth of CPS software.

1.1.1 Design challenges of CPS

In [11] it is stated that governing the complexity and design correctness issues of large-scale CPS software requires major advancement in algorithmic techniques. The same applies to methodologies and tools that address the problems of intrinsic concurrency and timing constraints over large spectrum of CPS time scale heterogeneous architectures. The most widely used networking techniques today introduce a large extent of variability in timing, safety, performance, security etc.

The systems will be unable to benefit from the variety of technology improvements without redoing the (extremely expensive) validation and certification of the software. Evidently, the CPS design concerns such as design optimality and functional efficiency are more and more paired with predictability. Predictability is difficult to achieve without relevant level of abstraction that extracts only the features of concern and their interactions and hides irrelevant details. One of the most radical transformations comes from the networking of feature-rich system components. Therefore, bench testing and encasing without exhaustive formal verification become inadequate for evolving networked CPSs.

It becomes impossible to test the software under all possible conditions or guarantee satisfaction of control goals when the design descriptions lack adequate information about these features. Moreover, general-purpose networking techniques themselves make program behaviour much more unpredictable. While features of functionality have gained major attention in CPS design, achieving the predictable timing in the face of such openness remains still serious technical challenge. The authors of [12, 13] have come to the conclusion that CPS software quality and software process productivity issues can be addressed properly by model-based techniques and the tools that operate on relevant level of abstraction. Only this can make the automated analysis and synthesis of CPS software regarding timing, concurrency and resource sharing aspects rigorous, tractable and comprehensible. As for the general characteristics of CPS software frameworks, author of [14] outlines following:

- Modularity that allows complex software to be manageable for the purpose of implementation and maintenance. The logic of partitioning may be based on related functions, implementation considerations, data links, or other criteria.

- Distributed Resource Sharing. Failure to properly resolve resource contention problems of distributed services may result in a number of problems, including deadlock, live-lock, and domino effect like dropping off services.

- Openness with many facets, including among others :
  - configurability, i.e. making changes to the basic look or feel or behaviour of the software;
  - extendibility, i.e. adding functionality to the software that was not in the original distribution. This is usually done via the software's plug-in structure and parametrization of functions;
  - interoperability, i.e. the ability to work with other programs over a network, using, e.g. a robust API, Open Network Protocol or Open Network Interface.
- Scalability, e.g. a web application that runs in an application server may have the number of executable lines of code easily thousands - not to mention concurrency issues; scaling this up to clustered environments like high availability deployments and clouds, it is likely to grow out of comprehension.
- Maintainability, that is a measure of the ease and rapidity with which a system or equipment can be restored to operational status following a failure.
- Fault Tolerance, distributed CPS must maintain availability even at low levels of hardware/software/network reliability. Fault tolerance is achieved by recovery and redundancy.
- Transparency [15], i.e. distributed CPS should be perceived by users and application programmers as a whole rather than as a collection of cooperating components.
- Self-monitoring, it is an ability to monitor the correctness of online performance, detect malfunction and faults, recover, etc.
- Model based high level user interfaces and operation control.

Most of these qualities apply to sub-classes of CPS such as multi-robot systems and networked control systems or industry automation software frameworks, e.g. UA OPC framework [16], ControlShell, IBM Rhapsody, NEXUS [17], PTIDES [18], etc. In addition to listed generic properties, the CPS software frameworks being used in industry must also have strong practical value in terms of reliability, adequate price, ease of use by software analysts/engineers, scalability and compatibility with main stream software technologies and standards.

1.2 Motivation

The general goal of thesis is to develop an adequate *execution framework* and toolchain for improving the quality of CPS software and related development processes by relying on state-of-the-art formal methods and software technologies.

1.2.1 Application of formal methods in CPS design

There is extensive effort and long history of applying formal methods for software synthesis and correctness verification dating back to 1960s. Deductive verification [19] presumes that the program is annotated with formalized requirements (pre-, post conditions, invariants, variants) and then using proof-assistants to (semi-) automatically show the requirements are satisfied.

Abstract interpretation based static analysis [20] is built on abstract domain and fix-point theories. Static analysis, which automatically infers dynamic properties of computer systems, has been successful in last years to automatically verify complex properties of real-time, safety critical, embedded systems. Typical applications to static analysis are the automatic, compile-time determination of run-time properties of programs and software verification(conformance to a specification).

Another group of formal methods that have been taken into practical use in safety critical systems industry rely on model checking [4] and are often combining model checking with symbolic constraint solving [21]. Alas, contemporary software industry practice shows that the industry still trusts on testing at most [22]. In case of testing there is the real system in the verification loop that provides higher confidence in verification results than just formalized assertions about the implementation. Combining the advantages of testing, planning theory and formal verification led to model based testing - the core technique the thesis is focusing on.

The other set of motivating factors of thesis has grown out from practical assistive robotics. Specifically, it is the problem in human adaptive robotics where guaranteeing the timing correctness of coordinated robot actions responsive to corresponding human action is of utmost importance [23]. This is where it is not only sufficient for the program to provide functionally correct behaviour, e.g. spatial trajectory, but equally important is *how* and *when* this behaviour occurs. It cannot be either too early or too late, not to mention that robotic applications require inherently concurrent movements of many joints or manipulators.

For instance, consider the task of trajectory planning for a manipulator arm having joints for 6 degrees-of-freedom. Since the joint motors are usually activated simultaneously one has to guarantee that the manipulator does not crash into itself. Alternatively, one could think of a mobile robot platform performing simultaneous localization and mapping (SLAM). How to guarantee that the robot does not hit an obstacle and decelerate, possibly to an emergency stop, to avoid it?

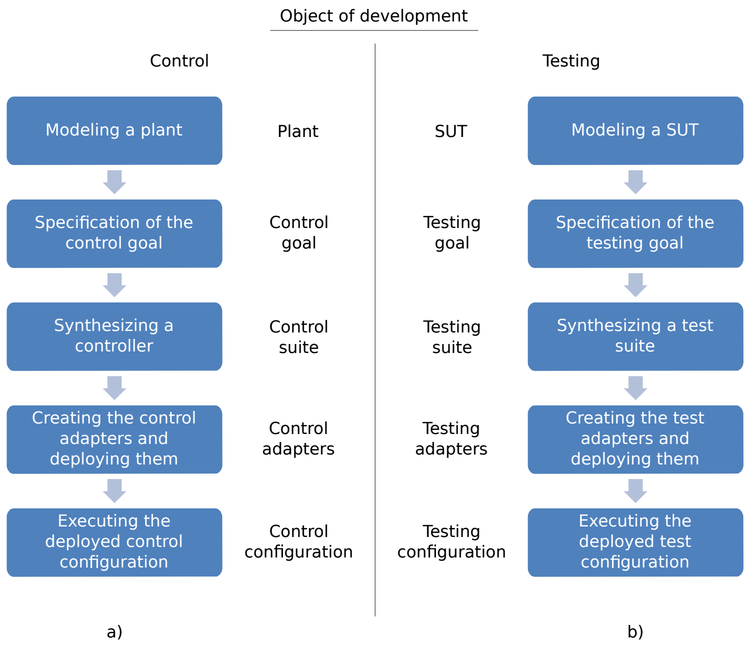
If we put these considerations into a safety critical context like robot assisted brain tumour surgery [24], the problem escalates to life critical one, e.g. robot manipulator holding a sharp scalpel could easily become a serious threat to the patient's life. Providing correctness of such human assistive robot designs is of utmost importance.

1.3 Scope of thesis

The general context of thesis is provably correct development (PCD) of time critical cyber-physical systems. In particular, it focuses on two types of CPS related artefacts: model-based supervisory control of assisting robots and model-based online test of distributed CPS software.

Model based design

The first set of issues that define the scope of thesis is related to the provably correct model-based design (MBD) workflow [25]. In model-based control design the development is manifested in four main steps [26] (see Figure 1.1a). Step 1: "Modelling a plant" means in case of robot control the modelling of behaviours observable and controllable at robot control interfaces. Step 2: "Specification of control goal" may apply different forms - a set of control scenarios, safety constraints to be followed, the target state of plant to be reached, etc. All of these goals need to be expressed in terms of the plant model and its attributes. Step 3: "Controller synthesis" results in a set of control rules or a controller model that when interfaced with plant model ensures that the control goal will be achieved. Mapping the abstract control to the real plant control deployment architecture requires Step 4: "Creating control adapters" that maps the model inputs/outputs to the ones of real physical plant (in our case of robot). And finally, Step 5: "Executing the control configuration" means running the implemented control solution on the execution environment.



Object of development
Control
Testing
Modeling a plant
Plant
SUT
Modeling a SUT
Specification of the control goal
Control goal
Testing goal
Specification of the testing goal
Synthesizing a controller
Control suite
Testing suite
Synthesizing a test suite
Creating the control adapters and deploying them
Control adapters
Testing adapters
Creating the test adapters and deploying them
Executing the deployed control configuration
Control configuration
Testing configuration
Executing the deployed test configuration
a)
b)

Figure 1.1: Comparison of development processes: a) Model-based robot control b) Model-based testing.

The model-based test development workflow depicted in Figure 1.1b and refined in Figure 1.2 follows similar steps, only the artefact to be developed is different. Instead of terms "plant", "control goal", "controller synthesis" there are terms "System Under Test" (SUT), "test purpose", and "tester synthesis" respectively. Similarly, both model based control and testing presume the development of adapters that interface models with real world objects and when the design product together with adapter is deployed they form the executable configuration (see Figure 1.2).

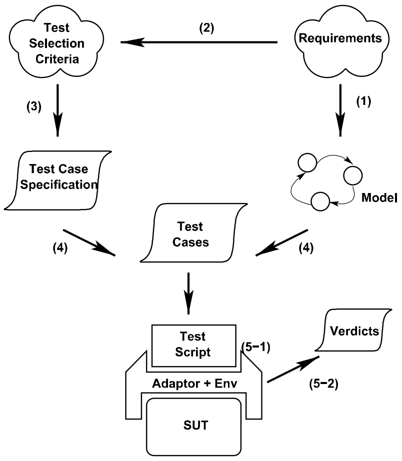


Figure 1.2: The process of Model-Based Testing.[1]

Though the processes outlined in 1.1 rely inherently on the notions of model and specification, in contrast to the provably correct development disciplines [27], they do not require verification of development results explicitly.

Model based testing

Therefore, the thesis unfolds the development process and considers development steps paired with verification and design correctness assurance steps as depicted in 1.3. In the implementation of this approach the templates of verification conditions can be derived for MBT process automatically assuming a concrete test generation method such as for reactive planning testers [22].

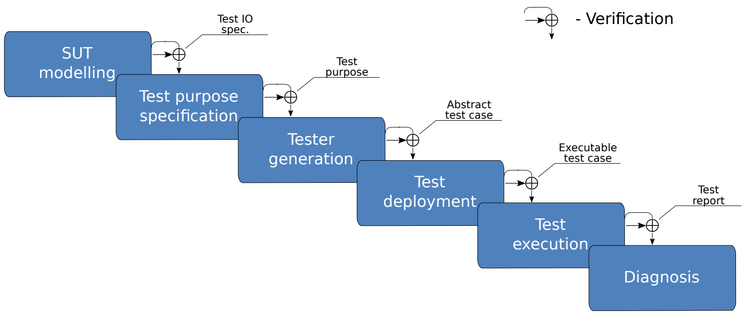


Figure 1.3: Provably correct MBT workflow.

Another general consideration, the thesis is targeting on, is uniformness of methods and tools needed to implement the provably correct development processes. It means covering both the plant or SUT model construction, control/test goal specification, controller/tester synthesis, control/test adapter building and deployed control/test configuration execution steps by the same tool set.

Under these general considerations, the thesis focuses on specific development steps such as modelling, deployment and execution. At first, the model construction (either of a plant or SUT) is considered to be one of the most laborious step in the MBD workflows [28]. In the supervisory control [29] it means the identification of plant behaviour at its input/output ports. Since the controllers used for robot action planning and control are discrete, the high-level model of plant behaviour is typically a state machine and the modelling step can be enhanced by means of automated model learning methods. For instance, the automata learning algorithm in [30] construct the transition relation of the model incrementally by observing the system control inputs and the responses on its outputs. In case of Scrub Nurse Robot (SNR) the robot has to learn the behaviour of interacting humans. It means that the sequences of interaction learned need to be partitioned by active parties of the interaction, i.e. surgeon and nurse - the surgeon whom to collaborate with later on, and the nurse who's role has to be taken over by the robot. After learning the interaction, the SNR responding to the surgeon's action needs to choose the reaction from those being recorded.

Alternatively, the SUT models for model-based conformance testing are constructed in the course of formalizing the requirements specification of SUT. Alas, the model learning can be applied here only as an assistive measure of model construction because learning the model by observing the i/o behaviour of SUT introduces same faults and flaws that need to be detected later by testing. It means that after learning SUT behaviour by i/o monitoring the model revision is needed to harmonize it with requirements specification, that can be tedious and time consuming process. Second drawback of applying the model learning approach for constructing SUT model is that SUT itself needs to be implemented before getting interaction logs. Therefore, instead of learning SUT behaviour it is more feasible to learn the SUT environment behaviour to emulate it in the test suite to achieve close as much as possible conditions to real operational environment. An example of this application is learning the load profiles of requests to system services.

Deployment and execution

Finally, in the centre of gravity of thesis there are deployment and model-based execution. These are the steps where beside formal semantics of models also real world constraints come into play, e.g. computational and communication deployment architecture, scheduling policy, software/ hardware jitter and implementation imperfections. When executing the model-based systems, the effects of such constraints and imperfections have to be addressed explicitly and their side effect minimized in the final product.

The main goal in designing the deployment architectures and execution environments is to achieve that the physical execution is following the model defined semantics as much as possible. When comparing the execution architectures of model-based control (MBC) (Figure 1.4) and MBT (Figure 1.5) following similarities on the level of components architecture can be outlined: *System Under Control* and the *System Under Test* are manipulated and their reactions observed either by *Hardware adapter* and *State estimator* or a *test adapter*; the *State control* corresponds to the *SUT environment emulator*; the *State knowledge* represents estimate of the plant state from mission perspective whereas the *Monitor* (*test oracle*) estimates if the observed state of SUT conforms with the state predicted by model; *Model* in MBC comprises the models of the plant as well as of the controller. Similarly - *TestSpec* comprises the model of implementation and its environment; *Knowledge goals* combined with *control goals* state same as *coverage criteria* state in testing; *Mission Planning & Execution* is analogue to *test case selection* and *test execution guiding* functionality in MBT.

Hypothesis

Correspondence between these two development flows and between the functionality of corresponding components led to the main hypothesis of thesis.

Namely, the model execution environment capable of executing abstract models that can be refined by domain specific semantics via adapters can be easily adapted for both the MBC and MBT.

Main functional adaptation effort remains only in the development of either plant or SUT adapters, that needs to be done with each new testing/control application per se.

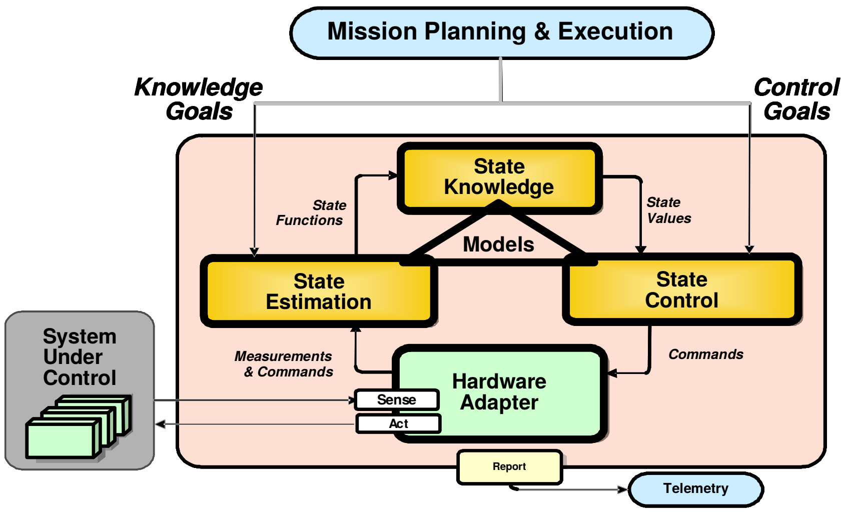


Figure 1.4: Conceptual view of the model-based control architecture.[2]

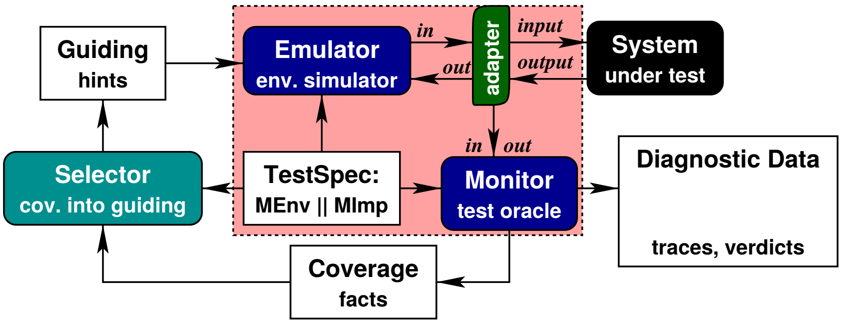


Figure 1.5: Conceptual view of the model-based testing architecture[3].

Thus, the specifics of the target application domain and the semantics of formalism used for modelling set the main requirements to the operational features of the model execution environment:

· Robot control sets requirements such as fast and deterministic response time, the need for interfacing with standard robot software platforms such as ROS (Robot Operating System).

· Distributed testing presumes handling issues that typically accompany execution in distributed computational environment: true parallelism, timing imperfections due to hardware/communication/software jitters; in practical terms it may lead to the need for relaxing the conformance relation between the model and SUT.

· The semantics of modelling formalism (e.g., Uppaal timed automata) sets the requirements such as parallel mixed synchronous/asynchronous execution of processes, event timing w.r.t. clock constraints etc.

· The framework implementation technology and platform sets constraints to the framework software architecture solution.

1.4 Goals

The goal of thesis is to develop a provably correct development workflow and execution framework for model-based robot control and testing that addresses the issues of concurrency, scalability and real-time constraints imposed by CPS design models. The broader context of this research is model-based development (MBD) paradigm and state of the art software technologies of building MBD automation tools.

Though the results of thesis are theoretically well founded the main intention is not advancing the underlying theory of provably correct development methods. Instead, the thesis rather capitalizes on the engineering approach by choosing the best viable existing theory, programming technology and building the framework with the perspective of being practically used for research as well as for industrial software projects.

Specifically, the goals of the thesis are following:

1. To analyse and extract the ingredients of model-based control and -testing theory and existing tools necessary for model-based applications in the field of robot control and distributed testing of CPS.

2. To study and formulate the verification conditions applicable in provably correct model-based testing development workflow.

3. To develop an algorithm of incremental learning of UPTA automata and demonstrate its applicability for model-based control of human assisting robots and conformance testing.

4. To study the aspects needed for deployment and execution of distributed control tasks and distributed tests with time constraints in real model execution environments.

5. To apply these results as system design requirements for developing the prototype framework for model-based control and model-based testing execution.

1.5 Methodology

The goals outlined in Section 1.4 presume underlying theoretical framework for implementing the model based (MB) workflow related tasks. The requirements specification and subsequent development steps need to be defined preferably within the same formal semantic space. Given the specifics of application domain and tasks of MBD the Uppaal Timed Automata (UPTA) is selected due to its expressive power and good tool support. The Uppaal tool set [7] is used throughout the development process both for MBC and MBT. The standard Uppaal tool includes a model editor, simulator and model checking engine for timed computation tree logic (TCTL). For implementing the provably correct development workflow the templates of verification conditions for MBT process are constructed and expressed in TCTL, and thereafter discharged using Uppaal model checker. Uppaal TRON is a testing tool, based on Uppaal engine, suited for black-box conformance testing of timed systems, mainly targeted for embedded software commonly found in various controllers [3].

The first step of the MBD workflow - the model construction for behaviour planning and control is implemented as an algorithm of unsupervised learning of human-robot interaction [30]. The algorithm produces a composition of timed i/o automata (TAIO) [31] where each automaton represents behaviour of an interaction party. Specifically, the class of TAIO used is non-deterministic, non-blocking input complete TAIO [32].

In MBT, the SUT specification and its environment are also encoded in UPTA although the model learning technique [30] as discussed above cannot be applied directly for constructing SUT models. The requirements model formalises the set of requirements the system is expected to comply with. These requirements are therefore subject to (abstract) test generation (and selection) and checking the conformance relation between the requirements specification model and SUT, namely, if the exposed i/o behaviour conforms to that of requirements model. This relation is called input-output conformance (IOCO) relation and the input-output testing is called *conformance testing*. The system under test (SUT) could be any system that is effectively controllable and observable through test adapters. This would include software, but also robotic platforms and CPS components controlled by software. An adapter is a piece of code that acts as an interpreter between the i/o events of the model and the SUT. An abstract event in the model possibly triggers executable code in the SUT. This is done by transforming the abstract event to SUT executable form in the adapter.

Though the case studies used in the thesis involve design and implementation examples of several adapters the theoretical aspects and general methodology of building adapters is out of the scope of thesis. Still, some timing aspects due to the adapters need to be accounted in IOCO testing of distributed CPSs. For time critical systems the correctness of timing needs to be included in the conformance relation, thus, applying in thesis RT-IOCO relation[33] instead of untimed IOCO is necessary. Ideally, the RT-IOCO needs to be supported in distributed testing but due the communication jitter and many other physical factors in distributed systems the RT-IOCO testing is practically infeasible. Therefore the thesis focuses on the notion of weaker conformance relation - delta-testability [34]that is one of the main design consideration for execution environment DTRON is designed for.

1.6 Contribution

Provably correct model based test development workflow has been introduced and verification conditions necessary to assure the correctness of development increments were outlined in the thesis. Two model learning algorithms were developed for automatic construction of models of interaction between the system (under test/control) and its environment and verifiable correctness criteria were specified as proof obligations suited for automatic model checking.

Main contributions of the thesis are:

- The design and implementation of distributed model-based control and test execution framework DTRON that enables model driven execution of robot control stacks as well as test suites of model based testing.
- “Scrub Nurse Robot” project case study on the applicability of DTRON framework and model-based techniques and
- a case study on distributed performance testing of Tartu City street light control system.

1.7 Thesis Structure

Chapter 1 gives the motivation, scope, main goals and the methodology of thesis by highlighting several practical considerations the results of thesis have to satisfy. The motivations and the scope of study sections refine the domain and research scope by listing requirements for the tool support of provably correct work flow for test and robot control development.

Chapter 2 gives the detailed view of the underlying theory and explains different forms and rationale of conformance relation to be used in model based testing and robot control.

Chapter 3 introduces provably correct test development workflow with verification conditions necessary to assure the correctness of development increments.

Chapter 4 presents two versions of a timed automata model learning algorithm, one relevant for learning from recordings of human motions during surgical procedures, the other adjusted for learning from network traffic monitoring logs.

Chapter 5 gives detailed architectural overview of the DTRON framework, its design principles, implementation considerations.

Chapter 6 evaluates the results with a series of performance and robustness experiments.

Chapter 7 demonstrates the applicability of the approach by describing two representative case-studies: SNR model based control and a distributed performance testing case study for the street light control system.

Conclusion sums up the key results of thesis and provides ideas for future work.

Finally, the appendix lists the papers being the theoretical foundations, related work and implementation technologies the results of thesis are built upon.

2 Preliminaries

2.1 Chapter overview

In this chapter the theoretical foundations of model-based robot control and testing of time-dependent distributed systems addressed in the Section 1.3 will be introduced. At first, the formal modelling alternatives are discussed from pragmatics, taxonomy, and expressiveness point of view and their relevance is motivated w.r.t. requirements stated in Section 1.5. Section 2.2.3 introduces the semantics of UPTA as one of the formalisms that meets the requirements outlined in Section 2.2. Relying on the semantics of UPTA the rigorous definition of notions such as conformance, responsiveness, observability, controllability applied later for MBT and MBC are defined. Section 2.3.1 outlines the underlying algorithmic methods for TCTL model checking, model based conformance testing that both rely on the syntactic and semantic notations of the UPTA that is used throughout the thesis.

2.2 Models for time-dependent concurrent systems

2.2.1 The taxonomy of models

The modelling languages are often separated into two broad classes: operational languages and descriptive languages[35]. Operational languages are well-suited to describe the evolution of a system starting from some initial state. Common examples of operational languages are the differential equations used to describe dynamic systems in control theory, automata-based formalisms (extended finite-state machines, Turing machines, timed automata) and other discrete event systems. Operational languages are based on the concepts of state and transition (or event), so that a system is modeled as evolving from a state to the next one when a certain transition/event occurs[36]. Descriptive languages (also called declarative languages) are better suited to describing the properties (static or dynamic) that the system must satisfy. Classic examples of descriptive languages are logic-based and algebra-based formalisms[36].

For practical purposes it is common to use a combination of operational and descriptive formalisms to model and analyze systems in a mixed-language approach. In mixed approach, an operational language is used to represent the dynamics of the system (i.e., its evolution), while its requirements (i.e., the properties that it must satisfy, and which one would like to verify in a formal manner) are expressed in a declarative language. Model checking techniques[37] and the combination of Petri nets with the TRIO temporal logic[38] are examples of the mixed language approach.

Almost all the models of time-dependent systems describe the systems in terms of behaviours. A behavior of a system is a mapping $b : T \rightarrow S$, where T is a time domain and S is a state space. The behavior represents the system's state (i.e., the value of its elements) in the time instants of T. Thus, one natural way of building the taxonomy of models can be based on how the system states, time domains and mapping b are defined. To motivate the selection of modelling formalism for this work we characterize the selection decisions by using the metrics proposed in [36]. Dealing with timed systems we examine in the first place the dimensions of models that characterize the way how time is interpreted in the models.

Discrete *vs* dense time domains. First categorization of the formalisms dealing with time-dependent systems and related time models is whether such a model is a discrete or dense set. A time set consists of isolated points, whereas for every two points t_1, t_2 of a dense set, with $t_1 < t_2$, there is always another point t_3 in between, s.t. $t_1 < t_3 < t_2$. The discrete time set is modelled by natural and integer numbers whereas the typical dense models are rational and real numbers. For instance, differential equations are normally stated with respect to real variable domains, whereas difference equations are defined on integers or rationals.

Bounded *vs* unbounded time domains. Reactive systems generally represent behaviors that may proceed indefinitely in time, so that it is natural to model time as an unbounded set. There are significant cases, however, where all relevant system behaviors can be *a priori* enclosed within a bounded "time horizon". For instance, starting an engine to its full speed requires at most 15 seconds. Thus, if we want to model and analyze possible phases of its speed up, without loss of generality we can assume a bounded time domain, say, the real range $[0 \ldots 15]$. In many cases this restriction highly simplifies analyses and/or simulation algorithms. In other cases the system under consideration is periodic; thus, knowing its behaviors during full period provides enough information to extrapolate these properties over the whole time axis[38, 36].

Single *vs* multiple time scales. Another dimension of time domains is their *granularity*. The behaviours are said to be on different *time scales* if they differ by orders of magnitude. This is typical when two processes e.g. mechanical deterioration of the cutting edge of a milling machine is modelled together with exchange of operational modes during processing a work piece. Here the time scales are hours *vs* milliseconds. Usually, due to the simplification purposes the behaviours of different time scales are modelled separately. When studying the fine-grain dynamics the one of coarser time scale is abstracted away. To avoid side effects caused by this abstraction, the behaviours are represented case-wise and the parameters of coarser time granularity processes are assumed to be static for the fine grain modelling case.

Qualitative *vs* metric time. For modelling real-time systems an essential issue is the expression of constraints that exploit the metric structure of the underlying time domain. A time domain is a *metric structure* when a notion of distance is defined on it. The domains $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ have a "natural" metric defined on them where the distance d between any two points t_1 and t_2 is $d(t_1, t_2) = |t_1 - t_2|$. When the formalism allows expressing only information about the relative ordering of events ("q occurs after p"), but not about their distance ("q occurs 100 time units after p"), we say that the language has *qualitative notion of time*, as opposed to allowing *quantitative constraints*, which are expressible with metric time. For the formal description of parallel systems as defined in[39], a purely qualitative language is sufficient. The correctness of the computation in such a system depends only on the relative ordering of computational steps, ignoring absolute distance between them. Also reactive systems[40] belong to this category because they are often modelled as parallel systems, where the system evolves concurrently with the environment. The correctness of real-time systems depends also on the time distance among events. Hence a complete description of such systems requires a language in which metric constraints can be expressed.

Linear *vs* Branching Time. Linear time notion refers to (a set of) behaviors, where the evolution from a given state at a given time is always unique. Conversely, a branching-time interpretation refers to behaviors that are structured in trees, where each "present state" may evolve into different "possible futures". Though a linear behavior is a special case of a tree (a tree might be thought of also as a set of linear behaviors that share common prefixes, i.e., that are *prefix-closed*), this notion is captured formally by the notion of *fusion closure*[41]. Thus, linear and branching time models can be put on a common ground in formalisms such as CTL^* [4] that has two sub-logics LTL (for linear time) and CTL (for branching time). In practice the choice of linear or branching time models is mostly based on the pragmatic considerations, i.e. what properties are more convenient to express and prove in given logic. For instance, the complexity of model checking often directs what logic and what underlying time concept to use (see Table 2.1, where TS denotes the state transition system and Φ the property expressed in the logic shown in the heading of the column).

	CTL	LTL	CTL*
model checking	PTIME	PSPACE-complete	PSPACE-complete
without fairness	size(TS) $\cdot$ $\lvert\Phi\rvert$	size(TS) $\cdot$ exp($\lvert\Phi\rvert$)	size(TS) $\cdot$ exp($\lvert\Phi\rvert$)
with fairness	size(TS) $\cdot$ $\lvert\Phi\rvert$ $\cdot$ $\lvert$fair$\rvert$	size(TS) $\cdot$ exp($\lvert\Phi\rvert$) $\cdot$ $\lvert$fair$\rvert$	size(TS) $\cdot$ exp($\lvert\Phi\rvert$) $\cdot$ $\lvert$fair$\rvert$
for fixed specifications (model complexity)	size(TS)	size(TS)	size(TS)
satisfiability check	EXPTIME	PSPACE-complete	2EXPTIME
best known technique	exp($\lvert\Phi\rvert$)	exp($\lvert\Phi\rvert$)	exp(exp($\lvert\Phi\rvert$))

Figure 2.1: Complexity of the model-checking algorithms and satisfiability checking.[4]

Determinism *vs* non-determinism. The notions of determinism and non-determinism are attributes of the systems being modelled or analyzed. Non-determinism is not necessarily only the system property, it may arise also due to abstraction from the elements of the model that define the constraints on behaviours. We say that for a given input sequence, the behavior of a deterministic system is uniquely determined by its initial state. Conversely, systems that can evolve to different future states from the same present state and the same input by making arbitrary "choices" are called nondeterministic. There is a natural coupling between deterministic systems and linear models, on one side, and non-deterministic systems and branching models, on the other side, where all possible "choices" are mapped to branches in the computation tree. In CPS where non-determinism is often a natural phenomenon the branching time and formalisms with non-determinism are natural choice for practical modelling.

The Time Progress. The problem of time progress arises whenever the model of a timed system exhibits behaviors that do not progress past some time instant. Such behaviors may be the consequence of some incompleteness and inconsistency in the formalization of the system. Although a truly instantaneous action is physically infeasible, it is nonetheless a useful abstraction for events that take an amount of time which is negligible with respect to the overall dynamics of the system[42]. When instantaneous transitions are allowed, an infinite number of such transitions may accumulate in an arbitrarily small interval to the left of a given time instant, thus modeling a computation where time does not advance at all. Behaviors where time does not advance are called *Zeno behaviors*. Some formal systems possess Zeno behaviors, where the distance between consecutive events gets indefinitely smaller, even if time progresses. These systems cannot be controlled by digital controllers operating with a fixed sampling rate, since in this case their behaviors cannot be suitably discretized[43, 44]. Some well-known problems of concurrent computation such as termination, deadlocks, and fairness[45] can be considered as dual problems to time advancement. In fact, they concern situations where some processes fail to advance their states, while time keeps on flowing.

There are two solutions to the time advancement problem: *a priori* and *a posteriori* methods. In *a priori* method, the syntax or the semantics of the formal notation is restricted beforehand in order to guarantee that the model of any system described with it will be exempt from time advancement problems. *A posteriori* methods do not pose the restrictions to the model. Instead, they provide means for checking if the model is free from *Zeno* computations. *A posteriori* method may be particularly useful to spot possible risks in the behavior of the real system. For instance, in some cases oscillations exhibited by the mathematical model with a frequency that goes to infinity within a finite time interval is the symptom of the risk of serious failure in the real system [36].

Concurrency and Composition. Most real CPS are too complex to model or analyse them as one whole monolithic system. That suggests their treatment as the composition of subsystems where each component is simple enough so that it can be analyzed. From the temporal evolution point of view the compositions may be *synchronous* or *asynchronous*. Synchronous composition restricts state changes of various units to occur at time instants that are strictly and rigidly related. Conversely, in an asynchronous composition of parallel units, each activity can progress at a speed unrelated with other. For instance, in geographically distributed systems the state of one component may change in a time that is shorter than the time needed to send information about the component's state to other components. For asynchronous systems, interaction between different components occurs only due to data dependences and according to rules of chosen communication mechanism.

Expressiveness and Relevance to Analysis. The fundamental features of models for time-dependent systems are their *expressiveness* and *amenability* to analysis. The last refers to the capability of probing the model of a system to be sure that it ensures certain desired features. In a widespread paradigm[35, 46], the model under analysis is called a *specification*, and the properties that the specification model must exhibit are called the *requirements*. The task of ensuring that a given specification satisfies a set of requirements is called *verification*[36].

Expressiveness. A fundamental criterion according to which the modelling formalisms can be classified is their expressiveness, that is, the possibility of characterizing classes of properties. Informally, a formalism is more expressive than another if it allows expressing properties that more finely and accurately partition the set of behaviors into those that satisfy or not the constraints expressed. The expressiveness relation between formalisms is a partial order, as there are pairs of formal languages whose expressive power is incomparable (recall the example of LTL and CTL). The formalisms are equivalent if they can express the very same properties. The notion of expressiveness does not cover features such as *conciseness*, *readability*, and ease of use.

Decidability and Complexity. There is a fundamental trade-off between expressiveness and decidability. A property is *decidable* for a formal language if there exists an algorithmic procedure that is capable of determining, for any specification written in that language, whether the property holds or not in the model. The trade-off between expressiveness and decidability arises because, when increasing the expressiveness one loses decidability. The complexity analysis provides, in the case when a given property is decidable, a measure (in terms of memory or time) of the computational effort that is required by an algorithm to decide whether the property holds or not for a model.

Analysis and Verification Techniques. There exist two large families of verification techniques: (*i*) the *model checking methods* are based on exhaustive enumeration procedures and (*ii*) the *deductive methods* are based on syntactic transformations like deduction or rewriting, typically in the context of some axiomatic description[4]. Exhaustive enumeration techniques are mostly automated, and are based on exploration of graphs or other structures representing an operational model of the system, or the space of all possible interpretations for the sentence expressing the required property. Techniques based on syntactic transformations typically address the verification problem by means of *logic deduction*[47]. Therefore, usually both the specification and the requirements are in descriptive form, and the verification consists of successive applications of some deduction schemes until the requirements are shown to be a logical consequence of the system specification.

Conclusive remarks on the models taxonomy

The taxonomy of features of timing and concurrency expressing modelling languages suggests that the formalism to be used for modelling CPS should preferably be expressive enough to model

· the phenomena that are characteristic to unbounded dense time domain: Zeno behavior, fairness, explicit reference to metric time constraints;

- different time scales by specifying their interrelations;
- timing as well as data dependent non-determinism of behaviours;
- both synchronous and asynchronous concurrency between the parallel components of the system.

The formal notation should be supported also by the analysis methods where the properties of practical interest (safety, bounded reachability, etc.) are decidable and their verification feasible from the complexity point of view. The last presumes *modelling* and *verification automation tools* that meet the requirements of comfort and ease of use. In Subsections 2.2.2-2.2.4 the theory of timed automata and their extension to UPTA are outlined and the relevance w.r.t. criteria shown above explored based on formal notation.

2.2.2 Uppaal timed automata

The Uppaal Timed Automata (UPTA) modelling language is an extended dialect of Timed Automata (TA) formalism, maintaining real-time clocks and finite control structure. The underlying Timed Automata (TA) theory has its roots in Extended Finite State Machine (EFSM) theory. Example 2.1 and Figure 2.2 depicts a simple EFSM model of *login* action of an hypothetical system.

Example 2.1. System starts from the state *init*. The transition from *init* state to state OK is labeled with input/output pair denoting that when the input *login* is observed then the output *success* is generated and system moves to state OK. This automaton is non-deterministic though, because an input *login* may cause also executing the other transition that outputs $fail$ instead and reaches a different after-state NOK.

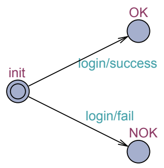


Figure 2.2: EFSM

Definition 2.2. An *extended finite state machine* (*EFSM*) is a tuple

$$M = (S, V, I, O, T, s_0) \tag{2.1}$$

where S is the set of states, V is the set of state variables, I is the input alphabet, O the output alphabet, T is the transition relation and s_0 is the initial state.

In TA the set V is limited with the set of clocks only, denoted by C. Let $B(C)$ denote the set of conjunctions over atomic propositions of the form $x \boxtimes c$ or $x - y \boxtimes c$ where $x, y \in C$, $c \in \mathbb{N}_0$ and $\boxtimes \in \{<, \leq, =, \geq, >\}$. The transition enabling conditions (*guards*) can be the conditions over subsets of C only and the variable updates are the resets of subsets of C.

Definition 2.3. A *Timed Automaton* is a tuple (L, l_0, C, A, E, I) where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ-action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I : L \to B(C)$ assigns invariants on clocks to locations.

A clock valuation is the function $u : x \to \mathbb{R}_{\geq 0}$ mapping a clock x from the set of clocks C to the set of non-negative reals $\mathbb{R}_{\geq 0}$. Let $\mathbb{R}^C$be the set of all clock valuations and $u_0(x) = 0$ for all $x \in C$. Guards and invariants are considered as clock valuations, and notation $u \in I(l)$ means that u satisfies $I(l)$.

When modelling concurrent components of systems the timed automata are composed into a network of timed automata (NTA) over a common set of clocks and actions consisting of n timed automata. Let the tuple $\left(L_i, l_i^0, C, A, E_i, I_i\right)$ denote an automaton $\mathcal{A}_i$ of the NTA, where, $1 \leq i \leq n$, a location vector for the whole network $\prod_i \mathcal{A}_i$ is $\bar{l} = (l_1, \ldots, l_n)$ and the invariant of the network is a conjunction over locations vector $I\left(\bar{l}\right) = \bigwedge_i I(l_i)$.

UPTA extend the expressiveness of NTA by introducing richer variable types and set of expressions used in the guards and update functions of edges and in the invariants. Expressions in UPTA range over clocks, booleans and integer variables and their arrays. The expressions are used with the following labels:

Guard is an expression that evaluates to a boolean; it includes terms such as clocks, integer variables, constants and function symbols (standard Uppaal functions and those implemented in the given model); clocks and clock differences are compared only to integer expressions; guards over clocks are essentially conjunctions although expressions over integers can be in disjunction.

Synchronization *label* is either of the form *Expression*! or *Expression*? or an empty label. The expression must be side-effect free, evaluate to a channel, and only refer to integers, constants and channels or channel arrays.

Assignment is a comma-separated list of variable updates where the expressions must only refer to clocks, boolean and integer variables and their arrays and constants. The clocks can be updated only with integer constants.

Invariant is an expression that refers only to clocks, constants and integer variables; it is a conjunction of conditions of the form $x < e$ or $x \leq e$ whenever x is a clock and e evaluates to non-negative integer.

Having described the UPTA requisites first informally we can now introduce the formal definition of UPTA as follows:

Definition 2.4. An *Uppaal model M* is a tuple

$$\left\langle \overrightarrow{A}, Vars\ , Clocks\ , Chan\ , Type \right\rangle$$

where

- $\overrightarrow{A}$ is a vector of processes $A_1, A_2, \ldots, A_n$ (a process A_j^i is j-th instantiation of an automaton template $\mathcal{A}^i$; for better readability we ignore the indexes of templates when it is clear from the context and the elements of a process A_j are referred to by applying the index of the process, e.g. L_j,l_j^0 , T_j.
- *Vars* is a set of variables (except clocks) defined in the model. It is a union over $Vars_j$ of processes and global variables of the model.
- *Clocks* is a set of clocks such that $Clocks \cap Vars = \emptyset$. Like *Vars*, *Clocks* is the union of all $Clocks_j$ in the processes of NTA.
- *Type* is a type function extending the type of locations to *urgent* and *committed* ones (their semantics will be defined in Section 2.2.3).

Definition 2.5. *Configuration* of an Uppaal model

$$M = \left\langle \overrightarrow{A}, Vars, Clocks, Chan, Type \right\rangle$$

is a triple $\left(\overrightarrow{l}, e, v \right)$ where $\overrightarrow{l}$ is a vector of locations, e is the valuation function of discrete variables and v is a clock valuation:

- $\overrightarrow{l} = (l_1, l_2, ..., l_n)$ where $l_i \in L_i$ is current location of processA_i
- $e : Vars \to \prod_i dom\,(v_i)$ maps every variable $v_i \in Vars$ to its value
- $v : Clocks \to R \geq 0$ maps the clocks to non-negative real numbers.

Configuration of the model corresponds to the state in the definition of NTA. The vector $\overrightarrow{l}$ is called *situation* (currently occupied locations in the automata of the model), pair$\left(\overrightarrow{l}, e \right)$ denotes the discrete part and v the continuous part of the configuration.

Before introducing the declarative property description language TCTL for time-dependent systems we need to define the operational semantics of UPTA that serves also as a common ground for checking satisfiability of TCTL formuli on UPTA models.

2.2.3 Semantics of the UPPAAL model

Given the fact that Uppaal timed automata are extension of NTA we define the semantics of NTA at first and the semantics of UPTA thereafter via semantics of NTA.

Definition 2.6. Let $\mathcal{A} = \prod_i \mathcal{A}_i$ be a network of n timed automata, where $\mathcal{A}_i = \left(L_i, l_i^0, C, A, E, I_i\right)$. Let $\bar{l}_0 = \left(l_0^1, \ldots, l_0^n\right)$ be initial value of location vector. The *semantics of NTA* $\mathcal{A}$ is defined as a transition system $\langle S, s_0, \longrightarrow\rangle$, where $S = (L_1 \times L_2 \times \ldots \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = \left(\bar{l}_0, u_0\right)$ is the initial state, and $\rightarrow \subseteq S \times S$ the transition relation is defined by

- $\left(\bar{l}, u\right) \rightarrow \left(\bar{l}, u + d\right)$ if $\forall d' : 0 \leq d' \leq d \implies u + d' \in I\left(\bar{l}\right)$
- $\left(\bar{l}, u\right) \rightarrow \left(\bar{l}\left[l_i'/l_i\right], u'\right)$ if there exists $l_i \overset{\tau,g,r}{\longrightarrow} l_i'$ and $u \in g, u' = [r \longmapsto 0]\, u$ and $u\prime \in I\left(\bar{l}\right)$
- $\left(\left(\bar{l}, u\right) \rightarrow \left(\bar{l}\left[l_i'/l_i, l_j'/l_j\right], u'\right), u\prime\right)$ if there exists $l_i \overset{c?,g,r}{\longrightarrow} l_i'$ and $l_j \overset{c!,g,r}{\longrightarrow} l_j'$ s.t. $u' = [r_i \cup r_j \mapsto 0]u$ and $u\prime \in I\left(\bar{l}\right)$,
  where $\bar{l}\left[l_i'/l_i\right]$ denotes the location vector $\bar{l}$ with its i-th element l_i replaced with l_i'.

UPTA state evolves either through enabled actions or delays. These steps define the behavior of the model. For configuration $\left(\overrightarrow{l}, e, v\right)$ a local action is enabled if there is a τ-transition in the underlying NTA. A synchronized action step is enabled iff for a channel b there exists the binary synchronization transition in the underlying NTA. A delay step with delay d is enabled iff such delay step is allowed in the underlying NTA.

Definition 2.7. Let $M = \left\langle \overrightarrow{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type}\right\rangle$ be a UPTA model. A sequence of configurations

$$\left\langle\left(\overrightarrow{l}, e, v\right)\right\rangle^K = \left\langle\left(\overrightarrow{l}, e, v\right)^0, \left(\overrightarrow{l}, e, v\right)^1, \ldots\right\rangle$$

of length $K \in \mathbb{N} \cup \infty$ is called a *well-formed sequence for M* iff

- $\left(\overrightarrow{l}, e, v\right)^0 = \left(\left(l_1^0, \ldots, l_n^0\right), \left[Vars \mapsto (0)^{|Vars|}\right], \left[Clocks \mapsto (0)^{|Clocks|}\right]\right)$
- if $K < \infty$, then for $\left\langle\left(\overrightarrow{l}, e, v\right)\right\rangle^K$ no further step is enabled
- if $K = \infty$ and $\left\langle\left(\overrightarrow{l}, e, v\right)\right\rangle^K$ contains finitely many k s.t. $\left(\overrightarrow{l}^k, e^k\right) \neq \left(\overrightarrow{l}^{k+1}, e^{k+1}\right)$, then eventually every clock exceeds every bound $\left(\forall x \in \mathit{Clocks}, \forall c \in \mathbb{N}, \exists k : v^k(x) > c\right)$,

Definition 2.8. A well-formed sequence for M is a *timed trace* for M if for every $k < K$, the two subsequent configurations k and $k+1$ are connected via a simple action step, a synchronized action step, or a delay step, i.e. $\left(\overrightarrow{l}^{k}, e^{k}\right) \xrightarrow{a} \left(\overrightarrow{l}^{k+1}, e^{k+1}\right)$ or $\left(\overrightarrow{l}^{k}, e^{k}\right) \xrightarrow{a} \left(\overrightarrow{l}^{k+1}, e^{k+1}\right)$ or $\left(\overrightarrow{l}^{k}, e^{k}\right) \xrightarrow{\tau} \left(\overrightarrow{l}^{k+1}, e^{k+1}\right)$.

Let M be a UPTA model, then the *trace semantics* of M, denoted $T(M)$, is the set of well-formed traces.

2.2.4 UPTA property specification language TCTL

Timed Computation Tree Logic ($TCTL$) is sufficiently expressive to allow for the formulation of an important set of real-time system properties[4]. Formulae in $TCTL$ are either *state* or *path formulae.* $TCTL$ state formulae over the set AP of atomic propositions and set C of clocks are formed according to the following grammar:

$$\Phi ::= \mathit{true}\ |a|g|\Phi \wedge \Phi|\neg\Phi|E\Box\varphi|A\Box\varphi|E\Diamond\varphi|A\Diamond\varphi$$

where $a \in AP$, $g \in B(C)$ (recall that $B(C)$ means atomic clock conditions defined in 2.2.2) and φ is a path formula defined by: $\varphi ::= \Phi U^{J} \Phi$ where U^{J} denotes *time bounded until* operator of CTL, $J \subseteq \mathbb{R}_{\geq 0}$ is an interval whose bounds are natural numbers. TCTL extends CTL with atomic clock constraints $B(C)$ over the clocks in C, the set of clocks in the timed automaton under consideration. The propositional logic operators $\vee, \rightarrow, \mathit{true}$, etc. are obtained in the usual way. The until operator is equipped with an interval J of real numbers. Timed variants of the modal operators $\Diamond$ and $\Box$ are obtained as follows: $\Diamond^{J}\Phi = \mathit{true}\ U^{J}\Phi$ and $E\Box^{J}\Phi = \neg A\Diamond^{J}\neg\Phi$ and $A\Box^{J}\Phi = \neg E\Diamond^{J}\neg\Phi$.

The formula $E\Box^{J}\Phi$ asserts that there exists a path for which during the interval J, Φ holds; $A\Box^{J}\Phi$ requires this to hold for all execution paths of M. Note that the path quantifiers E and A quantify over time-divergent paths only, and the interval $J \subseteq \mathbb{R}_{\geq 0}$ is either of the form $[n, m], (n, m], [n, m)$ or (n, m) for $n, m \in N$ and $n \leq m$. For right-open intervals, $m = \infty$ is allowed.

Like UPTA proposes richer modeling language than NTA, the version of TCTL used for stating UPTA properties and their model checking is an extension of "standard" TCTL which can be summarized as follows:

Given the UPTA model $M = \left\langle \overrightarrow{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type} \right\rangle$, a formula φ is syntactically correct iff it is formed according to the following rules:

$\varphi ::= deadlock\ \vert \nabla\triangle\phi$	where$\nabla \in \{A, E\}$, $\triangle \in \{\Box, \Diamond\}$ and
$\phi ::= A.l$	$A \in \vec{A}$, $l \in L_A$
$x \boxtimes c$	$x \in Clocks$, $\boxtimes \in \{<, \leq, =, \geq, >\}$, $c \in \mathbb{Z}$
$x{-}y \boxtimes c$	$x, y \in Clocks$, $\boxtimes \in \{<, \leq, =, \geq, >\}$, $c \in \mathbb{Z}$
$a \boxtimes b$	$a, b \in Vars \cup \mathbb{Z}$, $\boxtimes \in \{<, \leq, =, \geq, >\}$
(φ_1)	φ_1 - a local property
$not\ \varphi_1$	φ_1 - a local property
$\varphi_1\ or\ \varphi_2$	φ_1, φ_2 - local properties
$\varphi_1\ and\ \varphi_2$	φ_1, φ_2 - local properties
$\varphi_1\ imply\ \varphi_2$	φ_1, φ_2 - local properties

A local property is a condition that for specific configuration is either true or false. Condition can involve location names, variables and clocks which are only compared to integer values.

Definition 2.9. A local property ϕ is valid in configurations $s \in \left(\overrightarrow{l}, e, v\right)$, denoted $s \models \phi$ according to the following conditions:

$s \models deadlock\ iff\ no\ delay\ or\ action\ step\ is\ enabled\ in\ s$

$s \models A.l$	iff $l = l_i \in \overrightarrow{l}$ for $A = A_i \in \overrightarrow{A}$,
$s \models x \boxtimes c$	iff $v(x) \boxtimes c$, $\boxtimes \in \{<, \leq, =, \geq, >\}$,
$s \models x - y \boxtimes c$	iff $v(x - y) \boxtimes c$, $\boxtimes \in \{<, \leq, =, \geq, >\}$,
$s \models a \boxtimes b$	iff $e(a) \boxtimes e(b)$, $\boxtimes \in \{<, \leq, =, \geq, >\}$,
$s \models (\varphi_1)$	iff $s \models \varphi_1$
$s \models not\ (\varphi_1)$	iff $s \not\models \varphi_1$
$s \models \varphi_1$ or φ_2	iff $s \models \varphi_1$ or $s \models \varphi_2$
$s \models \varphi_1$ and φ_2	iff $s \models \varphi_1$ and $s \models \varphi_2$
$s \models \varphi_1\ imply\ \varphi_2$	iff $s \not\models \varphi_1$ or $s \models \varphi_2$

The trace semantics of $TCTL$ temporal operators is given in the following definition.

Definition 2.10. Let $M = \left\langle \overrightarrow{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type} \right\rangle$ be an UPTA model. Let φ and ψ be the local properties. The validity of temporal operators $A\Box$, $A\Diamond$, $\longrightarrow$, $E\Box$, $E\Diamond$, is defined as follows:

$$\begin{aligned}
M \models A\Box\varphi \quad & \text{iff } \forall \left\langle \left(\left(\overrightarrow{l}, e, v\right)\right)\right\rangle^K \in Traces(M), \forall k \in K : \left(l, e, v^k\right) \models \varphi \\
M \models A\Diamond\varphi \quad & \text{iff } \forall \left\langle \left(\left(\overrightarrow{l}, e, v\right)\right)\right\rangle^K \in Traces(M), \exists k \in K : \left(l, e, v^k\right) \models \varphi \\
M \models \varphi \longrightarrow \psi \quad & \text{iff } \forall \left\langle \left(\left(\overrightarrow{l}, e, v\right)\right)\right)\right\rangle^K \in Traces(M), \\
& \quad \forall k \in K : \left(l, e, v^k\right) \models \varphi \Rightarrow \exists k' \geq K : \left(l, e, v^{k'}\right) \models \psi \\
M \models E\Box\varphi \quad & \text{iff } M \not\models A\Diamond\ not\ \varphi \\
M \models E\Diamond\varphi \quad & \text{iff } M \not\models A\Box\ not\ \varphi
\end{aligned}$$

Figure 2.3 shows some examples how the $TCTL$ formulas are interpreted on a simple computation tree that represents traces of an hypothetical model M. Starting from an initial location a:

$A\Diamond\varphi$ (inevitable) true if local condition φ (e.g., valid in yellow locations) is reachable in all execution paths

$E\Diamond\varphi$ (possible) true if local condition φ (e.g., valid in a red location) is reachable in at least one execution path

$A\Box\varphi$ (always) is true if local condition φ holds in all locations of all execution paths not valid for given example assuming the initial location is a.

$\varphi \longrightarrow \psi$ (leads-to) is true if all paths involving a location where condition φ is valid include thereafter a location where ψ is valid

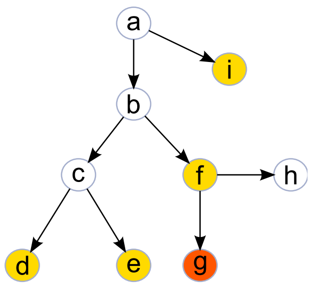


Figure 2.3: TCTL[5]

2.3 Implementation and conformance relations of timed models

The system correctness is often characterised formally using the notion of refinement relation. The *refinement* or *implementation relation* interrelates two models at various level of abstraction. For instance, the abstract model may be the description of system requirements and the less abstract model description of the system at some of its development phase. Similarly, these abstractions can originate from different development phases. The lower the abstraction level, the more implementation details are present. Implementation relations are used most often for comparing the models of the same system. Given a model of requirements specification and a more detailed system model the implementation relation allows checking whether the system correctly implements the requirements. For instance, given a requirement property φ, stated in TCTL, the correctness of implementation, represented by UPTA model M can be verified by solving the satisfiability problem $M \models \varphi$.

When the correctness of an already implemented system needs to be verified it is often the case that there is not such a model M and the satisfiability of requirements properties can be established only by comparing the externally observable behaviours of the system itself and those specified in the requirements. Such weaker form of behavioral equivalence is characterized by so called *input-output conformance* (IOCO) relation and its timed versions. IOCO(I, S) between implementation I and specification S is generally the subject of model-based testing. The preliminaries and methods applicable for checking implementation and IOCO relations are presented and their complexity classes characterized in the following sub-sections.

2.3.1 TCTL Model Checking

The TCTL model-checking algorithm with UPTA is built upon the method of model checking TA. The task is to check for a given timed automaton M_{TA} and TCTL formula Φ whether $M_{TA} \models \Phi$. It is assumed that M_{TA} is non-zeno because time convergence can be proven independently. Since a TA

$$M_{TA} = \left\langle \overrightarrow{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Chan}, \mathit{Type} \right\rangle$$

includes real valued clocks (set Clocks) its explicit state model has infinite set of states. Therefore, the TCTL formula is checked instead on so called *region transition system* $RTS\,(M_{TA}, \Phi)$ that provides a finite quotient of the set of states of M_{TA} still preserving the properties of TCTL to be proved. The states of $RTS\,(M_{TA}, \Phi)$represent equivalence classes of states of TA s.t. all states of TA being in the same class satisfy same atomic clock constraints. So, the the problem $M_{TA} \models \Phi$? is reduced to checking whether an untimed version $\hat{\Phi}$ of Φ holds in $RTS\,(M_{TA}, \Phi)$, i.e.

$M_{TA} \models_{TCTL} \Phi$ iff $RTS\,(M_{TA}, \Phi) \models_{CTL} \hat{\Phi}$, where $\hat{\Phi}$ is a formula obtained from TCTL formula Φ using the mapping described in Algorithm2.1

Algorithm 2.1 Algorithm (basic idea)[4]

Input: Timed automaton M_{TA} and TCTL formula Φ
Output:$M_{TA} \models \Phi$

- $\hat{\Phi}$:= eliminate timing parameters from Φ
- *determine the equivalence classes under* $\cong$
  ($\cong$ denotes the equivalence used to obtain the quotient state space of $RTS\,(M_{TA}, \Phi)$)
- *construct the* $RTS\,(M_{TA}, \Phi)$
- *apply the standard CTL model checking algorithm[4] to check* $RTS\,(M_{TA}, \Phi) \models_{CTL} \hat{\Phi}$
- $M_{TA} \models_{TCTL} \Phi$ iff $RTS\,(M_{TA}, \Phi) \models_{CTL} \hat{\Phi}$

The procedure of *eliminating timing parameters* uses auxiliary clocks z to measure the elapse of time until the property Φ holds, i.e. to check some subformula Φ^J of Φ in state s where clock z is reset in s and Φ^J is checked when the value of z lies in the interval J.

Using the notion of auxiliary (atomic clock constraint related) clocks the following theorem shows the idea of mapping of $TCTL$ formula into timing free $TCTL$ formula.

Theorem 2.11. Let TA be timed automaton $M_{TA} = (L, l_0, C, A, E, I)$ *and* $\varphi ::= \Phi U^J \Psi$ a $TCTL$ formula over C and set $B\,(C)$ of atomic propositions over C. For clock $z \notin C$ let

$M'_{TA} = (L, l_0, C \cup \{z\}, A, E, I)$, for any state s of M_{TA} it holds that

1. $s \models_{TCTL} E\,(\Phi \cup \Psi)$ iff $s\,[0/z] \models_{TCTL} E\,(\Phi \vee \Psi) \cup ((z \in J) \wedge \Psi)$
2. $s \models_{TCTL} A(\Phi \cup \Psi)$ iff $s\,[0/z] \models_{TCTL} A\,(\Phi \vee \Psi) \cup ((z \in J) \wedge \Psi)$

where $s\,[[0/z]]$ is a state in M'_{TA}.

The proof of theorem can be found in [4], (pages 707-708).

The *equivalence classes under* $\cong$ determine the quotient states by letting $\langle l, \eta \rangle \cong \langle l', \eta' \rangle$ if $l = l'$ and $\eta \cong \eta$' such that:

- equivalent clock valuations should satisfy the same clock constraints that occur in M_{TA} and Φ : $\eta \cong \eta' \Rightarrow \eta \models g$ iff $\eta' \models g\ forall\ g \in ACC\,(M_{TA}) \cup ACC\,(\Phi)$ where $ACC\,(M_{TA})$ and $ACC\,(\Phi)$ denote the set of atomic clock constraints that occur in M_{TA} and Φ, respectively.

- Time-divergent paths starting from equivalent states should be "equivalent". This property guarantees that equivalent states satisfy the same path formulae.
- The number of equivalence classes under $\cong$ is finite.

Definition 2.12. Let TA M_{TA} be a timed automaton, Φ a TCTL formula (both over set C of clocks), and c_x the largest constant with which $x \in C$ is compared with in either M_{TA} or Φ. Clock valuations $\eta, \eta' \in Eval\,(M)$ are clock-equivalent, denoted $\eta \cong \eta'$ if and only if either

- for any $x \in C$ it holds that $\eta\,(x) > cx$ and $\eta\,(x) > c_x$, or
- for any $x, y \in C$ with $\eta\,(x)$, $\eta'\,(x) \leq c_x$ and $\eta\,(y)$, $\eta'\,(y) \leq c_y$ the following conditions hold:
  - $\lfloor \eta(x) \rfloor = \lfloor \eta'\,(x) \rfloor$ and $frac\,(\eta\,(x)) = 0$, iff $frac\,(\eta'\,(x)) = 0$
  - $frac\,(\eta\,(x)) \leq frac\,(\eta\,(y))$ iff $frac\,(\eta\,(x)) \leq frac\,(\eta\,(y))$.

As stated above the equivalence classes under $\cong$ are called *clock regions*.

The number of clock regions is bounded from below and above as follows [4]:

$$|C|! * \prod_{x\in C} c_x \leq |Eval\,(C)\,/_{\cong}| \leq |C|! * 2^{|C|-1} * \prod_{x\in C} (2c_x + 2)$$

where for the upper bound it is assumed that $c_x \geq 1$ for all $x \in C$.

The following definition determines the successor region of a region that is obtained by delaying.

Definition 2.13. Let $r, r' \in Eval\,(C)\,/_{\cong}$, where r' is the successor (clock) region of r, denoted $r = succ\,(r)$, if either

1. $r = r_\infty$ and $r = r'$, or
2. $r = r_\infty, r \neq r$ and
   $\forall \eta \in r$:$\exists d \in \mathbb{R}_{>0} : (\eta + d \in r')$ and $\forall 0 \leq d' \leq d : \eta + d' \in r \cup r'$.

Definition 2.14. The successor state region is defined as

$$succ\,(\langle l, r\rangle) = \langle l,\, succ\,(r)\rangle$$

Having defined the successor relation on regions we can generalize the definition of transition systems to Region Transition Systems.

Definition 2.15. (*Region Transition System*) Let $M_{TA} = (L, l_0, C, A, E, I)$ be a non-Zeno timed automaton and let Φ be a TCTL reachability formula. Then the region transition system of M_{TA} for Φ is defined as follows:
$RTS\,(M_{TA}, \Phi) = (S', l_0, C, A \cup \{\tau\}, E', I)$ where

· if S is the set of all states in $TS\,(M_{TA})$, then the state space of $RTS\,(M_{TA}, \Phi)$ is $S' = S/_{\cong} = \{[s]\,|s \in S\}$, the set of all state regions,

· $I' = \{[s]\,|\,s \in I\}$,

· $AP' = ACC\,(M) \cup ACC\,(\Phi) \cup AP$,

· $L'\,(\langle l, r\rangle) = L\,(l) \cup \{g \in AP' \setminus AP|r \vDash g\}$,

· The transition relation $\to'$ is defined by:

$$\frac{l \xrightarrow{g:\alpha,D} l' \wedge r \vDash g \wedge reset\ D\ in r \vDash Inv(l')}{<l,r> \xrightarrow{\alpha}{}' <l', reset\ D\ in\ r)}$$

and

$$\frac{r \vDash Inv(l) \wedge succ(r) \vDash Inv(l)}{<l,r> \xrightarrow{\tau}{}' <l, succ(r)>}$$

where

reset D in $r = \{reset\ D\ in\ \eta|\ \eta \in r\}$,
i.e.

resetting the clocks D in region r can be considered as a transition between state regions.

The successor region r' of a clock r, denoted by $r' = succ\,(r)$ is defined as for $r = r_\infty$ and $r = r'$, or $r \neq r_\infty$, $r \neq r'$ for all $\eta \in r : \exists d \in \mathbb{R}_{>0} : (\eta + d \in r')$ and $\forall 0 \leq d' \leq d : \eta + d' \in r \cup r'$, and the successor state region $succ\,(\langle l, r\rangle) = \langle l, succ\,(r)\rangle$.

Following theorem establishes the correctness of the model checking of TA via RTS.

Theorem 2.16. *For non-zeno M_{TA} and TCTL reachability formula Φ*

$$M_{TA} \models_{TCTL} \Phi\ iff\ RTS\,(M_{TA}, \Phi) \models_{CTL} \hat{\Phi}.$$

The proof of 2.16can be found in [4](pages 729-730).

Having now the correctness result for TCTL model checking we sketch the basic idea of TCTL model checking algorithm.

Assume the TCTL formula is of the form $E\left(\Phi \cup^J \Psi\right)$ with $J \neq [0, \infty)$. The algorithm 2.2 summarizes the main steps of TCTL model checking method.

Algorithm 2.2 Main steps of TCTL model checking[4].

Input: non-Zeno, timelock-free TA and TCTL formula Φ
Output: “yes” if TA $M_{TA} \models_{TCTL} \Phi$, “no” otherwise.

```
$R := RTS\left(M_{TA} \oplus z, \Phi\right)$   // with state space $S_{rts}$ and labeling $L_{rts}$
for all $i \le |\Phi|$ do
 for all $\Psi \in Sub\left(\Phi\right)$ with $|\Psi| = i$ do
  switch($\Psi$):
   $true$: $Sat_R\left(\Psi\right) := S_{rts}$;
   $a$: $Sat_R\left(\Psi\right) := \left\{s \in S_{rts} | a \in L_{rts}\left(s\right)\right\}$;
   $\Psi_1 \wedge \Psi_2$: $Sat_R\left(\Psi\right) := \left\{s \in S_{rts} | \left\{a_{\Psi_1}, a_{\Psi_2}\right\} \subseteq L_{rts}\left(s\right)\right\}$;
   $\neg\Psi'$ : $Sat_R\left(\Psi\right) := \left\{s \in S_{rts} | a_{\Psi'} \notin L_{rts}\left(s\right)\right\}$;
   $\exists\left(\Psi_1 U^J \Psi_2\right)$ :
     $Sat_R\left(\Psi\right) := Sat\left\{s \in S_{CTL}\left(\exists\left(\left(a_{\Psi_1} \vee a_{\Psi_2}\right) \cup \left(\left(z \in J\right) \wedge a_{\Psi_2}\right)\right)\right)\right\}$;
   $\forall\left(\Psi_1 U^J \Psi_2\right)$ :
     $Sat_R(\Psi) := Sat\left\{s \in S_{CTL}\left(\forall\left(\left(a_{\Psi_1} \vee a_{\Psi_2}\right) \cup \left(\left(z \in J\right) \wedge a_{\Psi_2}\right)\right)\right)\right\}$;
  end switch
  // add $a_\Psi$ to the labeling of all regions where $\Psi$ holds
  for all $s \in S_{rts}$ with $s\left\{z := 0\right\} \in Sat_R\left(\Psi\right)$ do $L_{rts}\left(s\right) := L_{rts}\left(s\right) \cup$
$\{a_\Psi\}$ od
 od
od
if $I_{rts} \subseteq Sat_R\left(\Psi\right)$ then
 return “yes”
else
 return “no”
fi
```

Having a model M_{TA}, by means of a recursive extraction of the subformuli Ψ in the parse tree of Φ, the set of states $Sat\left(\Psi\right) = \left\{[s] \in S_{RTS} | [s] \models \Psi\right\}$ is found where Ψ is satisfied. Consider the case where Ψ is a path formula $\Psi = E\left(a\, U^J\, b\right)$. A CTL model checker can be applied on the $RTS\left(M_{TA}, \Phi\right)$ and the CTL formula $\hat{\Psi} = E\left(a \cup \left(\left(z \in J\right) \wedge b\right)\right)$. Recall that $\hat{\Psi}$ is a CTL formula over the set of atomic propositions that occur in Φ, clock constraints in Φ and clock constraints on z (e.g. $z \in J$). Theorems above yield that all states $[s]$ in $RTS\left(M_{TA}, \Phi\right)$ are labeled with proposition Ψ where Ψ is satisfied. When all subformuli Ψ are checked for satisfiability, $M_{TA} \models_{TCTL} \Phi$ if and only if all initial states on the $RTS\left(M_{TA}, \Phi\right)$ are labeled with Φ. In case M_{TA} refutes Φ, the path returned is a counterexample of the satisfaction query.

We conclude the subsection with time complexity result that establishes the feasibility constraints and gives some guidelines to be taken into account when

TCTL model checking is used for practical verification tasks.

Theorem 2.17. Time complexity of $TCTL$ model checking[4]
For timed automaton M_{TA} and TCTL formula Φ, the $TCTL$ model-checking problem $M_{TA} \models_{TCTL} \Phi$ can be determined in time $O((N+K) \cdot |\Phi|)$, where N and K are the number of states and transitions in the region transition system $RTS(M_{TA}, \Phi)$, respectively.

The worst-case time complexity of TCTL model checking is linear in the size of Φ due to the recursive descent over the parse tree of Φ and in the size of the $RTS(M_{TA}, \Phi)$. As the state-space size of $RTS(M_{TA}, \Phi)$ grows exponentially in the number of clocks (and maximal constants c_x), the time complexity of TCTL model checking is exponential in the number of clocks. As proved in [48] for worst case complexity the $TCTL$ model checking problem is PSPACE-complete.

2.3.2 IOCO testing of timed systems

The model-based testing theory makes an assumption that system under test (SUT) can be represented by some formal model (implementation model) that is system representation on appropriate level of abstraction. This assumption is referred to as a *test hypothesis.* Given two models - the specification and the system model, model-based testing theory studies the problem of how these two are relate to each-other. Specifically, the testing methods applied in the thesis rely on *the input-output conformance* (IOCO) relation. IOCO theory reasons about black-box conformance testing[1]. We say that an implementation I IOCO-conforms a specification S (denoted by $I \sqsubseteq_{ioco} S$) when at any point in execution it can handle at least as many inputs as the specification, and at most as many outputs. The one exception to this rule is that implementation is not allowed to be *quiescent* (i.e., not provide any output) when the specification prescribes at least one possible output [49]. The semantics of IOCO relation and related testing theory [50] is originally formulated using labeled transition systems (LTS) and input/output transition systems (IOTS).

Definition 2.18. Labelled transition system (LTS) *[51, 52].*

A labeled transition system is a *4-tuple* $\langle S, L, T, s_0 \rangle$ *where:*

· S is a countable, non-empty set of states,

· L is a countable set of labels,

· $T \subseteq \{S \times (L \cup \tau) \times S \mid \tau \notin L\}$ is the transition relation where τ is the unobservable internal action and

· $s_0 \in S$ is the initial state.

According to this definition the transitions can be labeled either by elements of L which are called *observable actions* or by distinguished internal (unobservable) *silent actions* τ. Given L is the set of all observable actions then the set of all labelled transition systems that could be constructed on L is denoted $\mathcal{LTS}$ and the set of finite sequences of actions over L is denoted by L^*[50]. Usually $\tau \notin L$ but if the label set is expected to contain the τ-transition then it is denoted by L_τ where $\tau \in L_\tau$. $\mathcal{LTS}$ are assumed to be *strongly converging* with regard to the τ-transition. This means there are no infinite sequence of internal τ-actions, i.e. there is possible to perform a concatenation ($\sigma = a_1 \cdot ... \cdot a_n$) of actions and reach some new observable state. In other words there cannot be infinite unobservable loops for the model to be stuck in and not proceed and/or produce any observable output. A *computation* is a (finite) sequence of transitions: $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n$. A *trace* σ is a sequence of observable actions of a computation. ϵ is an *empty sequence* that is used to denote the absence of observable actions in the sequence. The finite set of all sequences over a set of actions *Act* denoted by *traces* (s) captures all observable traces in L^* that a LTS could possibly perform.

Definition 2.19. $\mathcal{LTS}$ is strongly converging if there is no infinite sequence of internal actions[50].

$$a \in L \cup \tau, \sigma = a_1 \cdot \ldots \cdot a_n \quad s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s' \Leftrightarrow s \xrightarrow{\sigma} s' \tag{2.2}$$

As referred above another important notion in the ioco theory is *quiescence*, which characterises system states that will not produce any output response without the provision of a new input stimulus. In the setting of input/output systems one generally assumes the systems to be also *input-enabled*: all input actions are always possible in all system states, i.e., input can never be refused [53]. This means that a system with input/output actions is never formally deadlocked, since one can always execute further (input) actions. In this context quiescence becomes the meaningful representation of unproductive behaviour, comparable to deadlocked behaviour in the case of synchronously communicating systems. For further formal definition of these properties we sum up the notations describing LTS in 2.4 Let the actions to be noted as lowercase latin alphabet, e.g. $a, a_1, \ldots, a_n$ and sequences of actions in greek alphabet $\mu, \sigma, \sigma_1, \ldots, \sigma_n$.

When describing transition systems in general, the notation $s \xrightarrow{\quad a \quad} s'$ states that there exists a state $\exists s \in S$ such that by performing an action $a \in L$ we reach $s' \in S$. By noting $s \xrightarrow{a_1 \cdot \ldots \cdot a_n}$ we say that the given LTS is able to perform a concatenation of actions starting from its state $s \in S$. Conversely, the negated notation $s \overset{a_1 \cdot \ldots \cdot a_n}{\not\longrightarrow}$ tells that no matter the state, these actions cannot be performed. $s \xrightarrow{\sigma}$ denotes the ability to perform a (legal) sequence σ and reach some state. $s \xrightarrow{a} s'$ denotes the ability to drive the system from whatever state it is currently in, to accept the given action and then drive the system further to the desired (internal) state s'. $s\,after\,(\sigma)$ denotes the possible set of states a system could end up in after performing a sequence σ.

$s \xrightarrow{a} s'$	$\Leftrightarrow_{def}$	$(s, a, s') \in T$
$s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s'$	$\Leftrightarrow_{def}$	$\exists s_0, ..., s_n : s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n = s'$
$s \xrightarrow{a_1 \cdot \ldots \cdot a_n}$	$\Leftrightarrow_{def}$	$\exists s' : s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s'$
$s \not\xrightarrow{a_1 \cdot \ldots \cdot a_n}$	$\Leftrightarrow_{def}$	$\neg \exists s' : s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s'$
$s \xRightarrow{\sigma}$	$\Leftrightarrow_{def}$	$\exists s' : s \overset{\sigma}{\Rightarrow} s'$
$s \not\xRightarrow{\sigma}$	$\Leftrightarrow_{def}$	$\neg \exists s' : s \overset{\sigma}{\Rightarrow} s'$
$s \xRightarrow{\epsilon} s'$	$\Leftrightarrow_{def}$	$s = s'$ or $s \xrightarrow{\tau \cdot \tau \cdot \ldots \cdot \tau} s'$
$s \xRightarrow{a} s'$	$\Leftrightarrow_{def}$	$\exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s'$, $a \in L$
$s \xRightarrow{a_1 \cdot \ldots \cdot a_n} s'$	$\Leftrightarrow_{def}$	$\exists s_0 \ldots s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s'$
$s \xrightarrow{a_1 \cdot \ldots \cdot a_n}$	$\Leftrightarrow_{def}$	$\exists s' : s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s'$
$s \not\xRightarrow{a_1 \cdot \ldots \cdot a_n}$	$\Leftrightarrow_{def}$	$\neg \exists s' : s \xRightarrow{a_1 \cdot \ldots \cdot a_n} s'$
$traces\,(s)$	$\Leftrightarrow_{def}$	$\sigma \in L^* : s \overset{\sigma}{\Rightarrow}$
$s\,after\,(\sigma)$	$\Leftrightarrow_{def}$	$s' : s \overset{\sigma}{\Rightarrow} s'$

Figure 2.4: LTS notation.

Definition 2.20. Non-determinism

Let action $a \in T$ denote a transition label and states $s, s' \in S$. We write $s \xrightarrow{a} s'$ if there is a state s so that after an action a the state is s'. Non-determinism is when a LTS being in state s_1 can execute an action sequence $a_1 \cdot ... \cdot a_n$ so that either $s_1 \xrightarrow{a_1 \cdot \ldots \cdot a_n} s_2$ or $s_1 \xrightarrow{a_1 \cdot \ldots \cdot a_n} s_3$ where $s_2 \neq s_3$. See figure 2.5.

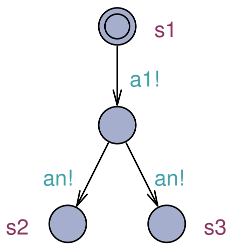


Figure 2.5: Nondeterministic automaton.

Definition 2.21. *The parallel composition* $S_1 \| S_2$ *of two LTS* S_1 and S_2 is a tuple $\langle S_1 \times S_2, (s_{01}, s_{02}), L_1 \cup L_2, T \rangle$ where *transition relation* T of $S_1 \| S_2$ is defined as:

$$\frac{s_1 \xrightarrow{a} s_1'}{(s_1, s_2) \xrightarrow{a} (s_1', s_2)} \quad \frac{s_2 \xrightarrow{a} s_2'}{(s_1, s_2) \xrightarrow{a} (s_1, s_2')} \quad \frac{s_1 \xrightarrow{a} s_1' \quad s_1 \xrightarrow{a} s_1'}{(s_1, s_2) \xrightarrow{\tau} (s_1', s_2')} \tag{2.3}$$

Figure 2.6 illustrates parallel composition as a directed graph.

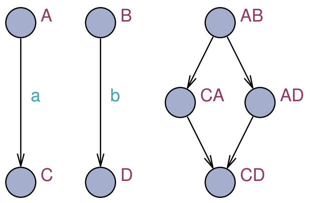


Figure 2.6: *LTS* parallel composition.

Based on work by Tretmans[50, 54] DeNicola & Hennessey[55] introduced the notion of *testing preorder* $\leq_{te}$. If S can perform a *trace* σ then S can also perform any "initial part" of σ - called the *prefix* of σ. This is formalized by the (*prefix*) *relation* $\preceq \subseteq L^* \times L^*$.

Definition 2.22. *Preorder*[50].

1. *traces* $(S) =_{def} \left\{ a \in L^* | S \xRightarrow{\quad a \quad} \right\}$
2. A trace a_1 is a *prefix* of a_2, $a_1 \preceq a_2$ if $\exists a' : a_1 \cdot a' = a_2$

Proposition 2.23. *Testing preorder.*
Let I and S be LTS describing implementation and specification respectively.

$$I \sqsubseteq_{te} S \textbf{ iff } \forall \sigma \in L^*, \forall A \subseteq L :$$
$$I \textbf{ after } \sigma \textbf{ refuses } A \textbf{ implies } S \textbf{ after } \sigma \textbf{ refuses } A$$

where I **after** σ **refuses** $A \Leftrightarrow_{def} \exists p' : p \overset{\sigma}{\Rightarrow} I'$ and $\forall a \in A : I' \overset{a}{\nRightarrow}$ and similarly to s.

Testing preorder $\sqsubseteq_{te}$ *says that whatever observable behaviour the implementation I can exhibit, the specification S can also exhibit and not vice versa. A* stronger *property* testing equivalence *says the behaviours of I and S match exactly.*

Definition 2.24. *Testing equivalence*[55, 56]

$$S =_{te} I \Leftrightarrow_{def} I \sqsubseteq_{te} S \wedge S \sqsubseteq_{te} I \tag{2.4}$$

According to [55] the *implementation relation* used in testing should base on the observations that an external observer can make on both the system and the specification. If the external observer o in some observer class O (e.g. the classes O can be partitioned by their properties) cannot distinguish between the systems they are considered equivalent and the implementation correctly implements the specification.

$$I \sqsubseteq_{obs} S \Leftrightarrow_{def} \forall o \in O : obs\,(o, I) = obs\,(o, S) \tag{2.5}$$

By choosing different observer classes O we can define various interpretations of the implementation relation $\sqsubseteq$, e.g. in MBT literature one can find relations such as strong/weak bisimulation equivalence, failure equivalence & preorder, failure trace equivalence, refusal testing, etc. In the rest of thesis we focus only on IOCO relation and its timed extensions. To meet the realistic testing conditions we introduce a weaker notion of enabledness and define the input-output transition systems (IOTS) before returning to the conformance relation.

Input-output transition systems While IOCO relation between LTS-s is based on *strong input enabledness* as in ([57] $\forall a \in L_I : p \xrightarrow{a}$), the IOTS allow input enabling via internal (silent) transitions - *weak input enabledness* ($\forall a \in L_I : p' \xrightarrow{a}$) [54]. Thus, we study input-output testing relation $\sqsubseteq_{iot} \subseteq \mathcal{IOTS}\,(L_I, L_U) \times \mathcal{LTS}\,(L_I \cup L_U)$, where L_U includes both output and silent actions.

Definition 2.25. *Input/Output transition system* ($\mathcal{IOTS}$) *is a* $\mathcal{LTS}$ *where* the label set L (actions) are split into input actions L_I and output actions L_U and all input actions are always enabled (weak input enabled). If (process) $p \in \mathcal{IOTS}$ then

$$p \overset{\sigma}{\Longrightarrow} p' \textit{ implies } \forall a \in L_I : \quad p' \overset{a}{\Longrightarrow} \tag{2.6}$$

Quiescent state and trace

- Quiescent state is noted by $\delta(s)$, where $\forall \sigma \in L_U \cup \{\tau\} : s \overset{\sigma}{\not\rightarrow}$
- Quiescent trace of s is a trace σ that leads to a quiescent state: $\exists s' \in (p\ \mathbf{after}\ \sigma) : \delta\,(s')$ (where $s \in \mathcal{LTS}\,(L_I \cup L_U)$)

Definition 2.26. IOCO is defined by

$$I \sqsubseteq_{ioco} S \Leftrightarrow_{def} \forall \sigma \in \mathbf{traces}\,(s) : \mathbf{out}\,(I\ \mathbf{after}\ \sigma) \subseteq \mathbf{out}\,(S\ \mathbf{after}\ \sigma) \tag{2.7}$$

where $I \in \mathcal{IOTS}\,(L_I, L_U)$, $S \in \mathcal{LTS}\,(L_I \cup L_U)$

The conformance relation in timed systems is defined using timed traces of timed IOTS (TIOTS) and of timed LTS ($TLTS$).

Definition 2.27. Timed Input Output Transition System *(TIOTS)[33] S is a* tuple (S, s_0, L_I, L_U, T), where

· S is a set of states, $s_0 \in S$,

· L_I is the input and L_U the output labels (actions),

· $T \subseteq S \times (L_I \cup L_U \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the constraints of

– *time determinism* (if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$then $s' = s''$)

– *time additivity* (if $s \xrightarrow{d_1} s'$ and $s' \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$) and

– *zero-delay* ($\forall s \in S.\, s \xrightarrow{0} s$)

where $d, d_1, d_2 \in \mathbb{R}_{\geq 0}$ denote time delays (non-negative real numbers).

Since TLTS differs from TIOTS only by the set of action labels (extended with silent action τ) and τ-labeled transitions we refer back to given TIOTS definition and proceed with the parallel composition of TIOTS and timed conformance relation.

Definition 2.28. TIOTS parallel composition

Given $S = (S, s_0, L_I, L_U, T)$ and $\mathcal{E} = (E, e_0, L_U, L_I, T)$ are TIOTS and the set of input (output) actions in $\mathcal{E}$ and output (input) actions in S are with identical action names but with different suffices ("?" and "!") then the parallel composition is

$$S \| \mathcal{E} = (S \times E, (s_0, e_0), L_I, L_U, T) \tag{2.8}$$

where transition relation T is defined as

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{(s,e) \xrightarrow{a} (s',e')} \quad \frac{s \xrightarrow{\tau} s'}{(s,e) \xrightarrow{\tau} (s',e)} \quad \frac{e \xrightarrow{\tau} e'}{(s,e) \xrightarrow{\tau} (s,e')} \quad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{(s,e) \xrightarrow{d} (s',e')} \tag{2.9}$$

Note that the timed conformance relation extends naturally to cases where $I = I_1||I_2 \in \mathit{TIOTS}\ (L_I, L_U)$ and $S = S_1||S_2 \in \mathit{TLTS}\ (L_I, L_U)$ [50, 54].

Definition 2.29. *Timed traces.* Let $s \in S$ be a state in TLTS, specification

Definition 2.30.

$$S \in TLTS\left(L_{in} \cup L_{out} \cup \{\delta\}\right)$$

where $\delta \in R \geq 0$, and implementation I be an $TIOTS\left(L_{in} \cup L_{out}\right)$. Then,

- $\cdot$ $ttraces\,(s) =_{def} \sigma \in \left(L_{in} \cup L_{out} \cup \{\delta\}\right)^* |s \overset{\sigma}{\Longrightarrow}$
- $\cdot$ $out\,(s) =_{def} \left\{a \in L|\; s \overset{a}{\Longrightarrow}\right\}$
- $\cdot$ $out\,(S) =_{def} \bigcup \{out\,(s)\,|s \in S\}$
- $\cdot$ $I \sqsubseteq_{tioco} S \Leftrightarrow_{def} \forall\sigma \in ttraces\,(spec) : \forall\sigma \in ttraces\,(s) :$ $\mathbf{out}\;(I\;\mathbf{after}\;\sigma) \subseteq \mathbf{out}\;(S\;\mathbf{after}\;\sigma)$

Up to now the parallel composition of TIOTSs or TLTSs has been assumed to be synchronous meaning that the input action has been executed simultaneously with respective output action and vice a versa. In remote testing the synchronous communication between the tester and implementation under test (IUT) cannot be implemented due to the communication delays. Asynchronous communication introduces two additional assumptions about the communication in the TA parallel composition model. The model is centered around a $2FIFO\,(\boxtimes, \Delta)$ architecture that consists of:

1. One first-in-first-out (FIFO) buffer for each direction of the communication between the communicating automata.

2. A communication latency bounded by Δ. The symbol $\boxtimes$ stands for either $\leq$ or $=$. The IOCO relation for asynchronous parallel composition $||_{async}$ is defined in terms of *asynchronous trace semantics [34].*

Definition 2.31. *Asynchronous semantics for TIOA.*

Let $A = \left\langle L, l^0, I, O, \varnothing, X, E\right\rangle$ be a $TIOA\,(I, O)$ with no silent action. Let $\bowtie \in \{\ \leq\ =\ \}$ and $\triangle \in \mathbb{N}$. The *asynchronous semantics* for A is an IOTTS $(I, O, \Lambda_{I\cup O})$,

$$\begin{aligned}\langle|\;A\;|\rangle^{\bowtie\Delta} \quad = \quad & < \left(L \times \mathbb{R}^X_{\geq 0}\right) \times \left(\mathbb{R}_{\geq 0} \times (I \cup O)\right)^* \\ & \times \left(\mathbb{R}_{\geq 0} \times (I \cup O)\right)^*, (l_0, O)\,, (I, O, \Lambda_{I\cup O})\,, M_{\bowtie\Delta} >\end{aligned}$$

where $\Lambda_{I\cup O} = \{\tau_a | a \in I \cup O\}$ is the set of silent actions. An asynchronous state is of the form $((l, v)\,, p, q)$ where p and q are input and output queues respectively. The set of asynchronous moves, $M_{\bowtie\Delta}$ is defined by the following five rules:

$$(l,v),p,q)\xrightarrow{?a}((l,v),p.(0\cdot ?a),q \qquad (r1)$$

$$\frac{((l,v),(\delta\cdot ?a).p,q))\xrightarrow{\tau a}((l',v[Y:=0]),p,q}{l\xrightarrow{g,?a,Y}l'\wedge v\models g\wedge\delta\bowtie\Delta} \qquad (r2)$$

$$((l,v),p,q))\xrightarrow{t}((l,v+t),p+t,q+t) \qquad (r3)$$

$$\frac{((l,v),p,(\delta\cdot !b).q))\xrightarrow{\tau !b}((l,v,p,q}{\delta\bowtie\Delta} \qquad (r4)$$

$$\frac{((l,v),p,q))\xrightarrow{\tau_b}((l',v[Y:=0]),p,q.(0!b)}{l\xrightarrow{g,!b,Y}l'\wedge v\models g} \qquad (r5)$$

Asynchronous timed traces are remote observations of local timed traces at communicating automata. The execution order of actions may differ from the observation order: this happens when inputs and outputs interleave in the communication channels. We characterize remote observations that may lead to action interleaving by introducing the notion of Δ-*testability[34].*

Definition 2.32. $\triangle$-*testability*

Let $A \in TIOA\ (I,O)$ and $\sigma \in TTraces\ (A)$ such that $\sigma = (t_i \cdot a_i)_{i=1..n}\, t_{n+1}$. The timed trace σ is $\triangle - testable$ if,

- either $n = 0$,
- or $(t_i \cdot a_i)_{i=1..n-1}$ is $\triangle$-testable and $a_n \in O$,
- or $(t_i \cdot a_i)_{i=1..n-1}$ is $\triangle$-testable and if $a_n \in I$, then for every $t_b \in \mathbb{R}_{\geq 0}$, every $b \in O$, and every $k = [1 \ldots n-1]$ such that $!b \in out\ (\llbracket A \rrbracket\ after\ \sigma\,[1 \ldots k] \cdot t_b)$ it holds that $t_n - t_b > 2\triangle$.

A is $\triangle$-testable if every $\sigma \in TTraces\ (A)$ is $\triangle$-testable.

Proposition 2.33. Let A be a $TIOA\ (I,O)$. Let $s, \rho \in Seq\left(\langle|A|\rangle^{\bowtie\Delta}\right)$ such that $s_0 \xrightarrow{\rho} s$. A is $\triangle$-testable iff $p\,(s)$ is non empty implies $q\,(s)$ is empty.

According to Proposition 2.33, Δ-testability implies that at most one queue is non empty at every reachable state. However, Δ-testability does not guarantee that the sizes of the queues are bounded. A fast environment can increase the size of the input queue by sending repetitively the inputs faster than the latency. It is shown in [34] that Δ-testable specifications are controllable. Indeed, having the condition that output response to each input stimulus arrives before the next input is given the outputs transmitted earlier are received before the transmission of new inputs. Thus, each observed output depends on input transmitted earlier and the specification is controllable. Given the maximum signal propagation delay is Δ the delay between the two consecutive test inputs must be strictly greater than 2Δ. In brief, Δ-testability criterion takes advantage of the timing information that is not available in untimed models.

Another important corollary of delta-testability criterion is that if if the specification is Δ-testable then, the asynchronous execution of the synthesized test cases is as simple as the synchronous execution, the TIOCO conformance is preserved and the tester can control the test. This result provides one of main motivations of current thesis to implement the distributed test execution environment that can implement the criterion of Δ-testability.

2.4 Summary

This chapter presented the theoretical foundations of model-based control and testing of time constrained CPS described in Chapter 1. The alternatives of modeling and verification formalisms were discussed from pragmatics, taxonomy, and expressiveness point of view and their relevance for model based control and testing was motivated. The semantics of Uppaal timed automata as one of the formalisms that meets the requirements outlined in Chapter 1 was introduced and relying on this the rigorous definition of control and testing related notions such as conformance, responsiveness, observability, controllability were defined. Thereafter, the the algorithmic methods for timed CTL logic model checking and model based timed conformance testing were specified based on the syntactic and semantic notations of LTS and Uppaal timed automata. These notations as well as algorithmic methods will be used further throughout the thesis.

3 Provably correct development of delta-tests

3.1 Chapter overview

The goal of chapter is to elaborate the provably correct workflow of developing models for MBT. We study and formulate the verification conditions necessary to assure the correctness of development increments for all major development steps. It is shown how verification conditions can be discharged by using model checking and syntactic analysis techniques. Also model transformations necessary for simplification of analysis steps are outline. The results introduced in this chapter are based on the publication:

1. Vain, Jüri; Anier, Aivo; Halling, Evelin (2014). Provably correct test development for timed systems. Databases and Information Systems VIII : Selected Papers from the Eleventh International Baltic Conference, Baltic DB&IS 2014. Ed. Haav, Hele-Mai; Kalja, Ahto; Robal, Tarmo. Amsterdam: IOS Press, 289–302. (Frontiers in Artificial Intelligence and Applications; 270).

3.2 Introduction

The provably correct MBT process introduced in Section 1.3 (Figure 1.3) comprises test development steps (modelling the system under test, specifying the test purposes, generating the tests and executing them against IUT) that alternate with verification steps. In this chapter, the verification of test development steps is described and how the test results are made trustable throughout the testing process. We focus on model-based online testing of systems with timing constraints capitalizing on the correctness of the test suite through test development and execution process.

In conformance testing the IUT is considered as a black-box, i.e., only the inputs and outputs of the system are externally controllable and observable respectively. The aim of black-box conformance testing according to [54] is to check if the behaviour observable on system interface conforms to a given requirements specification. During testing a tester executes selected test cases on an IUT and emits a test verdict (pass, fail, inconclusive). The verdict shows correctness in the sense of input-output conformance relation (IOCO) between IUT and the specification. The behaviour of a IOCO-correct implementation should respect after some observations following restrictions:

(i) the outputs produced by IUT should be the same as allowed in the specification;

(ii) if a quiescent state (a situation where the system can not evolve without an input from the environment [58] is reached in IUT, this should also be the case in the specification;

(iii) any time an input is possible in the specification, this should also be the case in the implementation.

The set of tests that forms a *test suite* is structured into *test cases*, each addressing some specific test purpose. In MBT, the test cases are generated from formal models that specify the expected behaviour of the IUT and from the *coverage criteria* that constrain the behaviour defined in IUT model with only those addressed by the test purpose. In our approach Uppaal Timed Automata (UPTA) [5] are used as a formalism for modelling IUT behaviour. This choice is motivated by the need to test the IUT with timing constraints so that the impact of propagation delays between the IUT and the tester can be taken into account when the test cases are generated and executed against realtime systems.

Another important aspect that needs to be addressed in testing CPS is *functional non-determinism* of the IUT behaviour with respect to test inputs. For nondeterministic systems only *online testing* (generating test stimuli *on-the-fly*) is applicable in contrast to that of deterministic systems where test sequences can be generated *offline*. Second source of non-determinism in remote testing of real-time systems is communication latency between the tester and the IUT that may lead to interleaving of inputs and outputs. This affects the generation of inputs for the IUT and the observation of outputs that may trigger a wrong test verdict. This problem has been described in [34], where the Δ-testability criterion (Δ describes the communication latency) has been proposed. The Δ -testability criterion ensures that wrong input/output interleaving never occurs.

3.3 Correctness of IUT Models

3.3.1 Modelling Timing Aspects of IUT

For automated testing of input-output conformance of systems with time constraints we initially restrict ourselves with a subset of UPTA that simplifies IUT model construction for timing aspects and will be extended later to capture full expressive power of UPTA. Namely, we use a subset of UPTA where the data variables, their updates and transition guards on data variables are abstracted away. We use the clock variables only and the conditions expressed by clocks and synchronization labels (channels). An elementary modelling pattern for representing IUT behaviour and timing constraints is *Action pattern* (or simply Action) depicted in Figure 3.1.

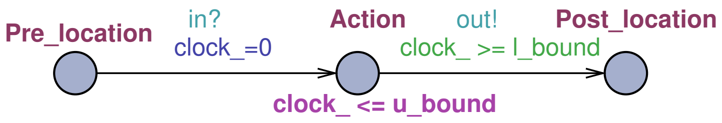


Figure 3.1: Elementary modelling fragment "Action".

An Action models a program fragment execution on a given level of abstraction as one *atomic step*. The Action is triggered by input event and it responds with output event within some bounded time interval (*response time*). The IUT input events (*stimuli* in the testing context) are generated by Tester, and the output events (IUT *responses*) are to make the reactions of IUT observable to Tester. In UPTA, the interaction between IUT and Tester is modelled with *channels* that link synchronous input/output events.

The major timing constraint we represent in IUT model is duration of the *Action*. To make the specification of durations more realistic we represent it as a closed interval $[l\_bound, u\_bound]$, where l_bound and u_bound denote lower and upper bound respectively. The duration interval $[l\_bound, u\_bound]$ can be expressed in UPTA as a pair of predicates on clocks as shown in Figure 3.1. The clock reset $clock = 0$ on the edge $(Pre\_location \rightarrow Action)$ makes the time constraint specification *local* to the *Action* and independent from the clock value accumulated during earlier execution steps. The clock invariant $clock\_ \leq u\_bound$ of location *Action* forces the *Action* to terminate latest at time instant u_bound after the clock reset and guard $clock\_ \geq l\_bound$ on the edge $Action \rightarrow Post\_location$ defines the earliest time instant (w.r.t. clock reset) when the outgoing transition of *Action* can be executed.

From tester's point of view IUT has two types of locations: *passive* and *active*. In passive locations IUT is waiting for test stimuli and in active locations IUT chooses its next moves, i.e. presumably it can stay in that location as long as specified by location invariant. The location can be left when the guard of outgoing transition $Action \rightarrow Post\_location$ evaluates to *true*. In Figure 3.1, the locations $Pre\_location$ and $Post\_location$ are passive while *Action* is an active location.

We compose the IUT models from Action pattern using *sequential* and *alternative composition*.

Definition 3.1. Composition of Action patterns

Let F_i and F_j be UPTA fragments composed of Action patterns including elementary Action with pre-locations l_i^{pre} ,l_j^{pre} and post-locations l_i^{post} ,l_j^{post} respectively, their composition is the union of elements of both fragments satisfying following conditions:

· sequential composition$F_i; F_j$ is UPTA fragment where $l_i^{post} = l_j^{pre}$;

· alternative composition $F_i + F_j$ is UPTA fragment where $l_i^{pre} = l_j^{pre}$ and $l_i^{post} = l_j^{post}$.

The test generation method of [59] relies on the notion of well-formedness of the IUT model according to the following inductive definition.

Definition 3.2. Well-formedness of IUT models

· atomic Action pattern is well-formed;

· sequential composition of well-formed patterns is well-formed;

· alternative composition of well-formed patterns is well-formed if the output labels are distinguishable.

Proposition 3.3. *Any UPTA model M with non-negative time constraints and synchronization channels that does not include state variables can be transformed to bi-similar well-formed representation $wf(M)$.*

The model transformation to well-formed representation is based on the idea that for those UPTA elements that do not match with the Definition 3.2, auxiliary pre-, and post-locations and internal ε-transition are added, that do not alter the i/o behaviour of the original model. The unbounded waiting of processes in parallel composition is avoided due to the well-formedness condition. Namely, by Action pattern construction at least one of the source locations of synchronized edges has delay upper bound specified in its invariant.

For representing internal actions that are not triggered by external events (their incoming edge is ε-labelled) we restrict the class of pre-locations with type *committed*. In fact, the subclass of models transformable to well-formed is broader than given by Definition 3.2, including also UPTA that have data variable updates, but in general well-formedness does not extend to models that include guards on data variables.

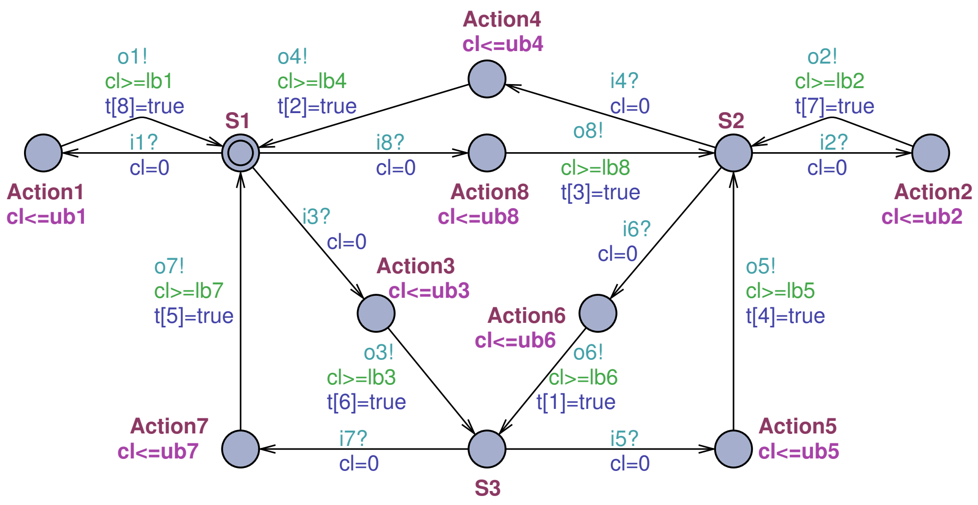


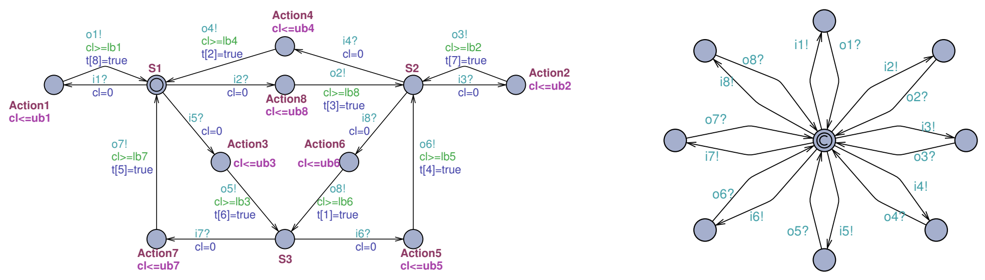
Figure 3.2: An example of well-formed IUT model.

In the rest of this chapter, for test generation we assume that M^{IUT} is well-formed and denote these models by $wf(M^{IUT})$. An example of the well-formed model we use throughout the chapter is depicted in Figure 3.2.

3.3.2 Correctness Conditions of IUT Models

The test generation method introduced in [59] and developed further for EFSM models in [60] assumes that the IUT model is *connected*, *input enabled* (i.e. also *input complete*), *output observable* and *strongly responsive*. In the following we demonstrate how the validity of these properties usually formulated for IOTS (Input-Output Transition System) models can be verified for well-formed UPTA models (see Definition 3.2).

Connected Control Structure and Output Observability We say that UPTA model is *connected* in the sense that there is an executable path from any location to any other location. Since the IUT model represents an open system that is interacting with its environment we need for verification by model checking a *nonrestrictive environment* model. According to [61] such an environment model has the role of *canonical tester*. Canonical tester provides test stimuli and receives test responses in any possible order the IUT model can interact with its environment. A canonical tester can be easily generated for well-formed models according to the pattern depicted in Figure 3.3b (this is canonical tester for the IUT model shown in Figure 3.3a). In fact, the canonical tester model itself is well-formed because it introduces zero delay when composed in parallel with Action pattern. Its central location is of type C that means instant response to IUT outputs.



(a) IUT (b) Canonical tester.

Figure 3.3: Synchronous parallel composition of IUT and canonical tester models.

The canonical tester composed with IUT model does not have guard conditions on edges and it implements the "random walk" test strategy. This strategy is useful in endurance testing but it is very inefficient when functionally or structurally constrained test cases need to be generated for large systems. Having synchronous parallel composition of IUT and the timed canonical tester (shown in Figure 3.3) the connectedness of IUT can be model checked with query $A[]$ *not deadlock* that expresses the absence of deadlocks in interaction between IUT and canonical tester.

The *output observability* condition means that all state transitions of the IUT model are observable and identifiable by the outputs generated by these transitions. The output observability is ensured by the definition of well-formedness of the IUT model where each input event and Action location is followed by the edge that generates a local (i.e. unique for outgoing edges of the location) output event.

Input Enabledness. Input enabledness of IUT models means that blocking of IUT due to irrelevant test input never occurs. That implicitly presumes input completeness. A naive way of implementing input enabledness in IUT models is introducing self-looping transitions with input labels that are not represented on other transitions that share the same source state. That makes IUT modelling tedious and leads to the exponential increase of its size. Alternatively, when relying on the notion of observational equivalence one can approximate the input enabledness in UPTA by exploiting the semantics of synchronizing channels and encoding input symbols as boolean variables $I_1 \ldots I_n \in \Sigma^{in}$. Then the pre-location of the Action pattern (see Figure 3.2) needs to be modified by applying following transformation:

· assume there are k outgoing edges from pre-location l_i^{pre} of action A_i, each of these edges e_j is labeled with one input symbol $I_j, j = 1, k$ from the input alphabet $\Sigma^{in}(M^{IUT})$;

· add a self-loop edge (l_i^{pre}, l_i^{pre}) that models acceptance of all inputs in $\Sigma^{in}(M^{IUT})$ except $I_j, j = 1, k$. To implement this construct we specify the guard of the auxiliary edge (l_i^{pre}, l_i^{pre}) with boolean expression: $not\left(\bigvee_{j=1,k} I_j\right)$.

Provided the branching factor $\mathcal{B}$ of the edges that are outgoing from l_i^{pre} is, as a rule, substantially smaller than the size $|\Sigma^{in}\left(M^{IUT}\right)|$ of input alphabet, we can save $|\Sigma^{in}\left(M^{IUT}\right)| - \mathcal{B}\left(l_i^{pre}\right)$ edges at each pre-location of the Action patterns. Note that due to the wf-construction rules the number of pre-locations never exceeds the number of actions in the model. That is due to alternative composition that merges pre-locations of the composition. A fragment of alternative composition accepting all inputs in $|\Sigma^{in}(M^{IUT})|$ is depicted in Figure 3.4 (time constraints are ignored here for clarity). Symbols $I1$ and $I2$ in the figure denote predicates $Input == i1$ and $Input == i2$ respectively.

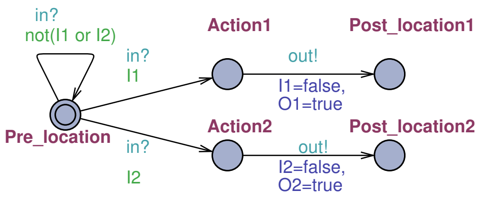


Figure 3.4: Input-enabled model fragment.

Strong Responsiveness Strong responsiveness (SR) means that there is no reachable *livelock* (a loop that includes only ε transitions) in the IUT model. In other words, the model should always enter the *quiescent state* after finite number of steps. Since transforming M^{IUT} to $wf\left(M^{IUT}\right)$ does not eliminate ε transitions there is no guarantee that $wf\left(M^{IUT}\right)$ is strongly responsive by construction (it is a built-in feature of the Action pattern). To verify the SR property of arbitrary M^{IUT} we apply Algorithm 3.1.

Algorithm 3.1 Checking strong responsiveness.

(i) According to the Action pattern in Figure 3.4 the M^{IUT} input events are encoded by means of channel $in?$ and a boolean variable I_i that represents the condition that $input = I_i$. Since input occurrence in Uppaal models can be checked only as a property holding in a location, we have to keep the input value indicating that the predicate is $true$ in the destination location of the edge that is labelled with given event and reset to$false$ immediately when leaving this location. For same reason the ε-transitions need to be labeled with update $EPS = true$ and following output edge with update $EPS = false$.

(ii) Reduce the model by removing all the edges and locations that are not involved in the traces of model checking query: $l_0 \models E[]\ EPS$, where l_0 denotes initial location of M^{IUT}. The query checks if any ε-transition is reachable from l_0 (that is necessary condition for violating SR-property).

(iii) Remove all non ε-transitions and locations that remain isolated thereafter.

(iv) Remove recursively all locations that do not have incoming edges (their outgoing edges will be deleted with them).

(v) After reaching the fixed point of recursion of step IV, check if the remaining part of model is empty. If yes then conclude that M^{IUT} is strongly responsive, otherwise it is not.

It is straightforward to infer that all steps except step 2 of Algorithm 3.1 are of linear complexity in the size of the M^{IUT}.

3.4 Correctness of testers

3.4.1 Functional Correctness of Tests

In this work we limit ourselves with considering testers that use structural test coverage criteria, e.g. test generator of [59] for online testing is aimed to cover IUT model structural elements such as edges and locations of Uppaal timed automata. The structural coverage can be expressed by means of boolean "trap" variables as suggested in [62]. The traps are assignment expressions of boolean trap variables and the valuation of traps indicates the progress status of the test run. For instance, one can observe what percentage of edges labeled with traps is already passed in the course of test run. Thus, the relevant correctness criterion for the tester in this context is its ability to cover traps.

Definition 3.4. Coverage correctness of the test

We say that the RPT tester is coverage correct if the test run covers all the transitions that are labelled with traps in the IUT model.

Definition 3.5. Optimality of the test

We say that the test is length (time) optimal if there is no shorter (resp. faster) test runs among those being coverage correct.

In the following we provide an *ad hoc* procedure of verifying the coverage correctness and optimality in terms of Uppaal model checking queries and model building constraints.

Direct way of verifying the coverage correctness of the tester is to run the model checking query:

$$A\Diamond\forall\left(i : int\left[1,n\right]\right) t\left[i\right]) \tag{3.1}$$

where $t\left[i\right]$ denotes i-th element of the array t of traps. The model is assumed to be the synchronous parallel composition of IUT and Tester automata. For instance, the tester automaton generated using RPT generator[59] for IUT modelled in Figure 3.2 is depicted together with IUT model in Figure 3.5.

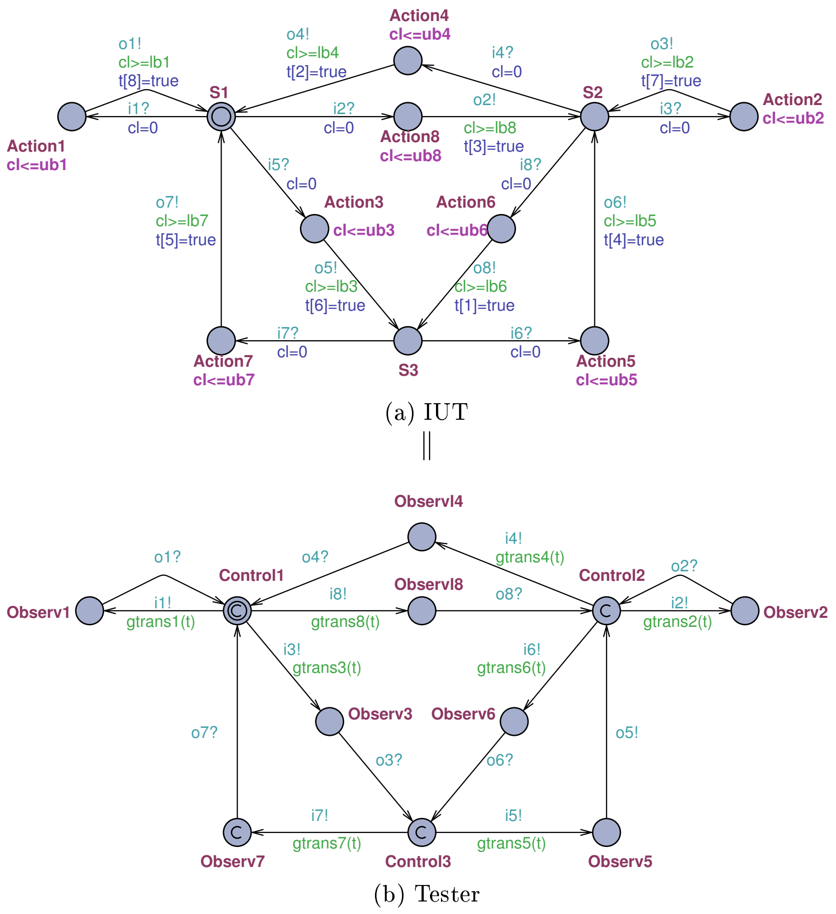


Figure 3.5: Synchronous parallel composition of IUT and tester automata.

3.4.2 Invariance of Tests with Respect to Changing Time Constraints of IUT

In previous section the coverage correctness of tests was discussed without explicit reference to time constraints of the IUT model. The length-optimality of test sequences can be proven in Uppaal when for each action in well-formed models both the duration lower and upper bounds lb_i and ub_i are set to 1, i.e., $lb_i = ub_i = 1$ for all $i \in 1, \ldots, |Action|$. Then the length of the test sequence and its duration in time are numerically equal. For instance, having some integer valued (time horizon) parameter TH as an upper bound to the test sequence length the following model checking query proves (or falsifies) the coverage of n traps with a test sequence of length at most TH stimuli and responses:

$$A\Diamond\forall\,(i : int\,[1,n])\,t\,[i]) \wedge TimePass \leq TH \tag{3.2}$$

where $TimePass$ is the Uppaal clock that represents global time progress in the model.

Generalizing this approach for IUT models with arbitrary time constraints we can assume that all edges of IUT model M^{IUT} are attributed with time constraints as described in Section 2.2. Since not all of M^{IUT} edges need to be labeled with traps (if their coverage is not requires by test goal) we apply compaction procedure to M^{IUT} to abstract away from the excess of information (for IOCO testing) and derive precise estimates of test duration lower and upper bounds. With the compaction procedure we aggregate a sequences of trapless edges and merge the aggregate with only one trap-labelled edge the trapless ones are neighbours to. As the result, the aggregate action becomes an atomic Action (see Figure 3.1) that copies the trap of the trap labelled edge included in the aggregate. The first edge of the aggregate contributes its input event and the last edge to its output event. The other i/o events of the aggregate will be hidden because all internal edges and locations are substituted with one aggregate location that represent the composite *Action*. Further, we compute the lower and upper bounds for the composite action. The lower bound is the sum of lower bounds of the shortest path in the aggregate and the upper bound is the sum of upper bounds of the longest path of the aggregate plus the longest upper bound (the later is needed not to yield premature test termination condition). After compaction of deterministic and timed IUT model it can be proved that the durationTH of a coverage correct tests have length that satisfies bound condition:

$$\sum_i lb_i \leq TH \leq \sum_i ub_i + \max_i (ub_i) \tag{3.3}$$

where index i ranges from 1 to n (n - number of traps in M^{IUT}). In case of non-deterministic IUT models, for showing the length- and time-optimality of generated tests the bounded fairness assumption of M^{IUT} must hold. A model M is $k-fair$ iff the difference in the number of executions of alternative transitions of non-deterministic choices (sharing same departure location) never exceeds the bound k. The bounded fairness property excludes unbounded "starvation" and "conspiracy" behaviour in non-deterministic models. During the test run the test execution environment (in thesis it is DTRON[63]) must be capable of monitoring the k-fairness and reporting about its violations. The violation of k-fairness condition induces automatic test verdict “inconclusive”. The safe upper bound estimate of the test length in case of non-deterministic models can be calculated for the worst case by multiplying the deterministic upper bound by factor k. The lower bound still remains $\sum_i lb_i$ (this corresponds to the angelic computation of the test case).

Proposition 3.6. Invariance of Tests with respect to changes of M^{IUT} timing [64].

Assume a trap labeled well-formed model $wf\left(M^{IUT}\right)$ is compactified, the tester automaton M^{RPT} being coverage correct is invariant with respect to the variations of time constraints specified in M^{IUT}.

The proof consists of two steps, showing that

(i) the control flow of the tester M^{RPT} does not depend on the timing of M^{IUT} and

(ii) the M^{RPT} behaviour does not influence the timing of controllable transitions of the M^{IUT}.

The practical implication of Proposition 3.6 is that a tester, once generated, can be used also for syntactic modification of M^{IUT} provided only timing parameters and initial values of traps have been changed. Note that the invariance does not extend to structural modifications of M^{IUT}.

3.5 Correctness of test deployment

Practical execution of generated tests presumes test adapters that map symbolic i/o alphabet used in the test model $M^{IUT} \parallel M^{RPT}$ (parallel composition of the IUT model and tester model) to executable inputs. Similarly, real outputs from IUT need to be transformed back to symbolic outputs. This mapping performed by test adapters may introduce additional delays that are not reflected neither in IUT nor tester models. Also, distributed test configurations may contribute extra delays and propagation time to test execution, that can alter ordering of test stimuli and responses specified in the model. By applying network monitors one can measure the latency of form $\triangle = \left[\delta^l, \delta^u\right]$ at each test input and output adapter. To verify the feasibility of the executable test suite, the latency estimates need to be incorporated also in the tester model and their impact re-verified against the correctness conditions defined in the earlier development steps.

The key property to be verified when deploying MBT test in distributed execution environment is Δ-testability introduced in [34]. Parameter $\triangle$ shows the delay between consecutive test stimuli necessary to maintain the ordering of input-output events at ports. Thus, when verifying the correctness of distributed deployment of test one needs to proceed as following:

***Step* 1:** estimate the latency at each input and output adapter/port. For any input symbol $a \in \Sigma^{in}\left(M^{IUT}\right)$ and any output symbol $b \in \Sigma^{out}\left(M^{IUT}\right)$ get the interval estimates of its total latency (including delay caused by adapters and propagation delays): $\Delta_a = \left[\delta_a^l, \delta_a^u\right]$ and $\Delta_b = \left[\delta_b^l, \delta_b^u\right]$ respectively.

***Step* 2:** modify the timed guards *Grd* and invariants *Inv* of each action of $wf\left(M^{IUT}\right)$ to produce the delta-extended well-formed model

$wf\left(M^{IUT+\triangle}\right)$ as follows:
$Inv \cong cl \leq ub \longmapsto Inv' \cong cl \leq ub + \delta_a^u + \delta_b^u$
$Grd \cong cl \geq lb \longmapsto Grd' \cong cl \geq lb + \delta_a^l + \delta_b^l$

***Step* 3:** Rerun the verification tasks of earlier verification steps with $\Delta-$ *extended* model $wf\left(M^{IUT+\triangle}\right)$.

The procedure of constructing the test adapters for testing framework DTRON are described in detail in Chapter 5.

3.6 Summary

This chapter presented the provably correct test development workflow and described how the correctness of MBT test development increments are verified. It presented the verification conditions by major development steps (modeling a system under test, specifying the testing goal Synthesizing a test Suite, Creating the test adapters and deploying them) and present a constructive way of discharging the verification obligations as model checking and syntactic analysis tasks.

4 Model learning

4.1 Chapter overview

The goal of Chapter 4 is to develop an algorithm of incremental learning of UPTA automata and demonstrate its applicability for model-based control of human assisting robots and for conformance testing of distributed applications. Two versions of timed automata model learning algorithm are presented, one relevant for learning from recordings of human motions during surgical procedures, the other adjusted for learning from network traffic monitoring logs. The application of algorithms is demonstrated on a surgical scenario and on network traffic monitoring log analysis examples.

The author participated in the development of the learning algorithm and laboratory setup for human-robot interaction learning. He designed and implemented the integration software for automated near-infrared camera based 3D measurement system (3DMS) to record and playback motion capture inputs for the learning algorithm. This resulted in a unified platform for the TDU capture system, Tallinn University of Technology 3DMS and offline use without multiple expensive laboratory setups.

The results introduced in this chapter are based on the publications:

(i) Vain, J., Miyawaki, F., Nõmm, S., Totskaya, T., & Anier, A. (2009, August). Human-robot interaction learning using timed automata. In ICCAS-SICE, 2009 (pp. 2037-2042). IEEE.

(ii) and partially in: Vain, J; Miyawaki, F.; Nõmm, S.; Totskaya, T.; Anier, A. (2009). Human-robot interaction learning using timed automata. ICCAS-SICE 2009 : ICROS-SICE International Joint Conference 2009, Fukuoka City, Japan, August 18-21, 2009, Proceedings. Tokyo: IEEE/SICE, 2037−2042.

4.2 Background

The review by Neto et. al. [65] stated that the most frequent reasons why model-based methods have not been easily accepted in software industry are:

· considerable modelling effort,

· MB approaches have poor integration with software development processes,

· they lack empirical evaluation from industrial environments.

In this section we address the problem how models for robot control and for MBT can be constructed by means of machine learning methods, specifically, by applying automata learning technique. We highlight two contexts of model learning: (*i*) learning the Human-Robot Interaction (HRI) for robot action control and (*ii*) learning interaction between the IUT and its environment for load generation in MB load testing. Two versions of the learning algorithm will be presented for the subclasses of Uppaal timed automata. The approaches will be demonstrated on application scenarios, one based on a fragment of surgeon's and nurse's collaborative motions during surgery, and the other, based on IEEE1394 distributed leader election procedure with 4 networked nodes. Since all nodes of IEEE1394 leader election procedure follow the same protocol scaling up the model to n nodes for running different load patterns is straightforward.

4.2.1 Human-Robot interaction learning

In this work, the Human-Robot Interaction learning problem is studied in the context of cooperative surgical task accomplishment by Scrub Nurse Robot (SNR) [8] and a human surgeon. The main challenge in SNR control and its adaptation to human surgeon is learning the proper reactions of human scrub nurse who is assisting in surgical procedures. Therefore, imitational learning has been regarded as more relevant approach in given context than generating a s et of "synthetic" (e.g. length optimized) robot manipulator trajectories. Although possibly more efficient (in terms of time, energy consumption etc.), the synthetic behavior of the SNR may feel unnatural and distract the surgeon's attention during critical phases of surgery.

In our approach SNR is supposed to learn the basic movements by observing initially the surgical procedure passively and later, when involved in cooperative action training, it improves its interaction model incrementally. The SNR learning architecture depicted in Figure 4.1 is layered into two levels as described in [66]: low-level gesture learning/recognition (not shown in the figure) and high-level behavior learning (modules 1 and 3 in the figure).

The layered leaning architecture provides flexible infrastructure to combine advantages of short-term gesture learning techniques [67, 66, 68] with a long-term behavior learning and recording in the form of Uppaal timed automata. In the supervised-unsupervised learning scale the SNR architecture implements the hybrid learning method. At first, low-level supervised learning is applied off-line for recognition of the reference set of primary gesture patterns, creating the alphabet of these patterns and studying inter-motion transition events that are used later for defining state transitions at high-level behaviour learning.

The high-level behavior learning is unsupervised. It is applied in off-line mode (box 1 in the figure) when constructing the initial model of the participants' interaction in the surgical procedure. Later, when the first model is constructed off-line, it is updated on-line (module 3) whenever the monitored interactions do not conform with this model (for detection of new cases the counter examples issued by Uppaal TRON in the course of RT-IOCO monitoring are used).

For high-level learning algorithm the input is the sequence of time stamped motion switching events issued by low-level motion recognition subsystem where each event is specified with *motion id*, *state information* at which the switching occurred and *switching detection time stamp*. Before reloading the updated model into robot action planning system (module 5), the model is checked against the correctness criteria such as non-Zenoness, safety, liveness, etc. (module 4). If the model checking fails, the updates together with diagnostic traces will be passed to the human analyst who decides the relevance of applied correctness criteria and/or new behaviors to be learned during operation (module 6). The outcomes of the analysis may influence the model correctness criteria in module 4 and possibly trigger completely new learning cycle in module 1.

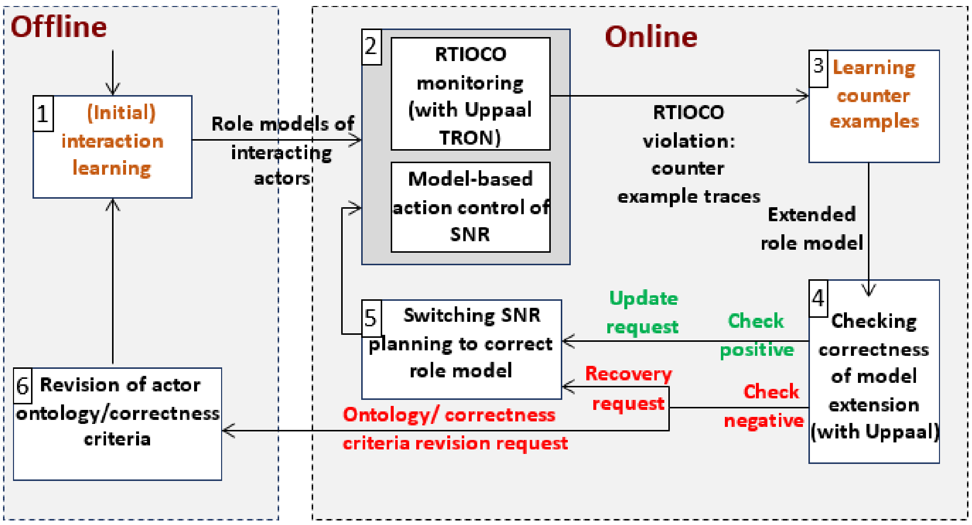

Offline
Online
1 (Initial) interaction learning
Role models of interacting actors
2 RTIOCO monitoring (with Uppaal TRON)
Model-based action control of SNR
RTIOCO violation: counter example traces
3 Learning counter examples
Extended role model
4 Checking correctness of model extension (with Uppaal)
Check positive
Update request
Check negative
Recovery request
5 Switching SNR planning to correct role model
Ontology/ correctness criteria revision request
6 Revision of actor ontology/correctness criteria

Figure 4.1: SNR high-level learning architecture.

4.2.2 Learning from network traffic monitoring logs

Performance testing presumes generating the test cases that represent different load patterns. One source of data for extracting workload profiles is the traffic log which shows how real users expectedly would interact with the system to be tested. [69] states that in performance testing, it is important that the traffic generated from workload models mimic the load generated by real users as closely as possible. Otherwise it is not possible to draw any reliable conclusions from the test results. The idea of using probabilistic timed automata (PTA) for encoding load patterns learned from logs has been proposed already in [70].

The probability estimates collected by log analysis give good metrics for developing the test cases for most typical load situations. On the other hand, when generating the tests that prefer behaviours of higher probability, the bugs occurring in behaviors of low probability may remain undetected and can cause considerable damage in rare but critical situations. Therefore, in CPS context we focus on learning the non-deterministic TA instead of PTA assuming implicitly that all non-determinstic choices encoded in the model are *strongly fair* [71]. Thus, fairness is another hypothesis concerning non-deterministic IUT models. Due to the different learning context the assumptions and the type of Uppaal automata used for learning load patterns differ from the ones used in HRI learning. In HRI learning the synchrony of cooperating actors' motions observed once cannot be extrapolated to further motions. That is because in case of unsupervised learning the learner does not have knowledge if the motions observed simultaneous once are causally related or they just happened to be simultaneous accidentally. Therefore, the traces used in HRI learning are not *forward stable* [72] regarding synchrony assumption. For instance, surgeon's hand stretching for muscle relaxing does not mean waiting for some instrument. On the other hand, the data communication between IUT and its environment once observed synchronous remains synchronous in all further communications as well. Synchronous communication needs to be represented explicitly by channels because their interaction always presumes the involvement of both parties. Hence, regarding synchrony, the traces are forward stable and the use of channels in Uppaal TA constructed from such traces is justified. The traces used for learning are abstract representation of network monitoring logs that record i/o event history at the network ports that later become test interfaces with SUT. The technical details of event monitoring feature developed within this work are highlighted in Chapter 5.

4.3 Timed automata learning: related work

The construction of models from observations of system i/o behavior is regarded as an automata learning problem [73]. For finite-state reactive systems, the active learning means constructing a (usually deterministic) finite automaton from the answers to a finite set of membership queries, each of which asks whether a certain sequence of input symbols (observed events) is accepted by the automaton or not. There are several techniques (see, e.g., [74, 75] for overview) which use the same basic principles; they differ in how membership queries may be chosen and in how an automaton is constructed from the answers. The techniques guarantee that a correct automaton will be constructed if sufficient information is obtained. In order to check the sufficiency of learning sets, the equivalence queries are used [74] that ask whether a hypothesized automaton accepts the correct sequences of symbols. Such a query is answered either by "yes" or by a counterexample on which the hypothesis and the correct language disagree. In [32] one of those learning algorithms, namely Angluin's [74], is extended to the setting of timed systems and named to event recording timed automata learning. This automata class is restricted to be event-deterministic in the sense that each state has at most one outgoing transition per action (i.e., such an automaton obtained by removing the clock constraints is deterministic). Under this restriction, timing constraints for the occurrence of an action depend only on the past sequence of actions, and not on their relative timing. As an alternative to the active learning method of [32] we present a passive learning algorithm that has following features (the features choice is motivated by the specifics of the robot control and load testing applications):

(i) The algorithm implements *online learning* strategy, i.e. the learner does not have a possibility to back-track and ask equivalence or membership queries about the arbitrary length prefixes of the learning input trace. As a result, the learning algorithm constructs relative to input trace complete non-deterministic timed i/o automaton[31], such that all observation sequences learned are also reproducible by that automaton.

(ii) Since our aim is learning interactions between multiple automata instead of learning a single automaton the algorithm constructs the *composition of interacting* Uppaal automata. In this work, two communication and synchronization cases are considered:

- *Case* 1: the processes are assumed to communicate over i/o variables and synchronize by means of *clock constraints* only. That is because the forward stability of synchronization hypothesis cannot be guaranteed in the *incremental* and unsupervised learning of human interactions. To ensure the incrementally of learning [76] we want to guarantee that the model built based on past observations will be updated only when new observation data are processed and there is no back-tracking during online model construction. Otherwise, if a new observation would violate the synchrony hypothesis (made based on past observations) then potentially extensive backtracking is required in the online learning process to replace the channels with relevant timing constraints in the component automata.
- *Case* 2: the processes are assumed to communicate over i/o variables and synchronize by means of *channels*. Forward stable synchronization assumption is due to the fact that observable communication actions always incorporate both communication parties and the learner can use channels according to Uppaal TA semantics for representing synchronous input-output action pairs.

(iii) The algorithm uses predicate abstraction for clustering the events and for encoding their occurrence conditions on clock and state variables. Non-deterministic constraints in the transition guards and location invariants are constructed in the form of linear interval constraints. Incremental adjustment (without backtracking) of the interval bounds during leaning process helps keeping the balance between the learning algorithm complexity and the precision of the model constructed by learning.

4.4 Contribution: Unsupervised learning of Uppaal timed automata

4.4.1 Learning with asynchronous communication assumption

The first version of the learning algorithm that addresses the synchronization assumptions of *Case* 1 takes the sequence of human motion switching events and constructs the Uppaal model where processes communicate over shared variables and synchronize using clock constraints. We assume that

(i) the motions of cooperating humans are observable in 3 dimensional space coordinates and augmented possibly by using additional sensor data, e.g. 2nd order dynamics from motion capture system;

(ii) the observer has holistic view of the system events (snapshots of system state and i/o variables) tagged with time stamps of a global clock;

(iii) each participant in the interaction the i/o of which is observable is modelled as separate UPTA process.

(iv) a set of possible motions (motions alphabet) occurring in an observation log is known from low level learning methods and it maps to the set L of UPTA locations.

The learning algorithm records the order and timing of motion switchings. For convenience of defining the learning algorithm we partition the transition relation D by locations of TAIO into n subsets (n is the number of locations in the model), s.t. $D = \bigcap\limits_{i \in [1,n]} D_i$ and D_i corresponds to the location $l_i \in L$ all the edges of D_i are departing from. By definition$\forall i, j \leq n, i \neq j \wedge D_i \cap D_j = \varnothing$. Note also that D_i-s are multisets, because there may be more than one edge between any pair of locations l_i and l_k, but we require that all the edges (l_i, l_j) are distinguishable. To distinguish the instances of (l_i, l_j) edges we introduce an index k and refer to an edge using notation $t\,(l_i, l_j, k)$.

The learning algorithm constructs a symbolic model in the sense that explicit variable values observed at motion capture are abstracted in the model using interval constraints. To define the *equivalence relation* between the event observations we introduce a robustness parameterR_i for each i-th state variable and similarly a robustness parameter R_t for observation time instances. Parameters R_i define the maximum distance between any two points in the equivalence class on i-th variable domain, i.e. the granularity of the TAIO factor space.

Before introducing the main steps of the algorithm we define the input, output and parameters of the algorithm:

Input: The sequence E of parametrised observations represent the ordering of switching events of one or more interacting parties (hereafter called actors). An example fragment of E is depicted in Table 1. Each element $e_i \in E$ (represented as a row in the Table 1) is described as a triple $e_i = \langle id_i, ts_i, X_i \rangle$, where id_i identifies the motion of an actor beginning with i-th switching event, ts_i is the timestamp of i-th switching event and $\bar{X}_i$ is the valuation of observable state variable vector X_i at time instant ts_i. Note that only those variables being relevant to the actors' interaction model are presented in X_i. The relevance of X_i for model state is defined by actors' observable i/o configuration model and by feature extraction algorithm to minimize it. For instance, in Fig. 2 both actors have two inputs and two outputs. Inputs model the observables the actor's behaviour depends on. The outputs model the observable effect of actor's concrete actions.E is implemented as FIFO buffer where motion detection system writes new events into buffer using ***put***-operation in the order of their occurrence, and the learning algorithm reads events using ***get***-operation that returns the oldest unread element of E. The emptiness of E is checked without shifting the read pointer of E.

Output: The model of observed behaviour defined as Uppaal TA. An example of the automaton learned from observations of Table 1 is represented in Figure 4.1.

Parameters: To reduce the model state space and to select only the state variables being important from the SNR control point of view we use the selector function P_r that filters out the observable state variables of importance (feature vector), i.e., P_r defines the subset $X_P \subseteq X$ the valuations $\bar{X}_i$ are mapped on. For instance, if the valuation of variables X_c is important for model behaviour we define $P_r(X) = \{\ldots, X_c, \ldots\}$. The observation robustness R allows defining equivalence classes used in atomic propositions in TAIO edge guards. Parameter RS denotes the rescaling vector that consists of scaling functions, one for each state variable. Rescaling is necessary for keeping the model in a compact non-negative integer domain (the restriction comes from limited set of data structures UPTA is currently supporting).

Algorithm: The algorithm comprises following basic steps:

***Step* 1:** Unless the buffer of event sequence E is not empty, read the motion switching event from buffer E and interpret it as an edge from last reached location to possibly new target location. The source location is supposed to be known as destination location of the previous event (read from E) or it is an initial location l_0 when the first event is taken from buffer. If the buffer E is empty go to *Step* 3.

***Step* 2:** Check if the edge and location representing the latest event e read from E is already included in any existing equivalence class of model edges. If the inclusion is established the algorithm returns to *Step* 1. If the model element is not in any existing equivalence class the new equivalence class is created and the algorithm returns to *Step* 1 thereafter.

***Step* 3** (Model post processing: model reduction): Model reduction minimizes the set of state variables necessary for specifying transition guards of the TAIO model. Reduction must not increase the non-determinism of model. It means that for each location l_i its outgoing edges' guards should preserve the determinism of choices, that is, there must exists at least one variable x_k for each pair of edges $t(l_i, l_j, .)$ and $t(l_i, l_k, .)$ $(j \neq k)$ s.t. their guards are mutually exclusive, i.e.,

$$\forall l_i \in Q, \forall t(l_i, l_j, .), t(l_i, l_k, .) \in D : \neg(g(t(l_i, l_j, .)) \wedge (g(t(l_i, l_k, .))).$$

***Step* 4:** Construction of location invariants. For each location l_i the invariant $I(l_i)$ is constructed from the guards of incoming and outgoing edges s.t.

$$I(l_i) \equiv \bigwedge_k g(t(l_k, l_i, .)) \wedge \bigvee_j g(t(l_i, l_j, .)).$$

***Step* 5:** (Reduction of causal non-determinism). *Steps* 1 to 4 may introduce non-deterministic guards if the set of observables is limited, observation robustness R is chosen too large, or the behavior to be learned is inherently non-deterministic. To reduce non-determinism in the model the history variable h is introduced that extends the model state space. The variable h uniquely encodes the sequence of k last steps of the trace prefix that lead to the location of non-deterministic branching. The parameter h allows distinguishing the (bounded) prefixes of traces and refer to them in edge guards. But this extension is context sensitive and may increase the model postprocessing complexity drastically.

Notations used in the algorithm

- g, asg, inv, ch are syntactic variables denoting the model elements such as guard condition, assignment, invariant and channel respectively;
- Text in *Times Italic* denotes the value of function or expression;
- $\bar{x}$ and $\bar{cl}$ denote explicit values of variables cl and x;

- h is the stack of location names generated by the algorithm;
- Interval extension operator: $[x^-, x^+]^{\updownarrow R} = [x^- - \delta, x^+ + \delta]$, where $\delta = R - (x^+ - x^-)$, and x^-, x^+ are interval lower and upper bound respectively;
- . denotes the concatenation of terms in syntactic expressions (term is either an atom if within quotes, e.g. '`clock <= `', or the value of an expression otherwise);
- .. denotes undefined value (used when the model terms have not been fully constructed yet).
- E event buffer, en event $e \in E$ is defined as a triple:

$$\langle \mathit{target\,action}\ , \mathit{switching\,time}\ , \mathit{switching\,state} \rangle$$

The detailed algorithm in pseudo code is depicted in the following:

Algorithm 4.1 Learning Uppaal TA with asynchronous communication assumption.

```
Initialization
  $L \leftarrow l_0$          % $L$ - set of locations, $l_0$ - (auxiliary) initial location
  $T \leftarrow \varnothing$          % $T$ - set of edges
  $k, k' \leftarrow 0, 0$    % $k, k'$ - indexes for distinguishing multiple edges between
                 same location pairs
  $h \leftarrow l_0$          % $h$ - history variable valuated with the $id$ of currently
                 processed motion in $E$
  $h' \leftarrow l_0$         % $h'$ - variable valuated with the $id$ of the motion before
                 previous
  $h_{cl} \leftarrow 0$        % $h_{cl}$ - clock reset history
  $l \leftarrow l_0$          % $l$ - destination location of the current switching event
  $cl \leftarrow 0$         % $cl$– local clock variable of the automaton being learned
  $g_{cl} \leftarrow \varnothing$        % $g_{cl}$ - 3D array of clock intervals where resets are enabled
  $g_x \leftarrow \varnothing$         % $g_x$- 4D array of state intervals that constitute the state
                 switching condition
                 % $E$ event buffer, consisting of switching triples:
                 [$target\_action\_ID$, $switching\ time$, $switching\ state$]
Algorithm
  1:while $E \neq \varnothing$ do % Exit when buffer empty
  2:  $e \leftarrow$ get $(E)$ % Read event $e$ from event buffer $E$
  3:  $h', h \leftarrow h, l$
  4:  $l, cl, \mathbf{X} \leftarrow e[1], (e[2] - h_{cl}), e[3]$
```

```
5: if $l \notin L$ then % if new type of event
6:   $L \leftarrow L \cup \{l\}$ % add new location
7:   $T \leftarrow T \cup \{t(h,l,1)\}$ % add new edge
8:   $g_{cl}(h,l,1) \leftarrow [cl, cl]$ % add clock reset point
9:   for all $x_i \in X$ do % for all observable state components
10:    $g_x(h,l,1,x_i) \leftarrow [x_i, x_i]$ % add the state switching point
11:  end for
12: else % If the state switching event $e$ is in the existing equivalence class
13:  if $\exists k \in [1, |t(h,l,.)|]$,
         $\forall x_i \in X : \bar{x}_i \in \mathbf{g_x}(h,l,k,x_i) \wedge \bar{cl} \in \mathbf{g_{cl}}(h,l,k)$ then
             % $\bar{x}_i$- value of variable $x_i$, $\mathbf{g_{x(.)}}$ - interpretation set of $g_{x(.)}$
14:    goto 34
15:  else% if switching $e$ extends existing equival. class
16:    if $\exists k \in [1, |t(h,l,.)|], \forall x_i \in X : \bar{x}_i \in \mathbf{g_x}(h,l,k,x_i)^{\updownarrow Ri}_{-} \wedge \bar{cl} \in \mathbf{g_{cl}}(h,l,k)^{\updownarrow Ri}_{-}$
17:    then
18:      if $cl < g_cl(h,l,k)^-$ then $g_cl(h,l,k) \in \left[cl, g_{cl}(h,l,k)^+\right]$ end if
19:      if $cl > g_cl(h,l,k)^+$ then $g_cl(h,l,k) \in \left[g(h,l,k)^-, cl\right]$ end if
20:        for all $\forall x_i \in X$ do
21:          if $\bar{x}_i < g_x(h,l,k,x_i)^-$ then
                 $g_x(h,l,k,x_i) \longleftarrow \left[\bar{x}_i, g_x(h,l,k,x_i)^+\right]$ endif
22:          if $\bar{x}_i > g_x(h,l,k,x_i)^+$ then
                 $g_x(h,l,k,x_i) \longleftarrow \left[g_x(h,l,k,x_i)^-, \bar{x}\right]$ endif
23:        end for
24:    else % if switching $e$ does not fit into any existing equivalence class
25:      $k \longleftarrow |t(h,l,.)| + 1$ % increment the number of $l$ departing edges
26:      $T \longleftarrow T \cup \{t(h,l,k)\}$ % Add new edge
27:      $g_{cl}(h,l,k) \longleftarrow [cl, cl]$
28:      for all $\forall x_i \in X$ do
29:        $g_x(h,l,k,x_i) \longleftarrow [x_i, x_i]$ % Add new state switching point
30:      end for
31:    end if
32:  endif
33: $a(h', h, k') \longleftarrow a(h', h, k') \cup Xc$ % add assignment to previous transition
34:endwhile
35:for all $t(l_i, l_j, k) \in T$ do % compile transition guards and updates
36: $g(l_i, l_j, k) \longleftarrow cl \in g_{cl}(l_i, l_j, k) \wedge \bigwedge_{s \in [1,|X|]} x_i \in g_x(l_i, l_j, k, x_s)$
37: $a(l_i, l_j, k) \longleftarrow X_c,\ cl \longleftarrow random(g(l_i, l_j, k)), 0$
        % assign random value in $a$
```

```
38:endfor
39:for all $l_i \in L$ do
40:    $inv(l_i) \longleftarrow \bigwedge_k g(t_{ki}) \bigwedge \bigvee_j g(t_{ij})$ % compile location invariants
41:endfor
```

4.4.2 Learning with synchronous communication assumption

Algorithm 4.2 introduced in this Section is extended version of Algorithm 4.1 (introduced in Section 4.4.1). It takes the log of input-output events recorded by network monitor at ports of distributed IUT and its environment and constructs the Uppaal model where processes communicate over shared variables and synchronize using both clock constraints and channels. The main difference between Algorithm 4.1 and Algorithm 4.2 is that if the former assumes pre-existing knowledge about the set of model locations (each motion maps to a location and switching of motions maps to edges that are known from low-level learning method), in Algorithm 4.2 the set of model locations is not known in advance. The set of locations is generated in the course of learning process by identifying the equivalence classes of environment components' outputs. When the interactions between IUT and its environment are learned we distinguish the components of environment by ports of the IUT they are directly communicating with. The Uppaal automata that model Environment components are assumed to be output deterministic, i.e. there is one-to-one correspondence between the locations of environment automata and the equivalence classes of environment components' outputs. The assumptions for Algorithm 4.2 coming from this learning context are summarised in the following:

Assumptions of model construction

(i) The model used for testing represents two types of components (or nodes), the components of IUT and the components of environment Env. The interactions between the components of IUT and Env are observable as events that update the values at components' ports. Each port is allocated to only one component either to IUT's or Env's. Thus, a link between components is identified by the pair of ports it is connecting.

(ii) The communication between ports is unidirectional. Given a connection between ports P_i^E and P_j^S belonging to environment component P^E and IUT component P^S respectively, is modelled in the UPTA as a channel $ch_{ij} = \left\langle P_i^E, P_j^S \right\rangle$ from the i-th output port of P^E to the j-th input port of P^S.

(iii) The learning objective is to construct a model that represents how Environment chooses inputs of IUT after IUT has responded to the Environment's earlier stimuli. Since the model learning algorithm is targeted to load test generation, the specific control structure of IUT model can be ignored. We simply assume the input enabledness of IUT w.r.t. all of its ports. Technically, it suffices introducing an automaton for each IUT component where the automaton has canonical control structure like proposed in [61]. The IUT component automaton responds either by writing data to its output ports or by executing unobservable internal actions that results in a special *timeout* event issued by network monitor.

(iv) From Environment perspective the set E of events observable at ports consists of two subsets $E = E^{IUT} \cup E^{Env}$, where E^{IUT} are the events produced by IUT (*events observable* to Env), and the events E^{Env} produced by the components of the environment (*controllable* events). It is assumed that set E^{IUT} includes also timeouts (TO), i.e. when IUT does not respond to input within given time period.

(v) The event log $Log\,(E)$ starts with the Environment event and ends with the event produced by IUT, i.e. $e_1 \in E^{Env}$ and $e_n \in E^{IUT}$, where $n = |Log(E)|$.

(vi) The observations of events e_i in the log are recorded as triples $\langle P,\, TS,\, X\rangle$, where

(a) the pair of ports ($P = \langle Port_i,\, Port_j\rangle$) identifies the channel ch_{ij} between send and receive *processes* (further, for shorthand we use directly channel label instead of the pair of ports);

(b) TS is a timestamp according to the network monitor's clock;

(c) the vector X of data variables communicated between ports is denoted by $X = X^i$ if data propagate from $Port_i \in P^E$ to $Port_j \in P^S$ and by $X = X^o$ if data propagate from $Port_i \in P^S$ to $Port_j \in P^E$;

(d) Timeout as special event is recorded in the form of triple$\langle ., TS, *\rangle$, where *for all* i, $x_i^{out} = *$ ("." and "*" are wildcard symbol for channel and for variable x_i^{out} value). To treat the symbol * uniformly with numeric values we extend the semantics of standard functions min and max as follows: $min\,(*, x) = min\,(x, *) = max\,(*, x) = max\,(x, *) = x$.

Elements of the model constructed

(i) By channel direction we distinguish two types of edges of Environment automata to be learned: the edges are *controllable* if they model controllable events and *observable* if they model observable events; no other types of edges exist.

(ii) The locations with observable incoming edges and controllable outgoing edges are called *active* and the locations are *passive* if they have controllable incoming edges and observable outgoing edges. There are no other sorts of locations.

(iii) Like in Algorithm 4.1 the non-deterministic assignments and guard conditions are specified with closed intervals $[lb, ub]$ with lb being lower and ub upper bound respectively.

(iv) In the environment automata the outgoing edge updates are identified by values of X^i sent by the environment components, and by values of X^o of the observable event, the guard condition of observable edge incoming to current active location are identified.

Notation

- `g,asg,inv,ch` are syntactic variables denoting the model elements such as guard condition, assignment, invariant and channel respectively;
- Text in *Times Italic* denotes the value of function or expression;
- $\bar{x}$ and $\bar{cl}$ denote valuation of variables cl and x;
- h is the stack of generated location names;
- . denotes the concatenation of terms in syntactic expressions (term is either an atom if within quotes, e.g. '`clock <= `', or the value of an expression otherwise);
- .. denotes arbitrary value (used when the model terms have not been fully constructed yet).

Implementation of the Algorithm The algorithm comprises two blocks: *BLOCK* 1 interprets the controllable events and *BLOCK* 2 observable events. Both blocks have two cases. In the *Case* 1 the event to be learned is within an already existing equivalence class of active locations and controllable edges or it can extend the classes within the limits that are defined by robustness parameter R. In the *Case* 2 the event is out of bounds of existing equivalence classes and a new class is introduced with the initial FIFO value defined by the value vector $\bar{X}$ of event observable attributes $x \in X$.

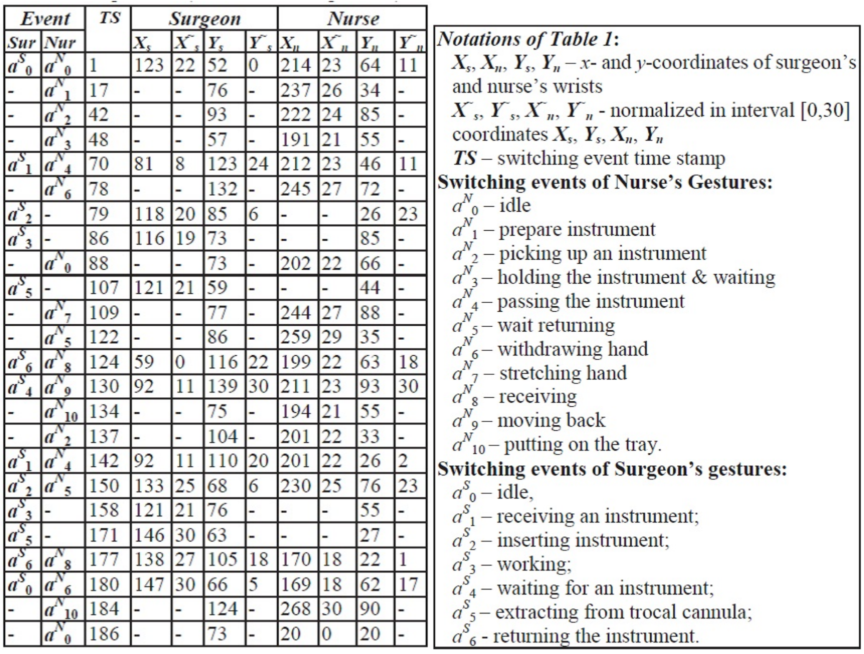
Algorithm 4.2 Learning Uppaal TA with synchronous communication assumption.

```
1:while $E \neq \emptyset$ do $e \leftarrow pop(E)$ % get the latest event recorded in the buffer $E$
2:  if $e \in E^{IUT}$ then $e^{-1} \leftarrow e$ % if the event is initiated by IUT, save it in $e^{-1}$
3:  else
4:    BEGIN BLOCK_1: % recording environment
          -controllable events $e = \langle ch^i, TS, X^i \rangle$
5:    $ch \leftarrow e[1]$ , $cl \leftarrow (e[2] - h_{cl})$, $x^{in} \leftarrow e[3]$, $push(h_{cl}, e[2])$
6:    if $\exists k, l^a, l^p : t(l^a, l^p, k) \in T(M^{env})$ for some $M^{env}$ such that
7:        $\bar{ch} = chan(t(l^a, l^p, k))$,$\forall x_i^{in} \in X^{in} : \bar{x}_i^{in} \in asg(t(l^a, l^p, k))^{\updownarrow R_i}$
          $\wedge \bar{cl} \in \left[g_{cl}^{lb}(t(l^a, l^p, k)), inv_{cl}^{ub}(l^a)\right]^{\updownarrow R_{cl}}$
8:    then % CASE 1: the parameters of event $e$ extend an
          existing equivalence class of $e^i$ events
9:      $g_{cl}(t(l^a, l^p, k)) \leftarrow$'cl>='$.min\left(\bar{cl}, g_{cl}^{lb}(t(l^a, l^p, k)), inv_{cl}^{ub}(l^a)\right)^{\updownarrow R_{cl}}$
10:     $inv_{cl}^{ub}(l^a) \leftarrow$'cl>='$.max\left(cl, inv_{cl}^{ub}(l^a)\right)$
11:     for all $x_i^{in} \in X^{in}$ do % extend the bounds of non-deterministic assignment
12:       $asg\left(t(l^a, l^p, k), x_i^{in}\right) \leftarrow$
          $'x_i^{in} :' . \left[min\left(\bar{x}_i^{in}, asg^{lb}(t(l^a, l^p, k))\right), max\left(\bar{x}_i^{in}, asg^{ub}(t(l^a, l^p, k))\right)\right]$
13:     end for
14:   else % CASE 2: if the attributes of event $e$ are
          not within existing equivalence classes of events
15:     $o \leftarrow |L^a(M^{env})| + 1$, % compute indexes for new locations
          $r \leftarrow |L^p(M^{env})| + 1$
16:     $L^a \leftarrow L^a \cup l_o^a$ where %create a new active location $l_o^a$
17:         $inv_{cl}(l_o^a) \leftarrow$'cl>='$.\bar{cl}$, % add location invariant
18:         $push(h, l_o^a)$ % add new active location to the stack
19:     $L^p \leftarrow L^p \cup l_r^p$ where $r = |L^p| + 1$, % create new passive location $l_r^p$
20:         $push(h, l_r^p)$, % add new passive location to the stack
21:         $k \leftarrow |t(l_k^a, l_r^p, .)| + 1$ % compute the index for new edge $t(l_k^a, l_r^p, .)$
22:     $T \leftarrow T \cup \{t(l_o^a, l_r^p, k)\}$, % add new controllable edge
23:         $g_{cl}(t(l_o^a, l_r^p, k)) \leftarrow$'cl>='$.\bar{cl}$, % add clock guard
24:         $ch(t(l_o^a, l_r^p, k)) \leftarrow \bar{ch}$, % add channel with suffix '?'
25:         $asg(t(l_o^a, l_r^p, k)) \leftarrow$'cl:=0' % add clock reset
26:     for all $x_i^{in} \in X^{in}$ do % create new state equivalence class for
          non-deterministic assignments
27:       $asg(t(l_o^a, l_r^p, k)) \leftarrow asg(t(l_o^a, l_r^p, k)) \cup' x_i^{in} :' . \left[\bar{x}_i^{in}, \bar{x}_i^{in}\right]$
28:     end for
```

```
29:   **end if**
30:   **END BLOCK_1**
31:   **BEGIN BLOCK_2**:
          % recording environment-observable events $e = \langle ch^o, TS, X^o \rangle$
32:    $e \leftarrow pop\,(pop\,(E))$ % get an observable event $e$
          preceding the latest controllable event in the buffer $E$
33:    $h^{-1} \leftarrow pop\,(h)\,, h^{-2} \leftarrow pop\,(pop\,(h))$,
34:          $\bar{ch} \leftarrow e^{-1}\,[1]$, $\bar{cl} \leftarrow \left(e^{-1}\,[2] - h_{cl}^{-1}\right)$, $\bar{x}^{out} \leftarrow e^{-1}\,[3]$
35:    **if** $\exists h^{-2}, h^{-1}, k : t\left(h^{-2}, h^{-1}, k\right) \in T\left(M^{Env}\right)$
          **for some** $M^{Env}$ **such that** $ch = chan\left(t\left(h^{-2}, h^{-1}, k\right)\right) \wedge$
36:          $\wedge \forall x_i^{out} \in X^{out}\left(M^{Env}\right) : \bar{x}_i^{out} \in g_x\left(t\left(h^{-2}, h^{-1}, k\right)\right)^{\updownarrow R_i} \wedge$
              $\bar{cl} \in \left[g_{cl}\left(t\left(h^{-2}, h^{-1}, k\right)\right), Inv^{ub}\left(h^{-2}\right)\right]^{\updownarrow R_{cl}}$
37:    **then** % CASE 1: the parameters of event $e$ extend
          an existing equivalence class of $e^o$ events
38:      $g_{cl}\left(t\left(h^{-2}, h^{-1}, k\right)\right) \leftarrow' cl \geq'.min\left(\bar{cl}, g_{cl}^{lb}\left(t\left(h^{-2}, h^{-1}, k\right)\right)\right)$
39:          $inv_{cl}\left(h^{-2}\right) \leftarrow' cl \geq'.max\left(\bar{cl}, inv_{cl}^{ub}\left(h^{-2}\right)\right)$,
40:          $asg\left(t\left(h^{-2}, h^{-1}, k\right)\right) \leftarrow$ % add clock reset
            $asg\left(t\left(h^{-2}, h^{-1}, k\right)\right) \cup' cl := 0'$
41:      **for all** $x_i^{out} \in X^{out}$ **do** % extend the bounds of non-det. assignment
42:      $g_x\left(t(h^{-2}l, x_i^{out}\right) \leftarrow$
43:          $x_i^{out} \in [min(\bar{x}_i^{out}, g_x^{lb}(t(h^{-2}, h^{-1}, k), x_i^{out}),$
                          $max(\bar{x}_i^{out}, g_x^{ub}(t(h^{-2}, h^{-1}, k), x_i^{out})]$
44:      **end for**
45:    **else** % CASE 2: if the attributes of event $e$ are not
                  within the existing equivalence classes of $e^o$ events
46:      $inv_{cl}\left(h^{-2}\right) \leftarrow' cl \leq'.max\left(\bar{cl}, inv_{cl}^{ub}\left(h^{-2}\right)\right)$ % add inv to loc $h^{-2}$
47:      $k \leftarrow \left|t\left(h^{-2}, h^{-1}, ..\right)\right| + 1$, % compute the index for new edge
48:          $T \leftarrow T \cup \left\{t\left(h^{-2}, h^{-1}, k\right)\right\}$, % add new observable edge
49:          $g_{cl}\left(t\left(h^{-2}, h^{-1}, k\right)\right) \leftarrow$'cl<='.$\bar{cl}$, % add clock guard
50:          $ch\left(t\left(h^{-2}, h^{-1}, k\right)\right) \leftarrow \bar{ch}$, % add channel with suffix '?'
51:          $asg\left(t\left(h^{-2}, h^{-1}, k\right)\right) \leftarrow$'cl:=0' % add clock reset
52:      **for all** $x_i^{out} \in X^{out}$ **do** % add guard on state variables
53:        $g_x\left(t\left(h^{-2}, h^{-1}, k\right), x_i^{out}\right) \leftarrow x_i^{out} \in \left[\bar{x}_i^{out}, \bar{x}_i^{out}\right]$
54:      **end for**
55:    **end if**
56:   **END BLOCK_2**
57:  **end if**
58: **end while**
```

4.4.3 Case-study 1: Learning surgeon and scrub nurse collaborative motions

The motion capture system outputs trajectories of moving objects (the hand points monitored) that are sent to motion recognition system. The motion recognition system detects motion switching events that are serialized in the event buffer E where each event has its spatial parameters and timestamp (a sample fragment of the sequence E is specified in Figure 4.1). Rescaling of a parameter x_i by operator RS_i results in its normalized value denoted by $\tilde{x}_i$. For all state variables $x_i \in X$ their value domain is normalized within interval $[0, 30]$. Robustness $R_i = 2$, for all $x_i \in X$.



Event		*TS*	*Surgeon*				*Nurse*			
Sur	*Nur*		X_s	$\tilde{X}_s$	Y_s	$\tilde{Y}_s$	X_n	$\tilde{X}_n$	Y_n	$\tilde{Y}_n$
a^S_0	a^N_0	1	123	22	52	0	214	23	64	11
-	a^N_1	17	-	-	76	-	237	26	34	-
-	a^N_2	42	-	-	93	-	222	24	85	-
-	a^N_3	48	-	-	57	-	191	21	55	-
a^S_1	a^N_4	70	81	8	123	24	212	23	46	11
-	a^N_6	78	-	-	132	-	245	27	72	-
a^S_2	-	79	118	20	85	6	-	-	26	23
a^S_3	-	86	116	19	73	-	-	-	85	-
-	a^N_0	88	-	-	73	-	202	22	66	-
a^S_5	-	107	121	21	59	-	-	-	44	-
-	a^N_7	109	-	-	77	-	244	27	88	-
-	a^N_5	122	-	-	86	-	259	29	35	-
a^S_6	a^N_8	124	59	0	116	22	199	22	63	18
a^S_4	a^N_9	130	92	11	139	30	211	23	93	30
-	a^N_{10}	134	-	-	75	-	194	21	55	-
-	a^N_2	137	-	-	104	-	201	22	33	-
a^S_1	a^N_4	142	92	11	110	20	201	22	26	2
a^S_2	a^N_5	150	133	25	68	6	230	25	76	23
a^S_3	-	158	121	21	76	-	-	-	55	-
a^S_5	-	171	146	30	63	-	-	-	27	-
a^S_6	a^N_8	177	138	27	105	18	170	18	22	1
a^S_0	a^N_6	180	147	30	66	5	169	18	62	17
-	a^N_{10}	184	-	-	124	-	268	30	90	-
-	a^N_0	186	-	-	73	-	20	0	20	-

***Notations of Table 1*:**
X_s, X_n, Y_s, Y_n – x- and y-coordinates of surgeon's and nurse's wrists
$\tilde{X}_s$, $\tilde{Y}_s$, $\tilde{X}_n$, $\tilde{Y}_n$ - normalized in interval [0,30] coordinates X_s, Y_s, X_n, Y_n
TS – switching event time stamp

Switching events of Nurse's Gestures:
a^N_0 – idle
a^N_1 – prepare instrument
a^N_2 – picking up an instrument
a^N_3 – holding the instrument & waiting
a^N_4 – passing the instrument
a^N_5 – wait returning
a^N_6 – withdrawing hand
a^N_7 – stretching hand
a^N_8 – receiving
a^N_9 – moving back
a^N_{10} – putting on the tray.

Switching events of Surgeon's gestures:
a^S_0 – idle,
a^S_1 – receiving an instrument;
a^S_2 – inserting instrument;
a^S_3 – working;
a^S_4 – waiting for an instrument;
a^S_5 – extracting from trocal cannula;
a^S_6 - returning the instrument.

Table 4.1: Observation sequence E of selected hand motion parameters.

Some of observables are inputs to other collaborative actors and some of them are feedback inputs that reflect internal causality of the agent's motions, i.e. it is assumed that the actions of agents are triggered when these parameter values satisfy certain constraints. These parameters are called observable inputs/outputs. System configuration example with inputs and outputs of agents participating in surgery is depicted in Figure 4.2.

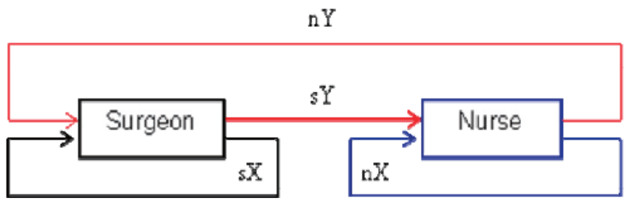
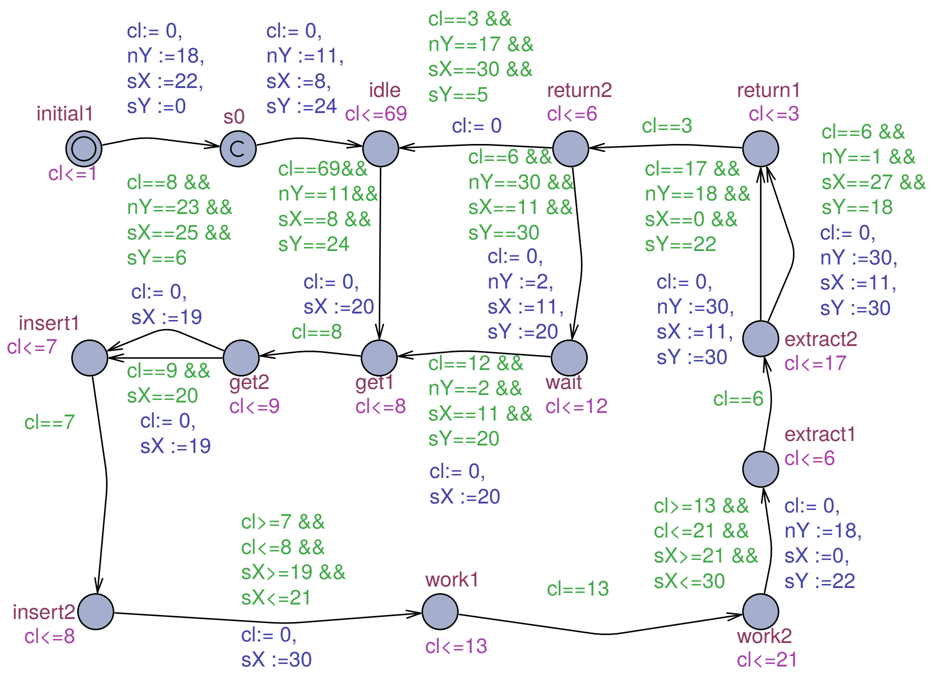


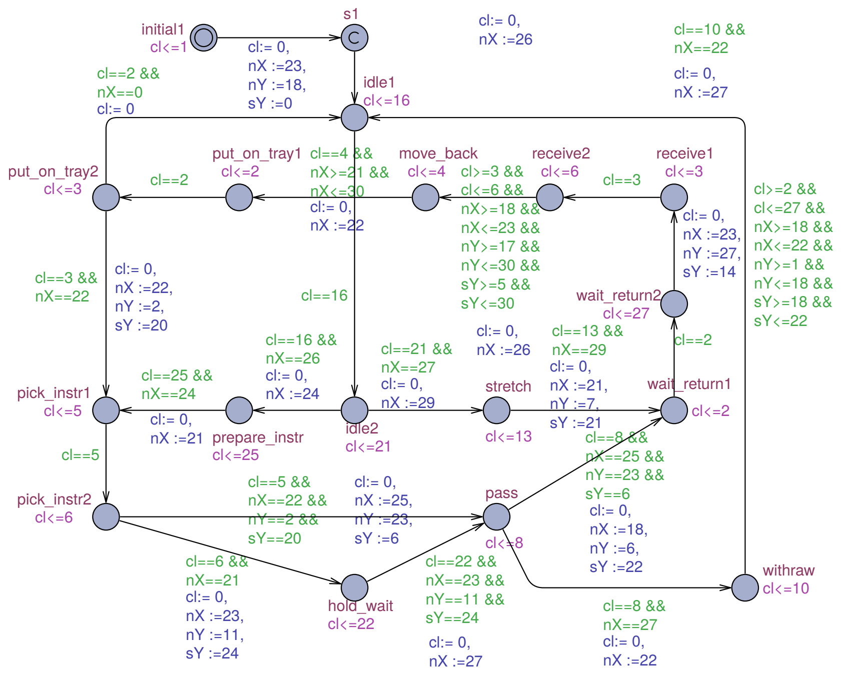
Figure 4.2: i/o configuration of Surgeon-Nurse interaction (self-loops denote self-dependencies).

The parallel composition of Surgeon's and Nurse's TAIO-s learned from interaction log recorded in event buffer E during a laparoscopic surgery (see table in Figure 4.1) is represented in Figure 4.3. The uniform value **for all** i $R_i = 2$ of robustness parameter is used for all observable state variables $x_i \in X$.


cl:= 0, nY :=18, sX :=22, sY :=0
cl:= 0, nY :=11, sX :=8, sY :=24
cl==3 && nY==17 && sX==30 && sY==5
initial1
cl<=1
s0
idle
cl<=69
cl:= 0
return2
cl<=6
cl==3
return1
cl<=3
cl==6 && nY==1 && sX==27 && sY==18
cl:= 0, nY :=30, sX :=11, sY :=30
cl==8 && nY==23 && sX==25 && sY==6
cl==69&& nY==11&& sX==8 && sY==24
cl==6 && nY==30 && sX==11 && sY==30
cl:= 0, nY :=2, sX :=11, sY :=20
cl==17 && nY==18 && sX==0 && sY==22
cl:= 0, nY :=30, sX :=11, sY :=30
cl:= 0, sX :=19
cl:= 0, sX :=20
cl==8
insert1
cl<=7
get2
cl<=9
get1
cl<=8
wait
cl<=12
extract2
cl<=17
cl==9 && sX==20
cl:= 0, sX :=19
cl==12 && nY==2 && sX==11 && sY==20
cl:= 0, sX :=20
cl==7
cl==6
extract1
cl<=6
cl>=7 && cl<=8 && sX>=19 && sX<=21
cl>=13 && cl<=21 && sX>=21 && sX<=30
cl:= 0, nY :=18, sX :=0, sY :=22
work1
cl==13
insert2
cl<=8
cl:= 0, sX :=30
cl<=13
work2
cl<=21

(a) proc_Surgeon

||



(b) proc_Nurse

Figure 4.3: Parallel composition of Surgeon's and Nurse's TAIO learned from event history of Figure 4.1

4.4.4 Case study 2: model learning for performance testing of IEEE1394 protocol

IEEE 1394 is an interface standard for a serial bus for high-speed communications and isochronous real-time data transfer. The tree identify process of IEEE-1394 is a leader election protocol initiated after a bus reset in the network. Initially all nodes in the network have equal status, and they know only to which other nodes they are directly connected to. A leader (root of the tree) needs to be elected as the manager of the bus. The protocol is designed for use in connected networks and will correctly elect a leader if the network is acyclic. Specifically, each node has two phases based on the number of children and the number of neighbours. If there is more than one neighbours, the node waits for requests from its neighbours to become their parent. If there is only one neighbour and this neighbour is not a child, then the node sends a request to the neighbour to become its parent. This implies that leaf nodes are the first to communicate with their neighbours, and that the spanning tree is built from the leaves. Furthermore, the protocol may not proceed in one run because the parent requests are not atomic and contention may arise (two nodes simultaneously send the parent requests to each other). Since only one node can be the leader, the contention must be resolved. This is achieved by timing. The IEEE1394 standard specifies that each node chooses randomly whether to wait for a long or short time. If, after the wait period is over, there is a parent request from the other node, then the node becomes the root. If there is no such request, then the node resends its own parent request and contention may result again. In Figure 4.4 an example of network consisting of four nodes is depicted where central node Node #0 is the system under test and the other nodes Node #1 to Node #3 constitute the Environment the tester needs to simulate. Thus, the learning goal is to construct the model of Environment i/o behavior by studying the logs recorded on interfaces between IUT and Environment nodes.

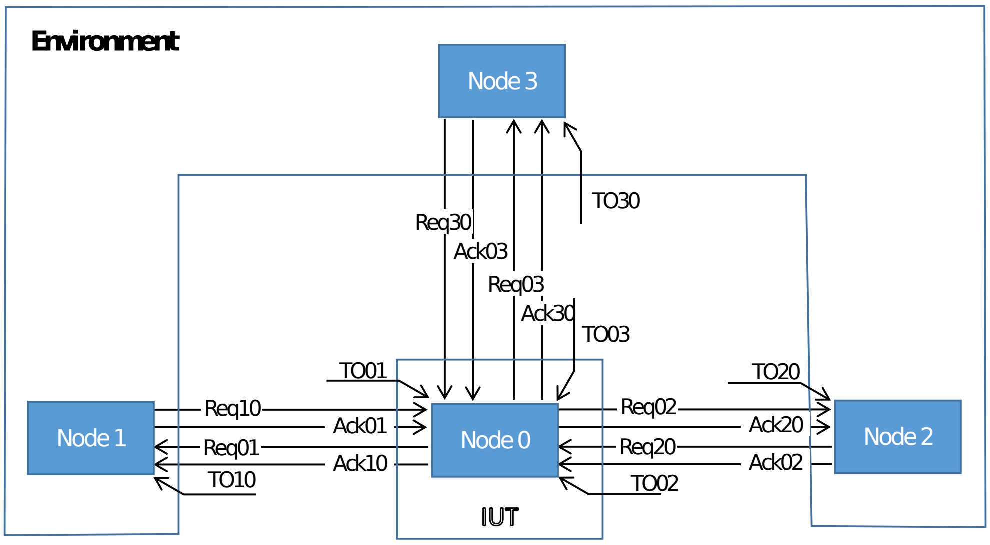


Figure 4.4: IEEE1394 test case: data flow between nodes.

In this example, Req_{ij} and Ack_{ij} denote parent request from node i to node j and its acknowledgement back from node j to node i; $\mathsf{TO}ij$ denotes delay of retrying either Req_{ij} or Req_{ji} after detecting contention between Req_{ij} and Req_{ji}.

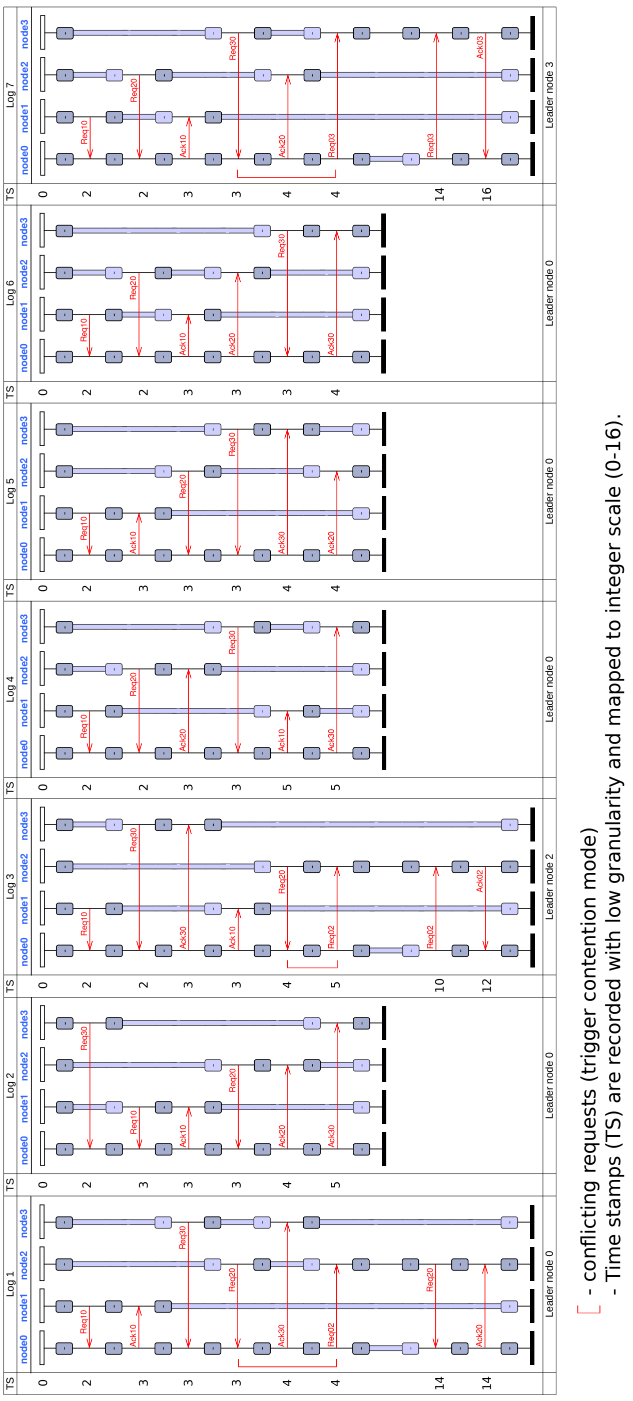
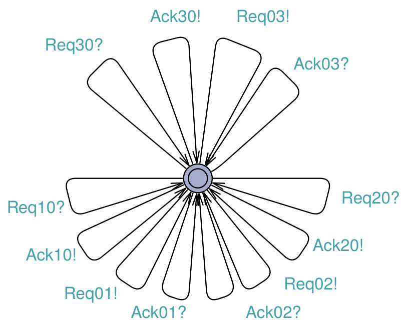
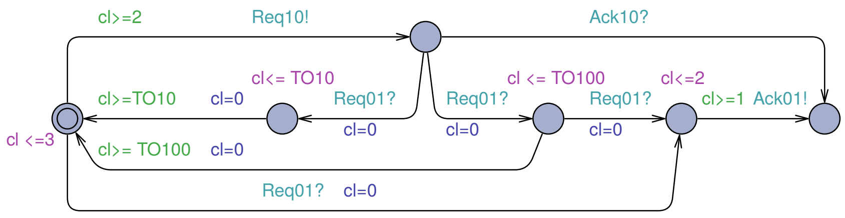


Figure 4.5: A selection of logs of monitoring network configuration of Figure 4.4

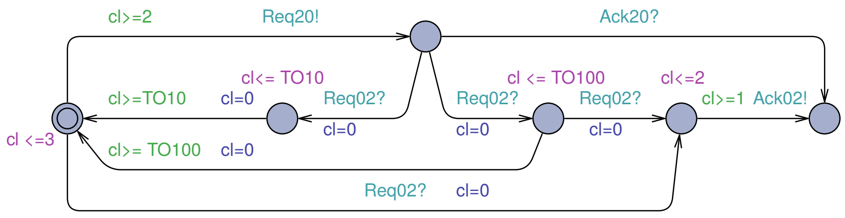
The learning algorithm introduced in Section 4.4.2 construct from recorded logs the model depicted in 4.4 that describes interactions of Node #1, Node #2 and Node #3 with Node #2. When browsing the events of each log Algorithm 4.2 simultaneously extends the automata involved in current interaction event.



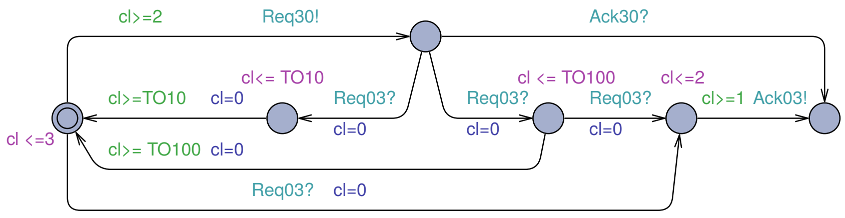
(a) node0



(b) node1



(c) node2



(d) node3

Figure 4.6: Model of IEEE1394 leader election protocol constructed by Algorithm 4.1.

4.5 Summary

We have proposed two algorithms of learning different subclasses of Uppaal timed automata: one with asynchronous parallel composition and the other with synchronous composition of extended timed i/o automata. The learning constraints and assumptions are due to the specifics of applications: learning collaborative human motions in the surgical scenario for training a scrub nurse robot high level action planning, and learning interactions in the distributed leader election protocol of standard IEEE1394 for synthesis of on-line testers for load testing. The key features of proposed unsupervised learning algorithms are following:

(i) learning is incremental, i.e., pre-existing knowledge about human scrub nurse behaviour can be re-used;

(ii) in the presence of predefined scenario models the functional correctness of learning results and the efficiency of the robot action can be verified by model checking before used for actual planning and/or test execution;

(iii) both versions of the learning algorithms can be tuned by choosing values for parameters such as feature vector, state space granularity, depth of control state history vector, scaling etc.; this allows generating families of models with different level of abstraction and of different profile tuned to specific control or testing task.

5 Model execution environment DTRON

5.1 Chapter overview

This chapter presents the DTRON model execution environment software architecture by providing the details of the subsystems, the communication model and the integration mechanism for re-use in other software. Common (automata) modelling and runtime limitations and considerations are also discussed.

The author designed the software architecture, implemented DTRON and set up the supporting website and repositories for the purpose of documentation and reuse by other parties (other bachelor/master theses and collaborating research & development projects).

The results introduced in this chapter are based on the prepared and presented publications by the author:

(i) “Anier, Aivo, and Jüri Vain. "Model based Continual Planning and Control for Assistive Robots." HEALTHINF. 2012.” and

(ii) “Anier, A., and J. Vain. "Timed Automata based provably correct robot control." Electronics Conference (BEC), 2010 12th Biennial Baltic. IEEE, 2010.”

5.2 General design context

The motivation and the need for a model based robot control and testing tool such as DTRON originally came from the *Scrub Nurse Robot* (SNR) project, a cooperative research project between *Tokyo Denki University* (TDU) and *Tallinn University of Technology* (TUT). The robot mechatronic platform design proposed by DTU is highly heterogeneous with many subsystems - controllers and algorithms for servo motors, pneumatically actuated manipulator arms and the base platform, pressure sensors for tactile feedback etc. The application scenarios for laparoscopic surgeries also require the use of auxiliary cognitive subsystem for human motion capture and recognition using video cameras, real-time 3D motion tracking, learning, and online decision making. The integration of such a versatile mixed software-harware designs presumes an integration middleware and well-coordinated use of the code base. So the question is how to preserve or gain confidence in the overall software quality including performance aspects. Aside from the software quality issues there is also the question of how to guarantee the safety of robot action. A robotic manipulator arm holding a scalpel is a high-risk threat to human in surgical scene, especially within confined spaces like a laparoscopic surgery.

The software development quality issues range from project management philosophies to deployment protocols. There are methodologies like Rational Unified Process, Agile software development, extreme programming, test-driven design and others each with its own strengths. There is no single choice to make out of the available methodologies, but rather the well-balanced composition of each.

In the traditional software development sense - this thesis focuses mainly on high level control and testing, specifically on integration testing, but as will be shown in Chapter 6- the results can easily be applied for unit, regression and integration testing scenarios and (model based) supervisory level control as well.

The thesis aims to develop a model based testing (MBT) and control framework that could be used in such a versatile project as a robotic platform development. The tool needs to be able to run on *multiple* hardware and operating systems platforms such as Windows, Linux, embedded/controllers, mobile operations systems (Android, iOS and others), but also be able to integrate with *many programming languages*. Typically, low level controller algorithms are programmed with languages like C and possibly run them on ARM architecture. But it would be equally obvious to write learning, planning and cognitive functions in high-level programming languages such Java, Python or Prolog.

The goal is to construct a model-based control and testing framework that would be able to *interconnect* possibly maximum of listed features to enable sophisticated model driven execution of different robot control stacks as well as test suites of integration testing. Although the focus is on robot control and testing applications we keep in mind that the same tool could in principle be used also for *online* (safety) *monitoring* applications. This is essential in large scale (legacy) projects where low level / unit testing is infeasible or not practical and is skipped. Then the main design validation effort falls into the high-level functionality monitoring and if necessary integration testing thereafter to ensure some global (safety) properties hold during *runtime*. For instance, consider the SNR project where the robotic platform has many manipulator arms with n-joints. It does not make much sense to focus testing an algorithm for single joint control. However, testing n-joints as whole enables to check if manipulator arms collide with itself or another arm. Such a system composed of a large number of simplistic components introduces “emergent behavioral properties” and we want to be able to address them on the level of emergent behavior.

In the following we consider the kernel component of DTRON tool - Uppaal TRON at first, and discuss the functionalities to be added to meet the goals of the distributed model-based execution framework discussed above.

5.3 Functional subsystems: Uppaal TRON

5.3.1 Background

Uppaal TRON is a model based *online* testing tool based on Uppaal engine. Tests are algorithmically generated[33], executed and checked simultaneously while maintaining connection to the system in *real-time*[77]. The Uppaal tool is used for explicit model definition and model checking with the integrated *verifier* based on a subset of timed computation tree logic (TCTL) query language. Uppaal model files are used as input to Uppaal tron.

In order to interface with actual systems in model based testing/control an *adapter* (see Figure 5.3) needs to be defined to interpret the model stimuli to SUT and transform the observations back to the symbolic form of the model. Uppaal TRON provides a C and Java *application programming interface* (API) for this. In principle, the API consists of two classes: the *Reporter* and the *Adapter* (see Figure 5.1).

We now briefly describe the usage of the API. The immediate connection to Uppaal TRON runtime and the underlying UPTA model is handled by a *Reporter*. Whenever a Reporter first connects to the runtime the settings for the following session are configured. This phase is called *"handshake"*. This happens at method $void\, configure(Reporter)$ that is invoked by the Uppaal TRON framework giving an access to the actual Reporter object for session configuration. There are few things to configure within the handshake: the *timeunit*, *timeout* and *inputs-outputs* for the *runtime* following the handshake.

Firstly, one needs to define how the *model clocks* are interpreted against the actual system clock. The Uppaal automata clocks are real-valued variables increasing monotonically. The API forces user to define how exactly this increase is to be translated to the actual system clock. The translation defines how many microseconds need to pass for every (integral) unit of model clock increase. This is a required parameter. The mapping is set by invoking the $setTimeunit(long)$ method on the *Reporter* object reference passed to $void\, configure(Reporter)$ when called.

Secondly, one needs to define the *timeout* that bounds the model execution time. This value defines the amount of logical time units previously described, that sets the upper bound to the testing session. When timeout is exceeded the session will be terminated by TRON. An Uppaal model defines abstract test sequences. Test sequence is a succession of i/o actions executed in the course of model run. Only in minimalistic and rare cases it is feasible to execute all possible runs. In many large test cases their execution may hit the timeout limit. Consider a model with a single state and a single transition that is time constrained to be enabled in a 100 years. It is hardly worth the wait. So we define a timeout we are willing to wait to cover the behaviours of interest. When timeout occurs it usually means no conformance violations were found during the execution, but since all the test sequences could not be executed it could not be said the test was a success. When this happens a message "*test inconclusive* (timed out)" is output instead, to denote this. The timeout mapping is set by invoking the *setTimeout*(*int*) method similarly to *setTimeunit*(*long*) on the *Reporter* just like previously described.

Example 5.1. If the timeunit is set to $u = 1000$ (microseconds) then an increase of 1000 units in model time would translate to $1000\mu s \times 1000 = 1s$ time elapsing during the execution of the model. Setting the corresponding timeout to $o = 2000$ would result an interpreted timeout of $u \times o = 1000 \times 2000 = 2s$

Uppaal automata are constructed using the Uppaal tool graphical user interface (GUI). At first, individual automata are constructed each with it's own initial location, variable set and clocks. The Uppaal tool then implements the parallel composition resulting in a product automaton that can be loaded into the *verifier* for model checking.

The process of automata construction is split in two: defining automata templates and their instantiating. The automata templates give the control structure and all main elements of the automaton, it leaves open only parameters that need to be instantiated when instances of templates, i.e. model processes are compiled. The parameters are instantiated in the system definition section of Uppaal API. Also synchronization constraints, *channels* between processes are instantiated in the course of compilation.

We now focus on *channels* since channels are the key constructs needed for tester and IUT communication, as well as for automata synchronization. A synchronization channel is defined on an edge between locations. When two transitions are synchronized using a channel they are bound to be taken simultaneously. Syntactically the channels are split to *initiating* and *receiving* actions denoted by suffices ! and ? respectively. A receiving synchronization (?) also disables an outgoing edge from its destination location until the corresponding initiating synchronization occurs.

Example 5.2. Consider a model extract shown in Figure 5.3. The transition labelled with receiving synchronization *move*? disables the transition until the transition labelled with *move*! is taken in the other automaton.

The *channels* serve as the basis for input-output alphabet between the model and the *system under test* (SUT). Not all the channels belong to the i/o alphabet, but only the ones explicitly declared to be so upon the "handshake" phase of the Uppaal TRON session.

So, in the testing context we need to define the *input* and *output* alphabet (implemented by channels) with regard to the Environment and SUT automata. Using the API user can register the adapter to get notified when a synchronization (denoted by a channel) event occurs in the model and possibly attach global integer variables to this event to get insight of the inner state of the model.

Since at this point we are defining an adapter that is inherently outside the automata at the SUT side - we refer to this as an “symbolic input”. An input assignment is declared calling the *int addInput*(*String*) method on the *Reporter* object reference. The string argument has the value of the corresponding channel name in the model.

Outputs are declared respectively by invoking *int addOutput*(*String*). Note that declaring a synchronization as output changes the semantics of its interpretation in the model. If normally the initiating side of the channel (!) does not disable the transition then after declaring it as “output” it does. The channel is not enabled until it has been triggered by the adapter. This is done by invoking the *report*(*int*) method on the *Reporter* object reference.

The protocol implementing the immediate connection between the model and the adapter is optimized in a way that it does not transport the synchronizations with the actual channel names. When registering inputs and outputs each channel is assigned an integer index instead. This index is returned by the corresponding *addInput* or *addOutput* method and used then to encode which synchronization exactly occurred during the runtime. Keeping track of these indexes was intended to be the responsibility of the programmer in TRON, but as will be demonstrated in following sections DTRON hides this complexity behind a more generic object-oriented API.

Example 5.3. Figure 5.3 denotes a simple composition of two automata to move a robotic joint, synchronized with channel named *move*. Synchronization forces automata running concurrently to take the transition simultaneously, i.e. blocking the transition labelled *move*? as long as necessary. Uppaal TRON API allows the Adapter to subscribe to get a notification whenever this synchronization occurs and given the example - possibly move a robotic joint. If *move* was declared an output channel instead the transition labelled with *move*! will block until it is “reported” by the adapter. Since *move*? is semantically bound to *move*! - it naturally gets blocked also.

Whenever a subscribed synchronization event occurs - the Adapter method perform(int,int[]) gets called, providing the related information about the event in the argument list. The first integral value denotes the channel index of the synchronization that occurred. The second argument of integral array gives the integral variable values “attached” to the event.

Attaching variables to a channel is done by invoking addVarToInput(int, String) and $addVarToOutput(int, String)$ on the *Reporter* object reference during handshake. The first integral parameter denotes the *channelId* index bound to have variables attached. The channel needs to be registered first in order to get the index assigned. The second String parameter defines the variable name in the model to be attached. This variable has to be strictly of the integral type.

Coming back to the $perform(int, int[])$ method - the second argument of integral array gives the values of the attached variables. Similarly, to optimize the underlying protocol the variable names are not transported, but only their values - in the order of declaration when attaching them to a channel.

Example 5.4. Declaring an input *addInput*(”*move*”) would return a result of an index 0. One can now attach a variable to the channel by calling $addVarToInput$(0,”i”). Whenever $perform(int, int[])$ is called - the $int[]$ array will be of length 1 and the first element value will hold the value of i that was held at the model during the synchronization.

Note that only variables declared in the “global” section of the Uppaal automata definition could be used to be attached to channels. If the access to local (template) variables is needed one should assign the values to a global variable just before the synchronization to carry the value.

Figures 5.1 and 5.2 show the relevant extracts of the Uppaal TRON API.

```
package com.uppaal.tron;

public interface Adapter {
  void configure(Reporter reporter);
  void perform(int chan, int[] params);
}
```

Figure 5.1: Uppaal TRON Adapter class.

```
package com.uppaal.tron;

class Reporter extends VirtualThread {
  int addInput(String channel);
  int addOutput(String channel);

  void addVarToInput(int channel, String variable);
  void addVarToOutput(int channel, String variable);

  void setTimeUnit(long microsecs);
  void setTimeout(int timeout_in_units);

  void report(int chan, int[] params)
}
```

Figure 5.2: Uppaal TRON Reporter class extract.

Example 5.5. Figure 5.3 shows a simple automaton extract denoting $m \in [0, 100]$ reflective transition on an initial location where m is enumerated uniquely across all the transitions and variable *amount* is given the corresponding value. So if a Uppaal TRON session is configured to get notified whenever "*move*" happens, $perform(int, int[])$ gets called to return the details about it.

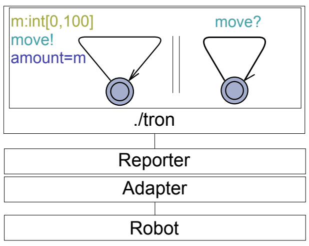


Figure 5.3: Simple robotic joint movement with Uppaal TRON.

5.3.2 Limitations of Uppaal TRON

Uppaal TRON was first hypothesized to be useful for the SNR project and turned out to be an excellent tool to do so. But as the project grew bigger the *scalability issues* began to uncover. The complexity of both the Adapter and the Model grew hand in hand. Making changes to parts of the model that modify the input or output alphabet contract implied immediate changes to the Adapter code as well. This is because the API did not support easy channel and variable mappings. It is left to the responsibility of the developer using the API. But this task can be easily underestimated that leads to cluttered code when managing the mappings. This in turn, leads to many rewrites of the Adapter code and engages one with developing the Adapter instead of making use of the advantages of the tool.

These considerations motivated the decision that the API of TRON needs to be reworked to support arbitrarily large software projects. Hiding away the low level *"book-keeping"* and exposing the API in a domain specific language (DSL) -like manner. The inner workings of the API would handle all the "book-keeping" of channels and variables relieving the programmer of this task. The API was intended to take advantage of modern *industry standard tooling* support and be multi-platform and multi-language compatible. So it would be easy and declaratively be *integrated* to software project of "industrial" size.

Second problem looking for a solution, was the applicability of Uppaal TRON in *distributed testing* scenarios. This is where system under test has multiple physical ports, possibly on different machines. This would require a *timing aware messaging medium* to coordinate the Adapters.

It was hypothesized that if there was a mechanism that enables messaging between multiple Uppaal TRON sessions it would enable *distributed execution* and allow to break down possibly large single models into many smaller and simpler ones. By achieving greater spacial modularity would lead to better *scalability* and model *maintainability* by *decoupling* large monolithic models. If there are changes made to one model there would not be an inherent need to change the adapters of model components executed by other TRON instances. Last but not least, the smaller component model would be easier to construct and read.

The main hypothesis of this improvement was to come up with a *communication architecture* to handle the messaging coordination between the adapters and the (Uppaal TRON) models. The primary design criterion for introducing an additional messaging layer was not to break the underlying formal semantics of the Uppaal timed automata. First of all, the messages would need to preserve their order in distributed execution Secondly, due to realtime constraints to model execution the *messaging transport overhead* needs to be kept to a minimum and possibly measurable to enable *compensation of sporadic communication delays.*

Having a messaging framework like that would enable the use of some *convenience features* as well. It would be possible to *monitor* the messages passing through the medium and use this information for *monitoring and debugging* purposes. Interconnected model components would also allow *centralized coordination actions* like remote model loading or re-loading from a known *resetting state.*

5.4 DTRON design considerations

5.4.1 DTRON overview.

DTRON is not intended to re-implement existing Uppaal TRON tool or come up with a new formal testing theory. The goal is rather to benefit from existing Uppaal toolset and build on top of it. The solution is intended to be *scalable* enough to allow applications in daily unit and regression testing as well as high-level (e.g. web testing) conformance testing and robot control.

To achieve this we need solid *messaging services* to coordinate the *distributed execution.* This means that the messaging framework needs to preserve the order of messages arriving to and leaving from each distributed model component. Since we have a special focus on real-time systems - the messaging *overhead* needs to be kept as *low* as possible.

The possible execution runtime environments are expected to be highly heterogeneous. Consider the Scrub Nurse Robot scenario previously covered. We expect the tool to be deployed on any operating system/platform and to be interfaced with major popular programming languages. All the possible runtime configurations could be used simultaneously during distributed execution scenarios. Low level robotic coordination modules will most likely be ran with drivers written in C, possibly on embedded architectures while high level machine learning algorithms could run Python scripts on Windows platform. This kind of inter-plaform and -language communication most obviously implies a *network socket based communications.* Socket based input-output capabilities are supported by all major programming languages so we consider this a realistic expectation.

Network socket based data interchange is quite straightforward - at first glance. Since we are dealing with multi-platform and -language scenarios we need to consider also the *data type differences.* For instance, C/++ has unsigned integers, Java does not. Java has UTF-8 strings but C/C++ essentially has (unencoded) byte-arrays instead of strings. C has *struct*-s. Java has *enum*-s and *classes* ... etc. To enable proper scaling we would need to support wide spectrum of data structures and possibly objects. This would contribute to the runtime *coherence* and minimize the possibility to accidentally “break the protocol”. We need a mechanism that eliminates the use of accidentally or intentionally *malformed messages* during runtime. That means a more sophisticated framework to meet the requirements and as much as possible avoiding implementing this from ground up.

Uppaal TRON adapter keeps the connection with the runtime to a minimum. Inputs and outputs are assigned an index and the complemented variables are positional in the corresponding array. This protocol is exposed by a minimalistic API that delegates most of the responsibility of keeping track of all of it to the developer. This approach is straightforward and sufficient in small projects having few classes, but is clearly insufficient when projects grow large. The developer is prone to lose track of assigned indexes and needs complicated supporting code to propagate the values across the code.

We explicitly identify the need for a better API that alleviates the developer from this “book-keeping” and enables a more “declarative” style programming where only control/testing-relevant code needs to be written. We also aim to have *syntactic guarantees* that malformed messages are not submitted to the data-interchange framework. This would allow *long-running* distributed tests/control routine, possibly having daemon-like agents and testers leaving and joining a coordinated execution on demand

This development emphasizes that the DTRON development process itself needs to utilize the modern industry standard *software management* and *build automation tools* (like Maven). The main goal of DTRON design is to enable *declarative style* use of distributed MBT and MBC with minimum effort on Adapter building and application sensitive model adjustment.

5.4.2 Project setup with Apache Maven

Apache Maven is a software project management and comprehension tool[78]. DTRON project is based on and makes heavy use of it in various ways.

The most common use of Maven is *dependency management.* DTRON incorporates various supplementary libraries to function. The full list of dependencies is shown in Figure 5.4. “Compile” indicates that the dependency is an integrated part of the final executable and “test” indicates this dependency is used only during unit testing, before the actual packaging of the final binary (target).

Note that aside from reusing various open-source tools the DTRON project itself is modular and is dependent of the sub-modules. We briefly describe the use of each relevant module:

com.uppaal:tron is so called *mavenized* (or repackaged) version of Uppaal TRON API to support its declarative use as set of Maven dependencies. This includes *Reporter* and *Adapter* classes implementing the original protocol to Uppaal TRON runtime. There are also few supplementary classes as part of the original API.

ee.ttu.cs.dtron:troninstaller is a module responsible for locating the Uppaal TRON executables depending on the specific platform - “tron.exe” on Windows or “./tron” on Linux. These executables are invoked internally from within Java runtime to execute local testing agents , conventionally to the original intention. This location task is modular to

```
ee.ttu.cs:dtron:jar:4.8
+- com.google.protobuf:protobuf-java:jar:2.3.0:
+- com.uppaal:tron:jar:1.5.4:compile
+- org.apache.commons:commons-exec:jar:1.1:
+- ee.ttu.cs.dtron:troninstaller:jar:2.0:
|  \- org.apache.commons:commons-vfs2:jar:2.0:
|  \- ...
+- commons-configuration:jar:1.8
|  \- commons-lang:commons-lang:jar:2.6:compile
+- commons-io:commons-io:jar:2.3:compile
+- org.slf4j:jcl-over-slf4j:jar:1.6.6:compile
|  \- org.slf4j:slf4j-api:jar:1.6.6:compile
+- commons-cli:commons-cli:jar:1.2:compile
+- org.slf4j:slf4j-simple:jar:1.6.6:compile
+- com.jayway.awaitility:awaitility:jar:1.3.4
+- org.hamcrest:hamcrest-library:jar:1.2.1
+- junit:junit:jar:4.10:test
+- org.spread:spread:jar:4.0.1:compile
+- commons-codec:commons-codec:jar:1.6:compile
\- ee.ttu.cs:antlrxta:jar:1.1:compile
   +- ee.ttu.cs.dtron:xtaapi:jar:1.0:compile
   \- org.antlr:antlr-runtime:jar:3.4:compile
         \- ...
```

Figure 5.4: Full list of dependencies generated with *mvn dependencies:tree.*

possibly support bundled TRON executables for unpacking on-demand. Note that the bundling is not allowed due to redistribution restrictions inherent in Uppaal license. Therefore, this module will seek to locate pre-installed binaries by parsing the TRON_HOME environment variable on Linux or %TRON_HOME% on Windows.

org.spread:spread is also a mavenized packaging of the original Spread Java API to support its use through declarative dependency management by Maven.

ee.ttu.cs:antlrxta module implements the parsing of Uppaal model source XML files using ANTLRv3. Abbreviate from "ANother Tool for Language Recognition", is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages[79]. During the initial releases this module relied on simple string processing though.

ee.ttu.cs.dtron:xtaapi is responsible for the actual loading of model files. It keeps an in-memory virtual file system and allows input models to be in the form of strings. Its generic mechanism allows model loading and reloading over network sockets for the purposes of online model (re-)loading. Naturally it allows model loading from command line by specifying the model file in the usual way.

Whenever Maven resolves project dependencies it looks for the binaries and their specific version from a *Maven repository.* The default built in repository is the *central* repository located at http://repo1.maven.org/maven2. When the artifact is found it is cached locally at user home in *.m*2 folder. This is one-to-one copy of the original repository and the layout without the unused artifacts. The next time a dependency is resolved the cache is consulted first.

Maven also supports *private* Maven *repositories.* Private repositories do not necessarily need to have restricted access though. They are just "not" the central one. The repository serving DTRON and all relevant artifacts is located at http://lab.cs.ttu.ee/maven. The most straightforward way to configure a Maven-enabled software project to include DTRON is to first declare the location of the dedicated repository and then the dependency itself as depicted in Figure 5.5. Any transitive dependencies DTRON itself is dependent upon are resolved automatically, resulting in a dependency tree similar to one shows in Figure 5.4.

```
<repositories>
  <repository>
    <url>http://lab.cs.ttu.ee/maven</url>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>ee.ttu.cs</groupId>
      <artifactId>dtron</artifactId>
      <version>4.14</version>
  </dependency>
</dependencies>
```

Figure 5.5: Maven pom.xml configuration to use DTRON.

Note that *pom.xml* (Project Object Model) file is the XML based project specific Maven configuration file holding information in a scripted executable. That is the way to manage most aspects of a software lifecycle - compilation, pre- and postprocessing, unit- and regression testing, reporting, packaging, and publishing.

Maven splits the build lifecycle into several phases that are executed in a succession. Build lifecycles are generic and customizable by nature. We now list the major build lifecycle phases for packaging the final executable and briefly describe how DTRON is configured to take advantage of each of them:

process-resources phase allows filtering and processing various resources and configuration files prior the actual compilation. This allows the use of variable placeholders in resource files that are propagated based on the *pom.xml* configuration. We use this to bundle the exact compile time project version number (the *versionId*) into the *target artifact* (the executable). This simplifies troubleshooting when bugs are found. When *releasing* the executable to be publicly available during the *deploy* phase we also tag the source revision in the source code management (SCM) tool with this *versionId* that enables a straightforward association between different executables to the code base. This makes it easier to track down possible software *bugs.*
Process-resource phase is also used to generate Google Protobuf *class definitions* from .proto files (exact definition is shown in Figure 5.7). Without the generated classes the classpath would not be complete and we would be unable to compile. Note that this kind of automated class generation also serves the purpose of regression testing. This is especially important when changing the protocol definitions or upgrading the class generator (Protobuf to be exact).

compile phase compiles the source code, including the generated class files from *process-resources* phase.

test-compile compiles the unit tests source files.

test phase executes the compiled unit tests and also servers regression testing purposes as the tests are executed each time the build is executed. The main code base has 3421 lines of code and the unit test suite 1534. The test suite archives overall average code coverage 71.6% while the core code coverage is mostly 100%.

package phase assembles the final build targets. There are actually two separate build targets. One is the minimalistic build without the external library dependencies embedded. This is the natural way of assembling libraries and convenient way of embedding DTRON into other software

projects. The dependencies are resolved based on the descriptors in *pom.xml* when the appropriate parent project is built.
The second target is all the dependencies embedded and the resulting class collection reduced. The reduction removes unused classes from the final build. This build is intended to be executed from the command line. For example - to spawn multiple local testing agents for distributed testing. This all-in-one bundle could also be used as a library - similar to the first target with the difference that no other library dependencies are required to be managed.

install phase caches the built artifacts to the local maven repository in the user home .m2 folder for further use. This is a required complementary phase for the following *deploy* we make use of.

deploy phase uploads the assembled artifacts to the Maven repository so they could be resolved by Maven builds for other developers. The details of how to configure Maven to resolve these dependencies from another project are given in Figure 5.5.

Maven build automation system is highly extensible and configurable. It incorporates a plugin-based architecture to enable a natural way to extend build behavior for various goals. DTRON build makes extensive use of its web site-generation capabilities. Site generation is used to statically generate the project website at https://cs.ttu.ee/dtron with all the appropriate documentation, generated directly from source code where appropriate.

5.4.3 Spread toolkit

DTRON utilizes the *Spread toolkit* for implementing the message interchange. The messaging pattern is generically named as *publish-subscribe.* Local testers subscribe for messages and get notified when new messages are available and get notified by callback methods.

The Spread toolkit provides a high performance messaging service that is resilient to faults across local and wide area networks. Spread functions as a unified message bus for distributed applications, and provides highly tuned application-level multicast, group communication, and point to point support. Spread services range from reliable messaging to fully ordered messages with virtual synchrony delivery guarantees[80].

Spread comes with official support for Java, C++ and Python programming languages all of which have been used by other case studies not covered in this thesis. The C++ library could also be used by programming languages having a “native” bridge to call C++ library functions. This basically restricts DTRON runtime environments to languages that are able to bridge the Spread library.

During the early stages of the project the *Thrift* and *JGroups* messaging frameworks were also studied. JGroups is extensively used in large Java based software systems (e.g Liferay portal server), scales well and its highly configurable. But lacks the support for accessing messages in other programming languages as Java which does not fit the vision.

Thrift on the other hand was an excellent alternative and very similar to Spread toolkit. Thrift also comes with its own data (protocol) definition language that is used to (pre-) generate classes. It was compatible with Java, C++, Python and has been extended to also support various other languages by now. Thrift was replaced with Spread during the early phases of DTRON development only because it turned out the C++ implementation was design for Linux platform and did not work on Windows. But early case studies in the Scrub Nurse Robot project involved the use of Windows and Microsoft Visual C++.

API Spread toolkit API design is conceptually split in two. *SpreadGroups* (groups) are used to define publishable *SpreadMessages (*messages). Each group has a name and a set of subscribed members. Members subscribe by *joining* the group or unsubscribe by *leaving* it. Note that members do not necessarily be a member of a group for publishing messages to it.

Commonly to this type of messaging Spread also relies on a *messaging broker* (broker) to handle the actual runtime. Broker is a server-like program that binds to a network socket to accept requests. Members connect to a broker to subscribe to groups or to publish messages. All the runtime overhead is handled by the broker. This involves keeping track of the groups and distributing published messages to subscribers.

The broker is also responsible for the actual message queuing mechanism. This is to make sure the messages arrive and leave groups in the appropriate order. The order is defined by message publishers and set in each message attribute. Spread has 6 levels of message delivery guarantees that can be set:

(i) *Unreliable* messages have no ordering. The message could also be dropped and will not be attempted for recovery.

(ii) *Reliable* messages improve unreliable messaging in a way that all messages will be delivered. Spread will also try to recover from when information is lost.

(iii) *FIFO* (by sender) - all messages sent by the sender (publisher) will be delivered in FIFO order to each subscriber (group member).

(iv) Casual - all messages sent by all senders are delivered in an order consistent with Lamport's definition of "casual" order. This order is consistent with FIFO.

(v) *Agreed* messaging achieves total order and is inherently consistent with casual messaging.

(vi) *Safe* (total order) - all messages sent by all senders are delivered in the exact same order to all recipients. It is provided by making the partial order defined by casual into total order. The total order uses the id of the sender to break ties.

Note that whenever a message is published to a group it is transported to all nodes that have joined the group at that time. After the published message is delivered to all nodes - it is purged from the queue. So, the messaging queue is not persistent.

DTRON wraps the Spread API and takes care of configuring the group memberships and message configuration. Note that each message is flagged as "safe" to gain the guarantee of total order and not possibly break the underlying Uppaal timed automata semantics.

Configuration As mentioned in previous section the DTRON is packaged in two different ways - *embeddable* and *standalone* (executable bundle). During the standalone execution the Uppaal model file given for input is parsed and looked for channel variables prefixed with "i_" (inputs) or "o_" (outputs). The naming convention is given from the "adapter" perspective - inputs to the adapter (of the system under test/control) and outputs from the adapter (inputs to model).

Whenever an input prefix is found it is registered with Uppaal TRON API to get notified when it occurred. Also the appropriate SpreadGroup is created at the broker or joined if it already exists. Whenever an input channel is then triggered at the model, its name and the associated variable values are published to the appropriate group. Integer variables can also be associated to a publishable channel by prefixing their names with the corresponding channel name. For instance, having a channel "i_test" and an integer variable "i_test_n" then a message to a group "test" is send with attached variable "n" with its value.

When an output prefix is found in the model it is registered to be an output in the adapter API. Whenever a model reaches a state with outgoing transitions labelled with output channels the channels are bound to be disabled until they get triggered by adapter and become enabled.

For each output channel a corresponding Spread group is subscribed to. Whenever a message is then published to this group the adapter triggers the correspondingly labelled transition to be enabled at the model. Similarly to incoming channels, outgoing channels could also have associated integer variables.

Data Spread messages encapsulate three types of information: firstly, the list of groups the message is intended to be published to; secondly, the attributes to configure the message handling (like order); thirdly, the *data*.

The data field is defined in the form of *byte*[] - a byte array. This makes it a universal way to transport any type of data in a platform independent way. The problem is how to construct the byte array in a way it could be unambiguously reconstructed afterwards. High-level programming languages provide minimalistic native support for *serialization*. It allows data structures and (class) objects to be converted into byte arrays for later de-serialization back to their original form. Note that each language has its own different serialization mechanism.

DTRON is intended to be platform- and language independent. But serializing data in one language is rarely de-serializable in another. There are many ways to approach this problem. Whatever the solution is, it needs to be fast. So we need a lightweight serialization approach, rather than a full-blown XML transformation engine.

5.4.4 Google Protobuf

There are many (fast) serialization frameworks available. The main criteria for choosing the appropriate one was that it needs to be as fast as possible and support language bindings for at least Java, C/C++ and Python. A free or academic license is a must.

The initial starting point for choosing the one was the 'jvm-serializers' project, now moved to GitHub[6]. The project implements repeatable benchmark tests between different serialization tools available. Figure 5.6 represents an extract of their results showing object creation, serialization and deserialization total round-trip time in nanoseconds.

Since we are expecting DTRON runtime to operate with around 1 millisecond granularity which was heuristically a realistic expectation, a microsecond scale serialization seemed to be enough. Protobuf has official support for Java, Python and C++ language bindings with third-party developed tools for additional languages around 30.

Protobuf relies on an intermediate data definition language (a "protocol") to define message types and structure used for serialization. This language uses generic data types and structures not specific to any single programming language. The protocol definition *.proto* file is used to declare data structure definition. Then a *protocol compiler* statically compiles it into any supported programming language binding. These bindings are a set of generated class files that can be imported into software projects. Within the DTRON project - language bindings for Java, Python and C++ (both Gnu and Microsoft based) have been used and known to work.

Figure 5.7 shows the .proto definition file used in DTRON to transform

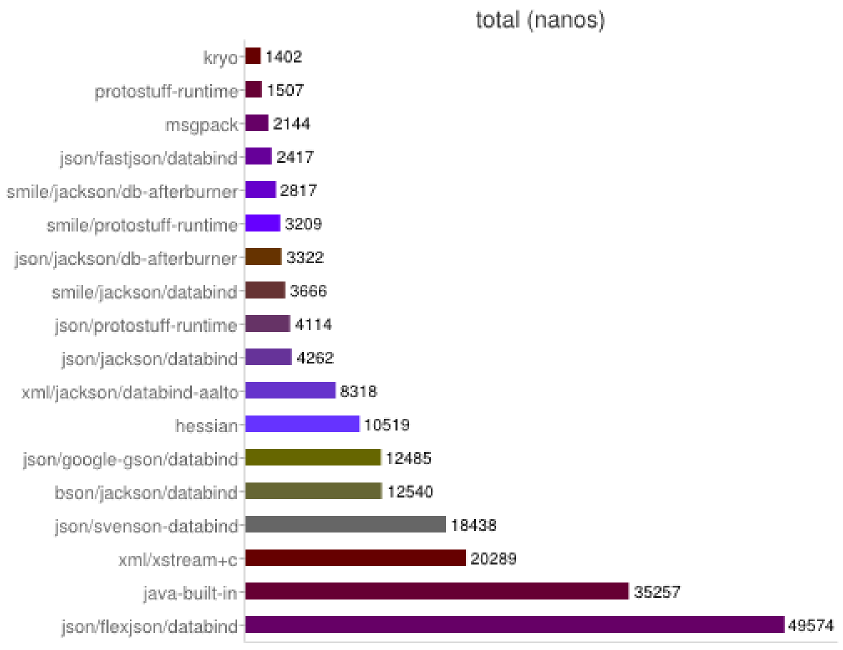


Figure 5.6: Serialization benchmark.[6]

channel and variable information into a byte array to be published to a Spread group. There are some special aspects to be emphasized.

Firstly, the data type *sint*32 represents a 32bit signed integer value. It is beneficial when interchanging integer type values between Java and C++ nodes. Java does not have "unsigned" data types, while C does.

The second aspect here is that Protocol Buffers (Protobuf) is a *"versioning protocol"*. Within each data structure an index is assigned. Whenever a de-serialization is needed this information is used to extract the individual fields. When a new version of this protocol is developed by adding an extra field to a structure it does not "break" the de-serializer receiving data compiled with the new version having more information in it.

This makes it convenient to preserve any "backwards compatibility". Note that this was also a contributing factor when choosing between Thrift and Protocol Buffers.

```
message Variable {
  required string name = 1;
  required sint32 value = 2;
}

message Sync {
  required string name = 1;
  repeated Variable variables = 2;
}
```

Figure 5.7: DTRON Protobuf protocol definition.

5.4.5 Architecture

Conceptually, DTRON could be thought of as a wrapper around Uppaal TRON to ease its use and flatten the learning and adaptation curve. The core functionality is written in Java, but it incorporates frameworks like Spread and Protocol Buffers to interface the runtime with other major languages and platforms, e.g. C/C++ and it's Microsoft variants, Python, etc. The core functionality, being written in Java, also allows the execution on any platform supporting Java. This essentially makes DTRON platform and implementation language independent.

DTRON is written in a way it could easily be embedded to other larger/-complex systems. On Java based Maven enabled software this is especially easy as demonstrated in Figure 5.5.

Figure 5.8 gives an overview of the architecture and shows how the data flow changes compared to Figure 5.3. The Adapter is automatically generated and the related information published to Spread multicast network by handling input and output prefixes as described in the previous section. The byte-level data traversing the Spread network is automatically serialized and de-serialized using Google Protocol Buffers.

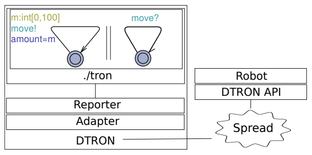


Figure 5.8: DTRON runtime data flow.

Figure 5.9 shows the prefixed model suitable for DTRON. Prefixing scheme allows automatic adapter generation for interchanging the event information between models. Whenever an input-prefixed channel “i_move” occurs in the model it is published to a “move” Spread group. All clients subscribed to this group will get notified of this event.

Attaching global integer variables to a channel is done by prefixing a variable name with channel name. So if we would like to accompany variable amount with channel “move”, we would define a channel “i_move” instead and the corresponding variable to “i_move_amount”. The variable name will also be automatically de-prefixed and the resulting message in Spread network would be channel “move” with attached variable “amount”.

Like with input channels (alphabet), output works similarly. Prefixing a channel in the model as “o_” will mark this channel as output. Whenever Uppaal TRON reaches this transition the execution will be blocked until this event is reported in to the model. Using DTRON this means the appropriate message would need to be submitted to Spread network. For instance, (returning to the running robot example), to get notified when the robot “move” operation completes (an asynchronous/blocking action), wildcard a channel “o_done” and whenever the robot reports the appropriate message to Spread the model execution will proceed.

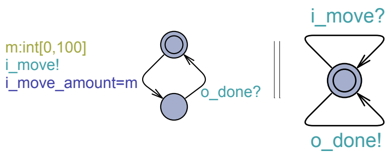


Figure 5.9: Prefixed model.

Example 5.6.

It is possible to join two or more models into a “conversation” over DTRON. If one model would emit and publish an event “move” and will wait for “done” afterwards, another model might wait for “move” first and then emit “done”.

Figure 5.10 shows an API extract for joining a Spread group to get notified of events. The constructor requires information about the channel to be subscribed to. This includes the channel name and also incorporates the template for attached variables. The template only defines the attached variable names without the values. Whenever a listener gets notified of event the callback method *messageReceived* is invoked by DTRON and the details of the channel (name) and variables with assigned values passed based on the original template.

```
package ee.ttu.cs.dtron.api.spread;

public abstract class DtronListener {
  public DtronListener(IDtronChannel);
  protected abstract void
   messageReceived(IDtronChannelValued);
}
```

Figure 5.10: DtronListener

The details of the API will be covered in the next section.

5.4.6 Domain model (API)

Figure 5.11 shows the API domain model. Class *DTRON* is responsible for handling the connection to the Spread broker, i.e. allocation and release of related resources. This serves as a *main entry point* for the API since that follows - requires a connection.

DTRON connection is used to assign *DtronListeners.* Listeners are built based on specific *IDtronChannels* that hold the details about the specific channel - the name and possible variable assignments. Whenever a (IDtronChannel) matching synchronization happens and is published to a Spread group, the listeners get notified by DTRON invoking the callback method *messageReceived(IDtronChannelValued)* and passing the specific values in the corresponding object as an argument (detailed description is given in Figure 5.10.

Note that when a listener is constructed it needs a "template" of the channel of interest. This template holds the channel name and the list of variable names assigned to it. When the callback is invoked this template is used to construct a "valued" version of this template to hold an immutable map containing the same variable name, but now with explicit values.

Figure 5.11 illustrates such a domain model.

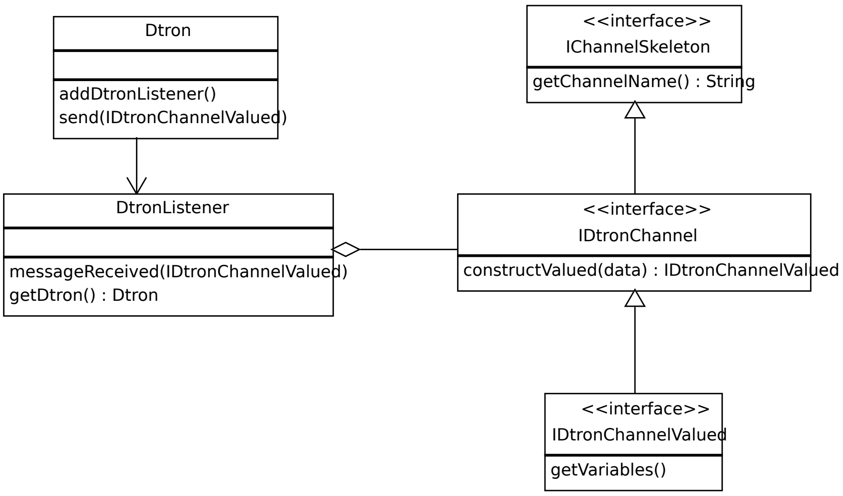


Figure 5.11: Domain model.

To publish events to a Spread group, the *Dtron* class provides a method *send*(). The required argument object cannot be constructed directly. IDtron-Channel "template" has to be constructed first to declare the channel name and related variables. The resulting template object holds the appropriate *constructValued*(*data*) method to assign concrete values to the variables. The assignment is cross-validated to the variable list declared in the template to avoid illegal assignments.

This *constructValued* method actual has two different signatures. Firstly, it accepts a *map* of variable names and associated values that is intended to be used by users to construct "valued" channels. The second signature accepts a *byte*[] array and is mainly used internally to de-serialize the corresponding *byte*[] field contained in the Spread messages.

While implementing a *DtronListener*-s as anonymous inner classes or a subclass, there is a convenience method *getDtron*() to get a handle to the *send*() method for immediate inline reply back to *Spread*. Otherwise, one needs to take care of the access to DTRON connection manually and this would introduce unnecessary overhead and suppress code readability.

The infrastructure code in *DtronListener* constructs a new object based on the provided template with immutable map of variable names and values for intentional use and eliminates direct access to underlying (immutable) pointers. So accidental variable manipulation would not cause the runtime to crash.

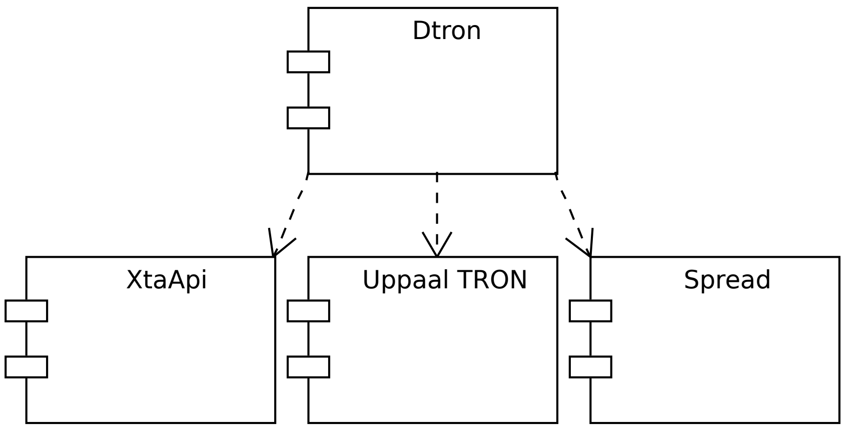


Figure 5.12: Component diagram.

5.4.7 Distributed execution

In distributed testing there are multiple *physical ports* for interactions between tester and the SUT (also called points of control and observation or PCO). We also distinguish that a DTRON instance running/testing on one port serves as a *local tester* and the publish-subscribe messaging allows the *observation* of a *global trace*[81].

Figure 5.13 gives an outline of a distributed testing deployment scenario with DTRON. Where local testers are interfaced to SUT ports via adapters and subscribed to their corresponding Spread broker. There can be many brokers while preserving the "safe" total order over all brokers. DTRON binds its communication socket to a specific broker to publish and subscribe for messages. Spread brokers can have a complex topology for connecting to each-other. Spread itself takes care of the network route discovery and planning. So a message published to one broker can be received by a subscriber to another broker in another network segment.

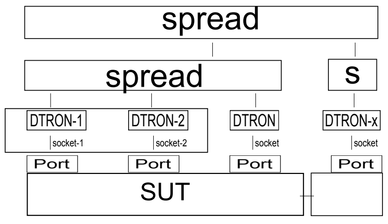


Figure 5.13: Distributed testing data-flow in DTRON.

Uppaal TRON has two ways of API integration: a native C library API and a network socket based connection. DTRON currently uses the socket based interface since it provides "out-of-the-box" support for Java integration, although technically it is possible to bind Java code directly to C by using a Java Native Interface (JNI).

There are two benefits when using a native C bridge. Firstly, the the speed is much faster and method invocation in Java takes essential time, whereas socket messages need to traverse the networking stack in the ISO Open Systems Interconnection (OSI) sense[82]. Secondly, we need to allocate a network socket that introduces unnecessary overhead when running multiple instances of DTRON in the same machine. The same socket cannot be bound to more than one program so we would need to enumerate the port numbers to be unique for every instance.The socket-1 and socket-2 shown in Figure 5.13 illustrate this situation when the corresponding DTRON instances are running on the same machine. The default port number is 666. If another instance is run this would have to be overridden with command line parameter $-p$, when running in embedded mode using the corresponding API $setPort(int)$ method.

The downside of the C-bridge is that it does not support *virtual clocks* which are only supported by the socket based interface to Java adapter. Virtual clocks are a mechanism to agree on how time passes. For example, if the tester must be bound to wait for an hour until next input delivery and doing nothing else, we could programmatically wind the clock ahead by one hour and go on with the testing.

Virtual clocks would also allow the use of Δ-testability [34] that will be discussed in the future works section8 below. This addresses the problem of non-uniform message transmission delays between distributed DTRON testers. Consider the Figure 5.13 example. We would normally expect that if DTRON #1 publishes a message at time point t_1 and DTRON #2 after that at t_2 then $t_1 < t_2$. But if $DTRON\#1$ exhibits an internal delay longer than $DTRON\#2$ it could happen that t_2 is actually published before t_1 and therefore $t_2 < t_1$ instead, thus leading to a conformance violation with the model. But with DTRON we could actually measure the delays at the Adapter level and use virtual time to "agree" that $t_1 < t_2$ even if it was actually $t_2 < t_1$ instead. We refer to Δ as the *time interval* during which we allow events to be swapped in this manner.

5.4.8 Selenium

In this section we demonstrate how DTRON interfaced with Selenium can be applied for (distributed) web application testing. The goals are two-fold. Firstly, we can use model based testing in a traditional way and check for functional properties of the system. For example, if we log in to the system and check if we can perform operation and observe valid outputs we define an appropriate model and execute it with DTRON.

Secondly, we can use the same model and execute multiple instances concurrently. Given the model is parametrized by user we can perform load-testing with arbitrary number of concurrent users. Meanwhile, the non-deterministic nature of the automata gives us a convenient way to model erratic user behavior. Since we use a timed formalism we can also assert for performance properties, like checking if the system gives an output within the required time.

Figure 5.14 shows an example test adapter performing a "login" test against a SUT. The SUT is basically a web application that allows a user to log in. The adapter utilizes the Selenium browser automation framework to implement the actual interface. It allows a programmatic access to a web browser (like Mozilla Firefox and Google Chrome) to execute various operations that a typical user would do: opening a web page, entering text, clicking on buttons etc. Testing a web-application through a web-browser is important because it also accounts for the cascading style-sheets (CSS) and JavaScript execution aspects of the output. Testing static HTML output or doing class-based unit tests might hide faults that may occur when the browser renders the final output with CSS and executes the (client side) JavaScript.

Whenever the model publishes a "login" synchronization it will be delivered to this adapter. Note the *AbstractSeleniumAdapter* that the adapter extends. This abstract parent class takes care of setting up the underlying DTRON. It subscribes to events equal to the implementing subclass, i.e. "login" at this case and delivers any corresponding publication in to the *execute(IDtronChannelValued)* method just as described in Section 5.4.6.

```
public class Login extends AbstractSeleniumAdapter {
 void execute(IDtronChannelValued v) {

  // assert we're seeing the login screen
  WebElement loginButton = findInput("LOG_IN");
  assertThat(loginButton, notNullValue());

  // fill username & password
  getWebDriver().findElement(By.name("username")).
    sendKeys("user");
  getWebDriver().findElement(By.name("password")).
    sendKeys("pass");

  // submit
  loginButton.submit();

  // assert we're logged in and seeing log-out button
  WebElement logoutButton = findInput("LOG_OUT");
  assertThat(logoutButton, notNullValue());
  }
}
```

Figure 5.14: Basic "login" test adapter.

DTRON emphasizes a clean API where the developer only needs to implement the adapter to interface the SUT, eliminating the overhead of channel-variable bookkeeping that clutters code. *AbstractSeleniumAdapter* serves as a demonstrator of how DTRON is extendable to include other frameworks like Selenium.

Figure 5.15 shows a simple model to match the described adapter. The upper automaton defines the test and the lower one the SUT interface. Tester starts with initiating a synchronization *i_login* that is published by DTRON being prefixed with "i_". The lower automaton completes the synchronization to enable this transition and enters a location labelled "wait". The outgoing transitions are labelled with *o_result* synchronization that makes them disabled until the appropriate synchronization is triggered by DTRON when receiving the corresponding subscribed *result* signal.

The results synchronization is also decorated by an attached variable to encode the result code of the operation. The assignment to this result code is required formally in the model too. When we report a value "1" for a variable *o_result_rescode* its value is compared with the value this variable currently holds in the model. If the values differ we have non-conformance between the model and the SUT because the (inner) states do not match. So we have to (formally) assign the reported value in the model, by the time it is actually sent into the model by the adapter.

In order to make this happen we decorate the SUT model with an assignment. We make use of the *select* operator to allow non-deterministic assignments. This example assumes the value “0” stands for a successful result code and any other value to serve as an “error code”.

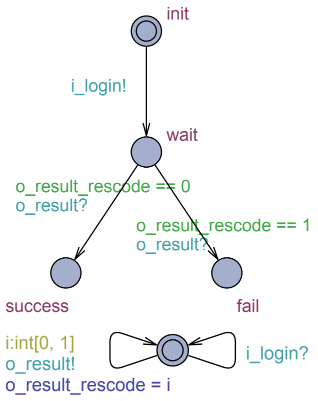


Figure 5.15: Simple model for Selenium testing.

It would be possible for DTRON to initiate the o_result channel any time, but if the tester automaton is not in the appropriate state to execute the enabled synchronization action that would result in conformance violation with SUT.

The sample adapter code in Figure 5.14 does not reveal what implements the result code assignment because this is implemented in the AbstractSeleniumAdapter provided by DTRON distribution. This abstract adapter invokes the *execute* method and if it returns normally, the result code is set to “0”. If it terminates abnormally (e.g. and exception is thrown) the result code is set to “1”.

The idea of Selenium integration is elaborated in the Tartu City Lighting Controller case study project described in Section 7.3.

5.5 Summary

This chapter presented the details of DTRON architecture - a tool based on Uppaal TRON enabling distributed execution of Uppaal timed automata. It provided the details of the subsystems and the communication model to present how it is built.

During the development some modeling, design and runtime limitations and considerations unrevealed and were listed to serve as a user/developer manual.

6 Performance evaluation

6.1 Chapter overview

Since DTRON is intended to be a real-time testing tool it is important to investigate its performance in real-time applications. Every abstraction layer added to the architecture to simplify the use or extend functionality is expected to have a computational impact and therefore introduce latency. This latency could lead to non-conforming testing results that wouldn't exist if there would be no latency.

One of the most important DTRON design principles was to have minimal timing overhead during execution while we allow computationally expensive setup before the actual execution.

This chapter presents the results of this investigation carried out by the author.

6.2 Introduction

The goal of this analysis is to determine the latency overhead DTRON has when introducing an extra layer of messaging abstraction. The focus is on measuring the effect of the Spread toolkit as messaging service with the combination of Google Protocol Buffers - a language-neutral, platform-neutral, extensible mechanism for serializing structured data. The latency benchmarking is done in three different execution environments to demonstrate the scalability with respect to different application constraints. The experimental results are presented and extreme cases explained. Finally, the conclusions made on the experimental results have been drawn to clarify the applicability limits of DTRON tool.

6.3 Experimental setup for performance evaluation

The experiment setup is based on the same latency analysis models that comes bundled with the UPPAAL *TRON* distribution, specifically the *Ticker* study. So the experiment would be independently reproducible. A *Ticker* is an UPPAAL *timed automata* (UPTA) model (see Figure 6.1) that executes a clock tick every certain time interval. The time interval is designated with variables $p = 250$ and $t = 50$, a guard condition on a reflective transition and a location invariant that forces a tick on average every 250 UPTA clock units with deviation within time interval $t_d = [-50, 50]$.

The experiment measures synchronization channel reports (messages) arriving to the TRON Java API as a baseline and then measures the extra delay

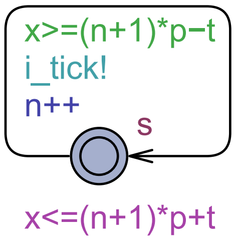


Figure 6.1: Ticker UPTA model.

it takes to pass this information through DTRON. Figure 6.2 outlines the data flow and timing points. Messages first arrive at the Spread-adapter and t_1 times the event. The second timing t_2 is taken when the message arrives at DTRON adapter. The difference $t_\Delta = t_2 - t_1$ is computed and then analyzed over time. We execute this sample model with DTRON only in eager mode - producing maximum stimulation. Then we measure the incoming synchronization times arriving to TRON Java adapter.

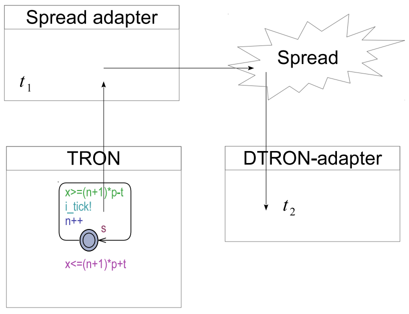


Figure 6.2: Experiment setup data flow (architecture).

6.4 The results of evaluation experiments

This chapter present the evaluation results with interpretations and possibly future work.

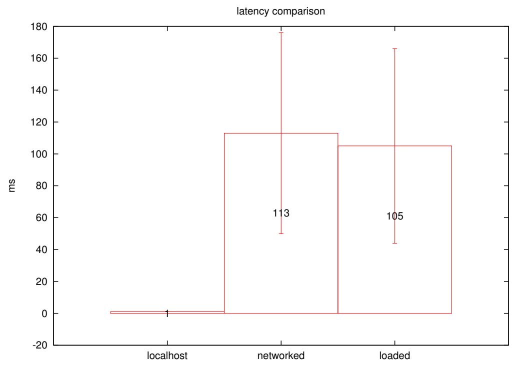
Figure 6.3 shows the results of latency benchmark results in three different execution environments:

(i) Messages transmitted over a network loop-back interface. That is the Spread broker is ran on the same machine as the Adapter.

(ii) Messages transmitted over a switched $1Gbps$ Ethernet network.

(iii) Messages transmitted over a loaded $1Gbs$ network with 50% of the bandwidth allocated by “Distributed Internet Traffic Generator”[83].

Figure 6.6 shows the same results on a graph where β is the mean, σ is the standard deviation and $t_{\triangle_{max}}$ is the maximum latency in milliseconds.



	β	σ	$t_{\triangle_{max}}$
loop-back	1	0.1	2
network	113	64	221
loaded	105	61	220

Figure 6.3: Latency results.

Figure 6.5 shows empirical results of an (Windows) operating system networking stack showing some form of caching symptoms where mean latency drops to a consistent $1ms$. Systematic attempts to reproduce this scenario under controlled environment have failed. Source data have been taken from a testing session log to give some idea of its nature.

To compensate extreme fluctuations the future plan is to incorporate some diagnostic loop into DTRON to work around this since this could lead to false non-conformance errors at the worst case. We will refer to this as a Δ testability problem in later studies. This has already been briefly covered in section 5.4.7 above and will be elaborated in the future work section 8 below.

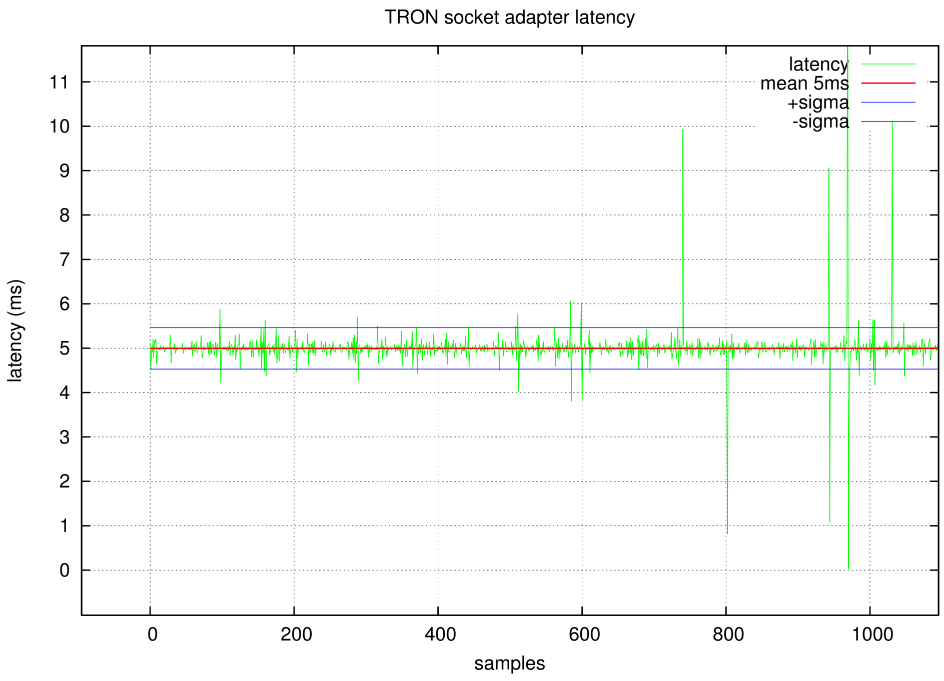


Figure 6.4: Eager UPTA at $5ms$.

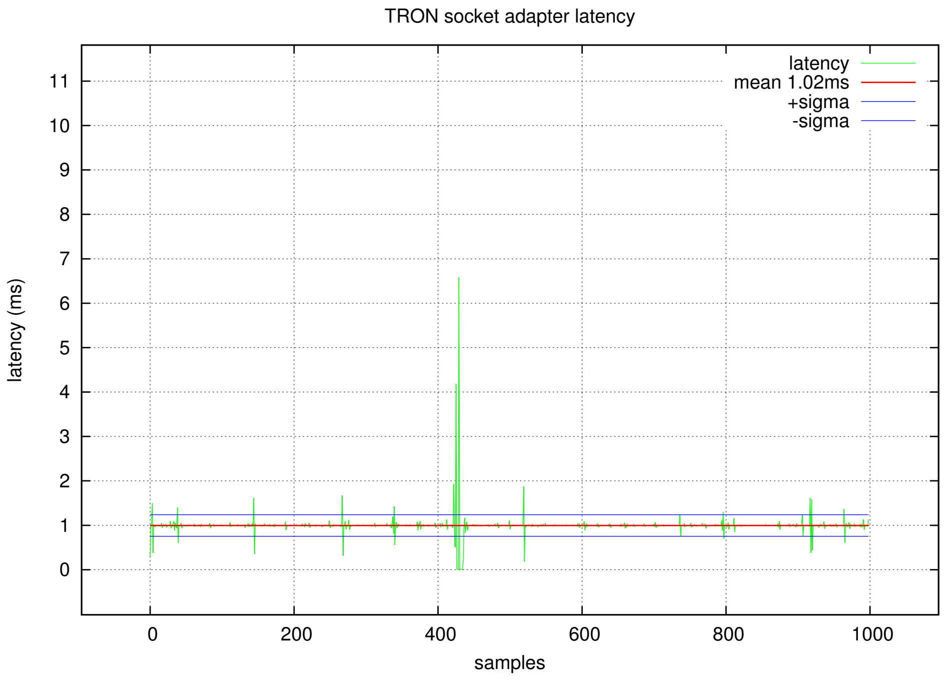


Figure 6.5: Anomaly at $1.02ms$.

Figure 6.6 shows aggregated latency results showing correlation between the latency and computation time. Computation time is emulated to $1ms$ by utilizing $sleep(1)$ for simplicity.

Instant low computation events overwhelm the networking stack. This in turn causes packet buffers to fill and maintenance subsystems try to compensate. This changes in network throughput and occasional bursts/peaks are to occur. Note that *"instant computation"* ($0ms$ sleep) is for informational purposes only. Stress test results in Figure 6.4 show a mean latency of $5ms$ while stimulating events with UPTA at maximum capacity. Even with anomalous behavior denoted in Figure6.5 the stimulation computation time is known not to fall below $1ms$ by experimental results.

Note that minimal $1ms$ latency lag is caused by the TRON Java adapter used internally. This Java adapter communicates with TRON executing an UPTA model over a *SocketAdapter*. That is because of latency due to the networking stack. An alternative way of interfacing the adapter is using the C library level callbacks which comes down to a single function call. This case has been covered earlier in Section 5.4.7.

Given a computationally non-intensive function the execution can be considered to be instantaneous. This would result in heavy packet-intensive traffic of messages to Spread network. Since the *Nagle's algorithm[84]* has been turned off for lower latencies this would result in huge overhead of decorating the actual message with TCP/IP packets and in significant loss and fluctuations in throughput. A sleep interval of $0ms$ in Figure 6.6 illustrates this scenario.

Figure 6.7 shows sleep-vs.-latency analysis when using $nanoSleep()$ instruction for computation simulation. It allows a nanosecond scale control over thread blocking time instead of milliseconds - that is with regular *sleep*. Although both functions seem to implement the same thing, the *nanoSleep* is computationally more intensive. A marginal increase in such computation time would result in substantial drop in latency and it's fluctuations.

Since this stress test runs fast the experiment is carried out with $10K$ samples instead of $1K$ previously used. This is to demonstrate how it would scale after $1K$.

Note that *nanoTime* uses processor core ticks to count time instead of *realtime clock* module. Although processor tries to coordinate core times to be equivalent, it is not always guaranteed to be. It is often the case that processor cores have minor (nanoscale) differences. Given the nature of processor coordination that individual tasks are distributed around cores, it might happen that the difference between consecutive *nanoTime()* readings turns out to be negative!

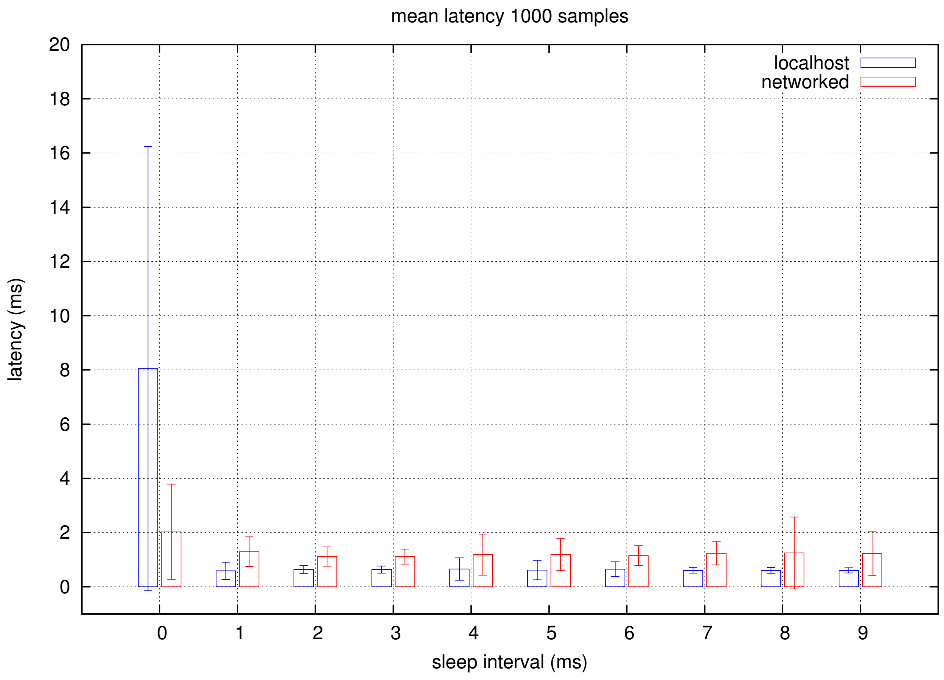


Figure 6.6: Aggregated latency results comparing *"computation time"* against latency.

6.5 Summary

This chapter presented the results of computational (and timing) overhead analysis of DTRON. The author designed DTRON specifically to reduce the runtime overhead and the practical experiments assure it to have been achieved. The resulting framework capable of operating in the millisecond scale is of reasonable precision for most applications - including time-critical robotic applications this theses was motivated from.

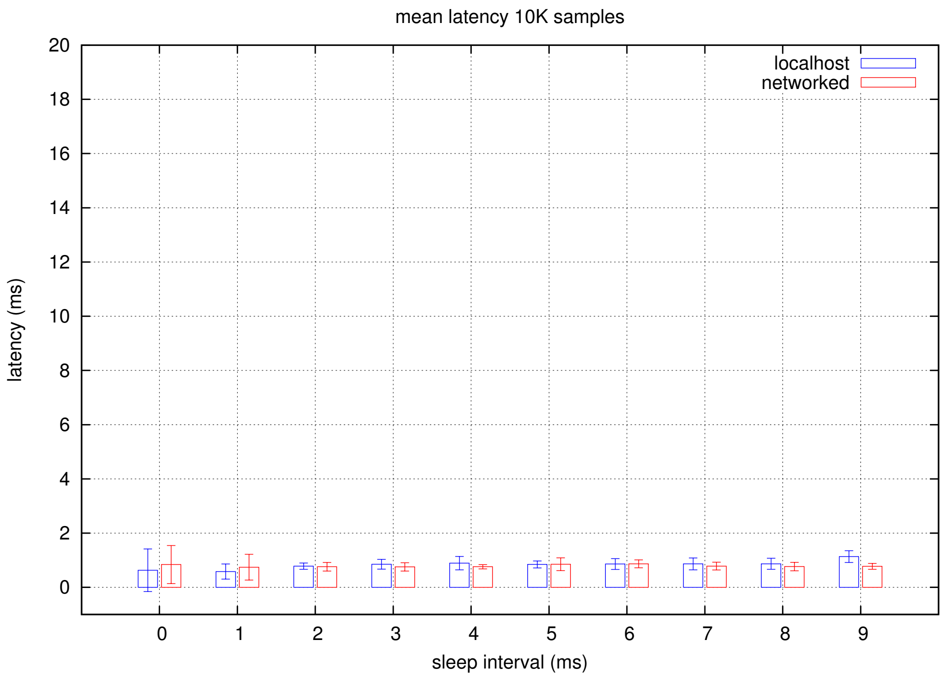


Figure 6.7: Aggregated latency using *nanoSleep()*.

7 Case studies

7.1 Chapter overview

In this chapter the applicability of DTRON framework and model-based techniques developed around it is demonstrated by describing two representative case-studies: Scrub Nurse Robot (SNR) model based control and a distributed performance testing of Tartu city street light control system. The SNR case study highlights the features related to handling real-time and safety constraints in robot high-level action control. The street light system case-study focuses on aspects of coordinating distributed systems and on how to generate test configurations for distributed web applications.

The contribution of the author in the Scrub Nurse Robot project is the analysis and development of JSNR (later DTRON), the corresponding automata models, the execution and result analysis.

The contribution in the Tartu city street light control was testing the prototype distributed system by automata modelling, the execution with DTRON and result analysis. The author was not involved in the development of the system under test, but was introduced to the project in final phases with the goal of testing the system as a "black box".

7.2 Scrub Nurse Robot

The SNR project was initiated as a sub-project in *Human Adaptive Mechatronics* (HAM) *Center of Excellence* at *Tokyo Denki University*. The overall goal was to build a robotic assistant to learn the interactions between a surgeon and a (human) scrub nurse during laparoscopic surgeries and then replace the nurse with a robot. This meant constructing the suitable robotic platform and developing the robot learning and control algorithms to go with it. The Denki University had extensive practice in the construction of robotic platforms, but was seeking for collaboration in developing the control algorithms. The *Department of Computer Science* (DoCS) of TUT joined the project to develop the learning and control algorithms.

The research started as a major case study, but as the project evolved, more and more shortcomings were discovered in the development tools used in the project. These shortcomings triggered the development of more generalized framework to surround the existing tools and make their use as practical as possible in day-to-day development.

Figure 7.1 shows the SNR physical layout with the reference to robot's manipulator degrees of freedom (DOF)

The robot was built onto a "stage" that was mounted to a vertical pillar. The height and rotation of the stage were pneumatically controlled, but initially without a hardware limiter. So, giving a *malformed input* to the pneumatic

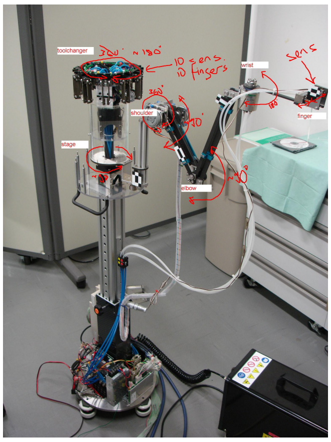


Figure 7.1: "MICHAEL" revision of the SNR.

controller the stage could be blown off over the top of the pillar. A similar faulty behavior could happen to the stage rotation also. Without a hardware limiter and with invalid controller input many rotations could result in that cables and pneumatic tubes would be forcefully teared broken.

The automata model-based control was introduced to avoid these situations. Specifically, by established *invariants* that would limit the input domain and the model of robot possible behaviors, the robot safety margins became formally verifiable before implemented in hardware. Extending the idea, also higher level *surgery scenario models* were intended to capture the succession of the most critical interaction events during the SNR application scenario. For example, if a robot picks up a scalpel, it would not (accidentally) release it before someone/something is ready to receive it. Another, safety critical, aspect would be to show that the robotic manipulator arm holding a scalpel always executes an "emergency stop" or back away if it was bound to get too close to surrounding humans.

7.2.1 Robot control software JSNR

The initial software platform for SNR control JSNR ("J" for Java) developed by the author of the thesis provided the proof-of-concept and motivation for the DTRON development later. JSNR framework was intended to simplify the application of the MBT tools for robotic control applications. It also made use of the Uppaal TRON framework and utilized its original adapter API directly.

JSNR provided the middle-ware to issue high-level controller commands to the robot directly from an Uppaal model without additional programming. It utilized the TRON API to prepare the Adapters to interface with the robot and defined a contract to be used when executing actions on the robot. For example, taking synchronized transitions labelled with channel *"closeFingers"* in the model would cause the manipulator arm to execute a grasping motion with the manipulator "fingerend-tips".

As shown in Figure 7.1 the robot has the following degrees of freedom (DOF): stage height and rotation, tool-changer rotation and 10 grasping sockets with sensors and status LED-s, 2-axis shoulder, elbow, wrist, finger actuators, with a sensor. JSNR implements the adapters for each DOF and maps the appropriate channel names to corresponding executable actions. Figure 7.2 shows an extract of the resulting domain model.

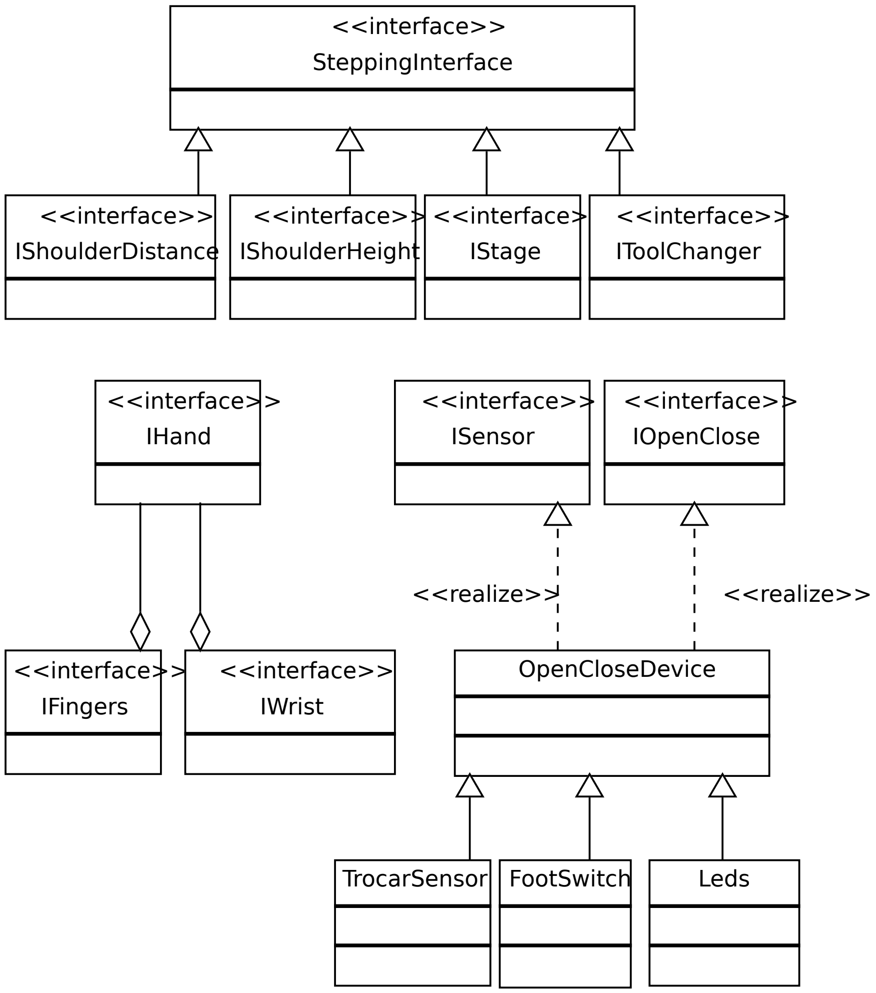


Figure 7.2: JSNR domain model.

The domain model is syntactically defined in terms of interfaces and provided with two separate implementations. The first implementation interfaced with the actual robot, but the second one with "mocked" robot (robot interface simulator). The mocked implementation allows off-site (model) development without the actual robot.

JSNR also provides a diagnostic screen as shown in Figure 7.3 to monitor the status of the robot during execution. The diagnostic screen allows online robot DOF status monitoring and also serves as a "virtual" robot for off-site use.

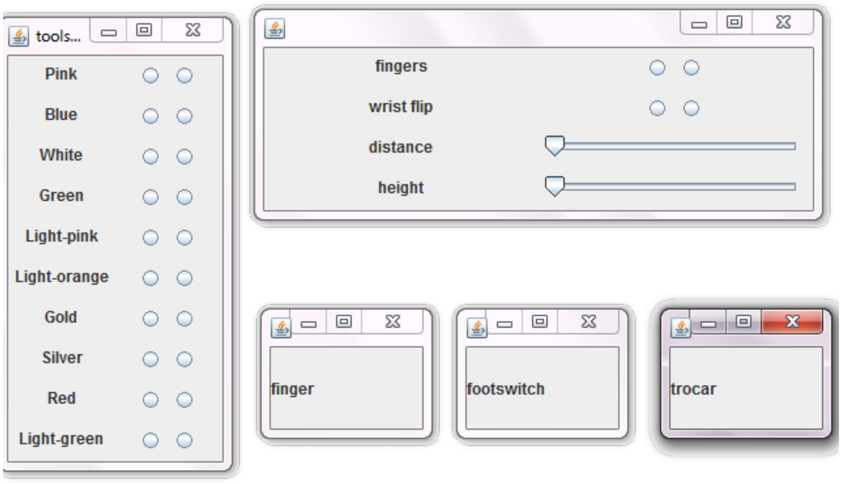


Figure 7.3: JSNR screenshot.

7.2.2 Model based API

In addition to the SNR direct control the the JSNR was also bundled with models defining the robotic platform features.

Figure 7.4 shows an extract of the model defining an invariant for the stage height control. Stage height is controlled in terms of "steps". Steps are actually instructions to the "stepping controller" driving a pneumatic actuator to raise or lower the stage pillar.

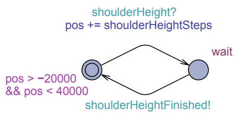


Figure 7.4: ShoulderHeight model.

If the step amount is signed to negative the platform will be lowered and vice versa. The early versions of the robot did not implement a hardware limiter for the maximum height. If the stage is in its maximum height and a positive number of steps is issued, the stage would be blown off the platform by the pneumatic actuator.

JSNR pre-modelled the degrees-of-freedom, established invariants and used model checking to verify that all the invariants would always hold. Figure 7.5 shows a conceptual example of modeling a “reaching” motion for the robot - manipulator arm extended and fingers open.

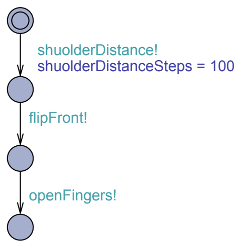


Figure 7.5: Reaching.

Invariants could be defined to assert that manipulator arm does not collide with itself or the surroundings. The SNR prototype set up also a 3D measurement system to track markers placed on humans. Making the marker tracking information available to the model enables to construct invariants to t avoid collisions between robot manipulator and human. Considering the inertial motion implied from the mass of the manipulator, an invariant is established to execute “emergency stop” protocol.

The other benefit of using model for robot control is that model control structure excludes some of conflicting actions. When modeling a manipulator joint like shown in Figure 7.4 it is implicitly assumed that the shoulder height manipulator could be started and stopped at the same time.

7.2.3 Results and conclusions

JSNR turned out to be relevant solution for single-developer projects and for prototyping conceptual designs, but lacked scalability. Modeling only the “?” side of the channel would require the developer to handle the pairing or the use of broadcast channels.

The control of robot functions using models and JSNR turned out to be excessively labour demanding when applied on low-level control due to the large number of implementation details that need to be addressed when interfacing directly sensoric and actuation functions. When applied to high level control functions and related state invariants the approach would allow higher level of abstraction in the model and better scalability of handling complex behavioral scenarios. This in turn, brought up the need for a proper messaging model to coordinate actions between multiple robotic agents and their environment. Models would serve then as abstract reference behaviors for online safety monitoring.

Having high-level coordination of distributed sub-systems like the robots, 3D motion tracking, external robot knowledge bases and other, that brought up the need for a proper communication mechanism. Every joint on a manipulator would be considered a self-contained subsystem, exhibiting only i/o behavior to be observed and controlled by other parties and keeping component local behavior implicit in the model. For instance, having a *local invariant* to stage height controller just rules out the stage "popping off" while leaving the internal stage control mechanism hidden. Similarly, when aggregating local components to coarser design items their models and invariants are derived compositionally from the ones of their constituent components. For instance, local invariants of the stage and the fingers conjoined would assert against possible collisions.

Figure 7.6 shows the execution architecture where the models run with established invariants and properties to be verified online. Models Mn hold the invariants (e.g. platform stage height), online model middleware intermediates messaging between SUT (components) and controller components Cn.

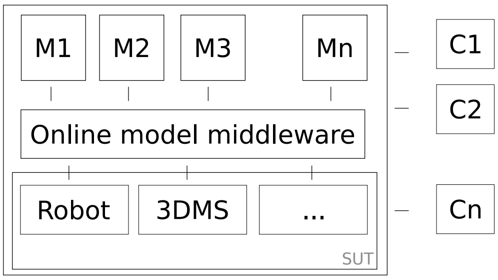


Figure 7.6: Extended model based control architecture for multi-robot systems.

7.3 Tartu city light controller project

7.3.1 Background

Testing the "Tartu city light-controller" (*light-controller project*) is an industrial size software/hardware development project performed in the Competence Centre in Electronics, Info and Communication Technologies - ELIKO.

The light-controller project aims to automate turning city lighting on and off or dimming in a "smart" way. There are many light-controllers localized to streets and public areas to keep the light conditions within preset range. Local light-controllers can be configured to turn city lighting - for example based on the atmospheric illumination threshold. So if it gets dark at one place in the city, only the lights of that place get turned on and not the other in different parts of the city. Naturally, the darkness usually arrives with nightfall, but could also occur with severe rainfall or any other natural phenomena, e.g. fog. Figure 7.7 shows conceptual deployment model for grouping lights under controllers where supervisory control of group controllers is implemented in the central server.

Controllers are low-power and -computation embedded systems. The controller communication medium is General Packet Radio Service (GPRS) over 2G Global System for Mobile communications (GSM). This medium introduces computation and communication delays that are hard to handle with non-distributed MBT methods due to the conformance problems caused by observation time uncertainties. DTRON and Δ-analysis were used to address conformance problems with extensive automated testing.

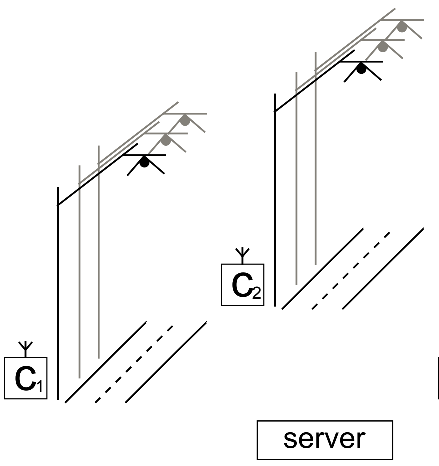


Figure 7.7: Light-controller project deployment model.

The controllers hold their status in-memory using a status string that determines the controller mode. The status string is a bit vector where bits encode the mode parameters the controller is following in given mode. The status string is encoded in *Base*64 form and is possibly reset when polling the coordinating server returns a different setpoint value to given controller.

7.3.2 Protocol

We now explain the communication protocol of polling the server for controller status update. Controllers poll a server attaching its *Base*64 encoded current status to a query. Server responds *OK* if the controller status matches the state known by the server and *NOK* (not OK) with the attached updated state if the server has any new status information. Successively, the controller polls the server again with the updated status to receive an OK that denotes that the states are synchronized. In principal, it is possible the server-side status gets updated meanwhile. The overall goal of this process is to converge in synchronized controller status string with the coordinating server side string.

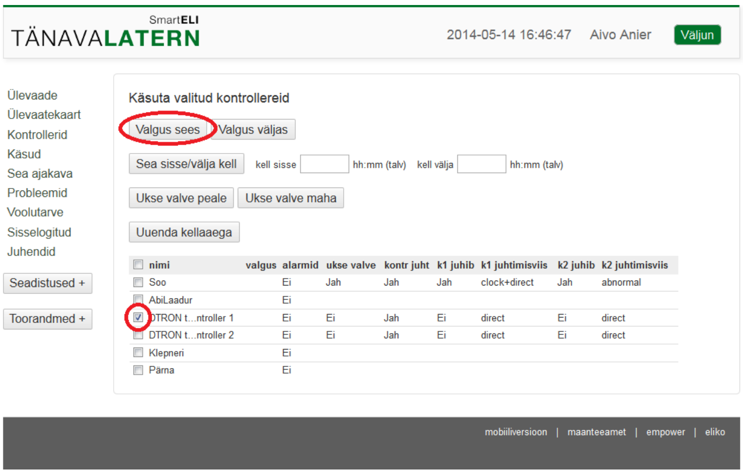
Figure 7.8 shows an example *Base*64 encoded status string.

c=10002&d=FF01602C00000100000
0000000000000000000080010000
000000000001000000000000D4FAF
FFFFFFF0000000000007C004C

Figure 7.8: Example Base64 encoded state string for controller ID 10002.

The intended controller status can be updated in the server-side when a user manipulates the server-side status using the web interface. For example, the user manually picks a controller and sets its status to turn on the streetlights. The status change is propagated to the actual controller upon next poll session by the controller. That helps avoiding too frequent switching of lamps on and off while the light conditions should change or the sensors should emit false signals too often. The optimal polling period depends on the dynamics of lighting conditions and the reaction time requirements to the light adaptation. This possibly varies from controller to controller.

Figure 7.9 shows a screenshot of the controller "command" interface.



	nimi	valgus	alarmid	ukse valve	kontr juht	k1 juhib	k1 juhtimisviis	k2 juhib	k2 juhtimisviis
☐	Soo		Ei	Jah	Jah	Jah	clock+direct	Jah	abnormal
☐	AbiLaadur		Ei						
✓	DTRON t...ntroller 1		Ei	Ei	Jah	Ei	direct	Ei	direct
☐	DTRON t...ntroller 2		Ei	Ei	Jah	Ei	direct	Ei	direct
☐	Klepneri		Ei						
☐	Pärna		Ei						

Figure 7.9: Tartu city light-controller web user interface.

Suppose a controller C_i polls the server at some arbitrary interval t_i and sends its current status encoded in the modified *BASE64 status string* s_i. Server will send corresponding to C_i response code $r_i = 1$ if there are no updates available and a modified BASE64 status string denoting the new status s_i' the controller should be switched to.

Let us look a simplified example where the system boots up first time. By system we mean all the controllers with all lights connected to them and the central server. The server boots up having lights-off status for all controllers and lights. Controllers now start reporting their status strings to the server at an interval, e.g. $t = 100ms$. Note that the local clocks in each controller are by no means synchronized. Given enough time elapses the polling sessions are carried out in uniformly randomized manner. Controllers report their status, e.g. C_i status s_i telling that lights ${L^1}_i$ to ${L^m}_i$ are off and since the same status is initially on server side, the server will respond OK.

Now if some user u would login in to the server and switches controller C_i lights' ${L^1}_i$ to ${L^m}_i$ state to new value ${L^1}_i = 1$ to${L^m}_i = 1$ (where 0 stands for light off and 1 for light on) the next time C_i reports its state s'_i it will be given a response code NOK and the corresponding new status string $s'_i \neq s_i$ denoting that L_1 and L_2 should be turned on. Controller C_i receives the new status, turns the lights on and reports it's newly arrived state s'_i back to the server, asserting the changes have come into effect. Server will acknowledge this by responding OK with no state string s attached.

7.3.3 Adapters

The test adapters are needed to interpret (abstract) events arriving from UPTA model via DTRON and mapping them to SUT input format. DTRON-Selenium extension was developed by the author to ease the development. The framework implements *dependency injection* (DI) paradigm and tooling to automatic adapter and resource discovery.

The adapter programmer first needs to decide the (Java) package to put the adapters into. The fully qualified package name is then passed to the DI container that takes this to the auto-discovery adapter loader. In order to write an adapter one needs to write a class to this package extending *AbstractCliContextAdapter* class to inherit the appropriate framework behavior. It is important to note that the framework expects the adapter classname to follow the Java naming standard and be written in camel-case format. The reason for this follows and serves as a contract for mapping the adapters.

The adapter is then picked up by the framework and its classname is mapped to the set of UPTA channel names. For example, an adapter class *LoginSomething* would be mapped to channel *i_loginSomething* in the UPTA model. The framework follows the DTRON naming convention and utilizes *i_* and *o_* prefixes in channel names.

The adapter execution starts with the *execute*(...) method where relevant Selenium code needs to put into. Handle for the browser instance can be fetched with *getWebDriver*() method following the Selenium API and *getDtron*() fetches the handle to the underlying DTRON session to allow manual synchronization invocation.

If DTRON handle is not used for manual reporting, the framework automatically invokes a *result* synchronization back to UPTA model attaching a variable *resCode* $= 0$ to denote success. If exceptions are raised from the *execute*(...) method the *resCode* would be assigned an "error code" other than 0, i.e. $resCode \neq 0$. API user can manually assign or overriding this by calling or override the *report*(*int*) method to provide a custom error code. Doing this will disable the automatic result code reporting discussed before.

UPTA channel synchronization arriving to the *execute*() method may also have attached variables. The handles for both the synchronization and the variables will be passed to the *execute*() method as an argument using the same class as underlying DTRON.

Note that *getDtron*() method handle can be used to manually interface with the underlying DTRON for implementing additional behavior to *execute*() a specific adapter. Having this behavior does not intervene with the (automatic) result code reporting system. That still gets invoked strictly after *execute*() has finished.

Figure 7.10 presents an adapter demonstrator. Note that HTML web elements required for *WebDriver* manipulation get automatically injected and are actually externalized to a *.properties* configuration file to allow runtime configuration (without re-compilation). The injection system can also be utilized to obtain handles to other adapters. For example, defining an adapter *Logout* one could declare @*Inject Login login* field to gain access to *login.postCondition*() that would serve as a pre-condition for the *Logout* adapter.

```
public class Login extends AbstractCliContextAdapter
implements PostCondition {

@Inject @Named("STARTURL") String STARTURL;
@Inject @Named("LOGIN_BUTTON")      String LOGIN_BUTTON;
@Inject @Named("LOGIN_USERNAME")   String LOGIN_USERNAME;
@Inject @Named("LOGIN_PASSWORD")   String LOGIN_PASSWORD;
@Inject @Named("LOGIN_LOGGEDIN")   String LOGIN_LOGGEDIN;

@Inject
public Login(Cli context, SeleniumDriverFactory sdf) {
  super(context, sdf);
}

@Override
protected void execute(IDtronChannelValued v) {
  super.execute(v);

  getWebDriver().get(STARTURL);

  WebElement loginButton = findInput(LOGIN_BUTTON);
  assertThat(loginButton, notNullValue());

  findElement(LOGIN_USERNAME).sendKeys( // fill
        getContext().username);
  findElement(LOGIN_PASSWORD).sendKeys(
        getContext().password);

  loginButton.submit(); // submit

  postCondition();
}

@Override
public void postCondition() {
  // assert we're seeing the login screen
  final WebElement logoutButton =
        findInput(LOGIN_LOGGEDIN);
  assertThat(logoutButton, notNullValue());
}
```

Figure 7.10: Example adapter class for performing a “login”.

7.3.4 Model

Figure 7.11 presents an illustrative UPTA model. This is the model of a single user logging in to the web UI, picking a controller to command the lights to turn on, checking for success and then logging out. This is modeled with a succession of the following synchronizations:

(i) Start out with *login*. Note that the pre-condition to check if we have the login screen at the first place is implemented by the corresponding adapter in Figure 7.10.

(ii) Navigate to the "commands" sections

(iii) non-deterministically pick and "tick" an adapter that has to be switched "on".

(iv) The model itself defines a general approach to turn lights both "on" or "off", but the additional guard restricts this on "on" only. An extra variable is attached to the channel with the chosen operation. So "on" or "off" states are encoded as *integer* values that by contract will be delivered as an argument to the adapter *execute*(...) method as described in Section 7.3.3.

(v) Navigate back to the "commands" page, looping this until the light gets turned on

(vi) close the session with *i logout*.

Following the UPTA semantics we need to match the initiating synchronization channels (with symbol "!") with corresponding receivers (denoted with symbol "?"). Figure 7.11 provides this mapping. This model control structure can be defined having only reflexive transitions with source and destination locations and the initial state, but with this case we also model SUT to be "blocking". This means one operation has to be completed with some returning result code before another can be executed.

Though the model and adapter represent a single user case performing actions upon the web UI, the same template enables to run parallel instances of the template to implement a more realistic scenario where many users perform actions on the SUT simultaneously.

Figure 7.12 shows the deployment model of this case study. We ran two users in parallel to cover scenarios where poorly implemented SUT would start interleaving between users and for example allows a situation where one user starts logging in, but is actually logged in as other user who is performing the same action simultaneously.

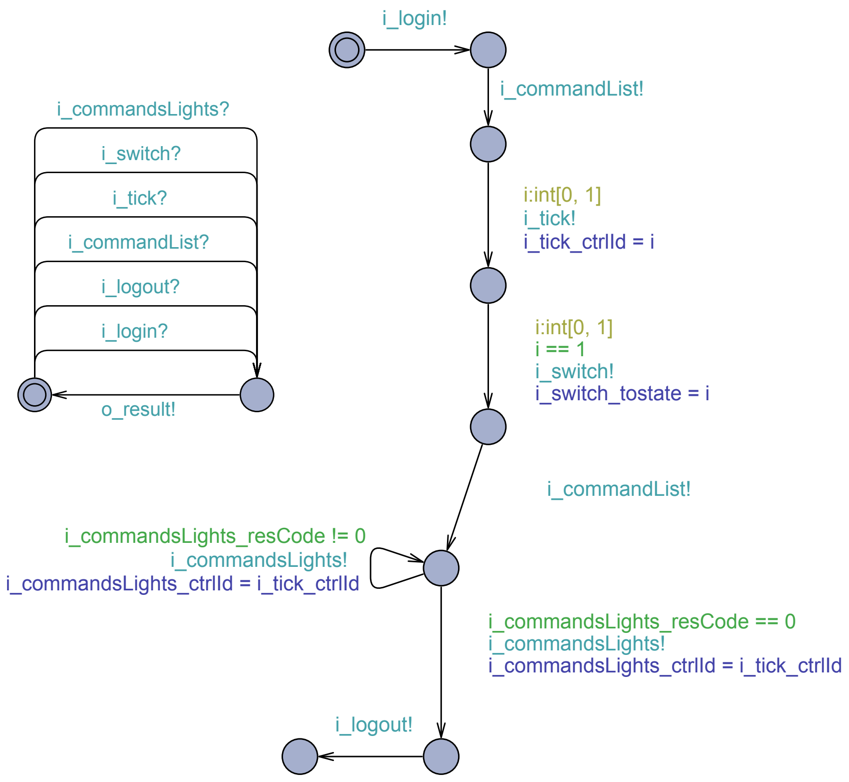


Figure 7.11: Light-controller UPTA model.

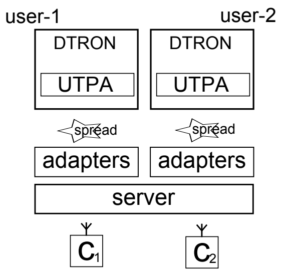


Figure 7.12: Multi-user deployment model.

The light-controllers were virtualized to allow quick and straightforward control over the whole testing configuration. Virtual controllers imitate the same synchronization protocol and are indistinguishable from the server-side.

For more extensive load testing the UPTA model depicted in Figure 7.11 was extended with an additional loop enabling to re-execute the same test case, i.e. after i_logout, starting again with i_login. Test configuration was defined to run each test-session for $100s$.

The SUT exhibited the expected behavior most of the cases, but for some cases the server showed the lights to be off, while the controllers had actually the lights on and the status was expected to be synchronized with the server. In depth diagnostics and test fail causes have not been carried out yet. An example test-run was screen-captured and is available for design analysis on DTRON website[85].

7.4 Summary

This chapter presented the results of two case studies. Firstly the Scrub Nurse Robot project that served the motivational purpose to later develop DTRON. Secondly the Tartu city lighting project that applied the later developed DTRON for distributed black-box testing.

Both case studies were considered a success as a proof of concept. The Scrub Nurse Robot mechanical design limitation were overcome by automata modelling and enabling the execution with JSNR. This enabled better collaborative development of the robot as a shared resource with less downtime.

The Tartu city lighting project had two major results. Firstly, that it was possible to apply model based testing with DTRON to black box distributed systems with moderate effort. Secondly there was a non-deterministic bug found when executing multiple copies testers in the context of different users where one or the other user was denied access to the system. The erratic occurrence of this bug couldn't be made reproducible, but it gave enough insight to the developer to fix the (session management) problem.

8 Conclusions and future work

8.1 Main results

The thesis presents a framework to automatize the model based control and testing of time critical Cyber-Physical Systems. The research focus is verifiable modelling and execution of models to enable provably correct on-line testing, model-based control and monitoring in robotic and web applications.

Firstly, the requirements for selecting an appropriate modelling formalism for model based control and testing of the target class of Cyber-Physical Systems are outlined. The formal modelling alternatives are discussed from pragmatics, taxonomy, and expressiveness point of view, and their relevance is motivated with respect to the requirements. The analysis of models for timed systems concludes that one of the most relevant formalisms in given context are Uppaal Timed Automata which have rich semantic domain and advanced toolset for modelling, testing and verification.

Secondly, provably correct model based test development workflow is introduced together with verification conditions necessary to assure the correctness of development increments. Traditional model based test development process comprises steps such as modelling the system under test, specifying the test purposes, generating the tests, deploying test components on the execution architecture and executing them against system under test. The correct-by-construction approach advocated in thesis introduces the verification steps that alternate with development steps. Since the focus of thesis is model-based online testing of critical systems with timing constraints we capitalize on the formally provable correctness criteria of the test suite for all its development steps. We also define the model transformations needed for creating well-formed models of system under test as well as of testers. The verifiable correctness criteria are specified as proof obligations suited for automatic proof by model checking.

Thirdly, the thesis addresses one of the key issues faced in model-based techniques - tool supported model construction, particularly, the problem how models for robot action control and for model based testing can be constructed by means of machine learning methods. Two contexts of model learning by automata learning technique have been studied: (i) learning the human-robot interaction for robot action control and (ii) learning interaction between the system under test and its environment to generate load patterns for load testing. For these learning cases two versions of the learning algorithm have been developed. The core algorithm implements online learning strategy, i.e. the learner does not have a possibility to back-track and ask equivalence or membership queries about the arbitrary length prefixes of the learning input trace. The learning algorithms construct relative to input trace complete non-deterministic automata networks, such that all observation sequences learned can be reproduced by that network.

The specialized version of the core algorithm rely on different assumptions on communication and synchronization mechanism applied in the model. In the first learning case the Human-Robot Interaction learning problem is studied in the context of cooperative surgical task accomplishment by Scrub Nurse Robot. In this case the interacting processes are assumed to communicate over i/o variables and synchronize by means of clock constraints only. That is because the forward stability of synchronization hypothesis cannot be guaranteed in the incremental and unsupervised learning of human interactions.

In the other learning case, the environment model for a node of IEEE1394 distributed leader election procedure is constructed. The model processes are assumed to communicate over i/o variables and synchronize by means of channels. Forward stable synchronization assumption is motivated here due to the fact that observable communication actions always incorporate both communication parties and the channels in Uppaal model represent such synchronous i/o action pairs.

Fourth main contribution of the thesis is the design and implementation of distributed model-based control and test execution framework DTRON that enables model driven execution of different robot control stacks as well as test suites of model based testing. The practical experiments have shown that DTRON can be extended also for online (safety) monitoring applications and to serve as a tool integration middleware. DTRON relies on Uppaal model checker and on-line test execution tool TRON extending these tools by enabling coordination and synchronization of distributed components and providing a consistent API based on standard Java technology.

The novel feature of DTRON is its capability of implementing the time keeping mechanism needed for Δ-criterion based remote testing. The maximum allowable delay criterion (Δ-criterion) for remote testing was recently proposed by Uppaal team. The performance evaluation experiments with DTRON were conducted to determine the latency overhead DTRON has when introducing an extra layer of messaging abstraction. The focus of measuring experiments was to study the effect of the Spread toolkit as messaging service with the combination of Google Protocol Buffers - a language-neutral, platform-neutral, extensible mechanism for serializing structured data. The measurement results allow concluding that DTRON provides performance guarantees sufficient of applying Δ-criterion also in distributed test architectures with message propagation time in millisecond rage (depending on the networking configuration).

Finally, the applicability of DTRON framework and model-based techniques developed around it have been demonstrated in two representative case-studies: Scrub Nurse Robot model based control and a distributed performance testing of the Tartu City street light control system. The Scrub Nurse Robot case study highlights the features related to handling real-time and safety constraints in robot high-level action control. The street light system case-study focuses on aspects of coordinating distributed systems and on how to generate test configurations for distributed web applications. Due to these practical results the author has a ground to believe that the ideas presented in thesis contribute also to improve the practical development processes of broader class of industrial scale time critical cyber physical systems.

8.2 Future work

The provably correct test development process model, proposed in Chapter 3 of thesis specifies a tool chain that can be used to automate the development steps. As stated in thesis this is not a fixed set of tools that can be used. Currently the test generator involved is reactive planning online black-box tester generator that synthesises a symbolic tester as UPTA for non-distributed systems. To apply similar approach for distributed testing means generating a local tester separately for each system component by counting on the fact that the local context of testers needs to be specified separately. One way of overcoming this limitation is extending the current test generation approach for online distributed testing so that the local testers are generated with locality specific coordination constraints involved already in their models.

The learning algorithms introduced in the Chapter 4 can generate potentially very complex models in terms of the size of model state space, regardless the interval abstraction and variable domain rescaling operators are implemented in the algorithm. High complexity can prevent model checking of learned models for their correctness and implementability. Though there is not general minimization theory for the network of non-deterministic timed automata the deterministic fragments of the model can still be minimized by applying bisimulation quotient method [86] and the classical Hopcroft minimization algorithm. This allows merging the non-distinguishable (from trace semantics point of view) locations and edges in UPTA templates and in the product automaton of the parallel composition of templates. Further development of optimization theory poses interesting challenge for research specially combined with contract based model development theory assumptions [87] can be exploited for incremental component-wise learning of timed models.

8.3 Concluding remarks

This thesis was based on an extensive survey on model based testing with the focus on developing DTRON tool. It is the authors position that the survey alone gives a solid starting point to base any further development of DTRON or similar tools and is a result on its own.

DTRON was re-implemented 4 times to reach its generality and maturity it now has. The author is confident that the resulting framework is correctly and effectively (in terms of memory and processor resource usage) implemented and easily reused for its purpose.

Good usability of DTRON is also supported by the fact that several student and research projects have been based on the usage DTRON. "Google Analytics" report on the supporting website stating around 4000 visiting users (200 revisiting) last year.

DTRON desperately needs good tool support for the runtime visualization/-diagnostics and the internal messaging protocol needs to be supplemented with proper support for NTP based timing corrections (Δ-criterion).

Some development has been carried out addressing these problems, but have not been published within DTRON and remain outside the scope of this thesis.

Abstract

This thesis presents results on developing DTRON framework to automatize the model based control and testing of time critical applications. DTRON relies on Uppaal model checking tool and on-line test execution tool TRON extending these tools by enabling coordination and synchronization of distributed components and providing a consistent API based on standard Java technology.

The research is based on an extensive survey of model based testing with focus on verifiable modelling, and execution of models to enable provably correct on-line testing, model-based control and monitoring in robotic applications.

This work was originally motivated by the Scrub Nurse Robot project where human adaptive control scenarios and on-line safety monitoring were main design concerns. The author implemented the JSNR framework for the robot that was later developed to be DTRON.

The same design principles and technical solutions appeared to be relevant also for remote and distributed testing of web-based applications as demonstrated in a street light control software testing case-study. The author refactored DTRON to be more generic and support distributed execution of model based tests in addition to robot control.

The novel ideas of DTRON framework design revealed also extension opportunities for its application in broader class of Cyber-Physical Systems (CPS) capitalizing on dynamic reconfiguration, self-calibration depending on the dynamic delay estimates of the deployment configuration, and on-the-fly feasibility checks of models used in control and testing applications.

DTRON was re-implemented 4 times to reach its generality and maturity it now has. The author is confident that the resulting framework is effectively implemented and easily reused.

The author believes that the ideas presented in thesis contribute also to improve the practical development processes of industrial scale time critical CPSs.

Kokkuvõte

Küberfüüsikalised süsteemid (KFS) pakuvad suuri võimalusi, kuid ka suuri väljakutseid mitmetes valdkondades nagu näiteks elektroonikatööstus, transpordisüsteemid ja tööstuse automatiseerimine. Kõrge keerukusega KFS tarkvara disaini korrektsuse tagamine nõuab uusi arendusmetoodikaid ja vahendeid, mis peavad olema suunatud laiale arhitektuurilahenduste spektrile. Samuti peavad KFS arendusvahendid lahendama olulise paralleelsuse ja ajastamiskitsendustega seotud probleeme.

Käesolev artikkel käsitleb mudelipõhise testimise vahendit DTRON, mis on välja töötatud ajatundlike hajusarhitektuuriga süsteemide testimiseks. DTRON on loodud mudelkontrolli vahendi Uppaal ja online testimisvahendi TRON baasil laiendades nende funktsionaalsust online hajustestimiseks vajalike koordineerimis- ja sünkroniseerimisfunktsioonidega.

Et tagada hajustestide juhitavus, on DTRONi projekteerimisel lähtutud Δ-testitavuse nõudest. Artiklis esitatakse DTRONi arhitektuurilahendus ning analüüsitakse selle jõudlusnäitajaid võttes arvesse võrguühenduse ning testiadapteritest tingitud hilistumisi. Jõudluseksperimentide abil näidatakse, et implementeerimiseks kasutatud vahevara SPREAD sõnumite järjestamisteenus ja võrgu ajakorraldusprotokoll Network Time Protocol võimaldavad kahandada hajustestide juhitavuse tagamiseks vajaliku parameetri Δ alla 1 ms piiri. See näitaja on piisav paljude võrkarhitektuuriga küberfüüsikaliste süsteemide hajustestimiseks.

DTRONi rakendatavust valideerivad kolm rakendusnäidet: tänavavalgustussüsteemi kontrollerite võrgustiku, pankade-vahelise kauplemissüsteemi ja mobiilse roboti navigatsioonisüsteemi testimine.

Acknowledgements

This work was partially supported by:

(i) Nations Support Program for the ICT in Higher Education "Tiger University",

(ii) Estonian Science Foundation grant #7667,

(iii) This research is partially supported by ELIKO and the European Union through the European Regional Development Fund.

Euroopa Liit
Euroopa
Regionaalarengu Fond

Eesti tuleviku heaks

Curriculum vitae

Personal data

Name:	Aivo ANIER
Date of birth:	14.apr.1983
Place of birth:	ESTONIA
Citizenship:	ESTONIA

Contact data

Address:	Akadeemia tee 15a, 12618 Tallinn
Phone:	+372 620 2325
E-mail:	aivo.anier@ttu.ee

Education

2004 – . . .	Tallinn University of Technology, Ph.D. studies
2001 – 2004	Tallinn University of Technology, B.Sc., M.Sc.
1989 – 2001	Highschool

Professional employment

2014 - . . .	Girf OÜ, software architect
2010 - 2014	Tallinn University of Technology, lecturer
2009 - 2013	ELIKO, researcher
2010 - 2010	Tokyo Denki University, researcher
2007 - 2010	Tallinn University of Technology, researcher
2005 - 2007	ELIKO, researcher
2004 - 2005	Department of Defence, security advisor
2002 - 2004	Girf OÜ, system administrator

Elulookirjeldus

Isikuandmed

Nimi:	Aivo ANIER
Sünniaeg:	14.apr.1983
Sünnikoht:	ESTONIA
Kodakondsus:	ESTONIA

Kontaktandmed

Address:	Akadeemia tee 15a, 12618 Tallinn
Telefon:	+372 620 2325
E-post:	aivo.anier@ttu.ee

Hariduskäik

2004 – . . .	Tallinna Tehnikaülikool, Doktoriõpe
2001 – 2004	Tallinna Tehnikaülikool, B.Sc., M.Sc.
1989 – 2001	Keskkool

Teenistuskäik

2014 - . . .	Girf OÜ, tarkvara arhitekt
2010 - 2014	Tallinna Tehnikaülikool, lektor
2009 - 2013	ELIKO, teadur
2007 - 2010	Tallinna Tehnikaülikool, teadur
2010 - 2010	Tokyo Denki Ülikool, researcher
2005 - 2007	ELIKO, teadur
2004 - 2005	Kaitseministeerium, teabeturbe nõunik
2002 - 2004	Girf OÜ, süsteemiadministraator

Appendix I: Timed automata based provably correct robot control

Timed automata based provably correct robot control

A.Anier[1], J.Vain[1]
[1]Department of Computer Science, TUT, Ehitajate tee 5, 19086 Tallinn, Estonia,
E-mail: {Aivo.Anier@ttu.ee, vain@ioc.ee}

***Abstract*—This paper presents a feasibility study on the usage of Uppaal Timed Automata (UPTA) for deliberative level robotic control. The study is based on the Scrub Nurse Robot case-study. Our experience confirms that UPTA model based control enables the control loop to be defined and maintained during the robot operation autonomously with minimum human intervention. Specifically, in our robot architecture the control model is constructed automatically using unsupervised learning. Correctness of the model is verified on-the-fly against safety, reachability, and performance requirements. Finally, it is demonstrated that UPTA model based robot control, action planning and model updates have natural implementation based on existing model execution and conformance testing tool Uppaal Tron.**

I. Introduction

Scrub Nurse Robot (SNR) project is a joint research of Miyawaki Lab at Tokyo Denki University and the Department of Computer Science at Tallinn University of Technology. The goal of the project is to develop a human-adaptive Scrub Nurse Robot (SNR) that can adapt to surgeons with various levels of skill and experience in order to replace the human scrub nurse and compensate for the severe shortage of scrub nurses [1].

Surgery sets very high safety, reliability and performance requirements not only to surgical robots but also to SNR robots that assist the surgeon, e.g. by handling over and receiving instruments during surgery. The requirements to SNR can be met by means of formal methods and provably correct design only. In addition, the need for adaptability to surgeon's behaviour and environment changes induces the need for robot learning during its interaction with humans. Since SNR must ensure provably correct behaviour during surgical procedures, the learning quality is of critical importance. For characterizing the quality of behaviour learning we propose the real-time input-output conformance relation (RTIOCO) [2]. The SNR behaviour is monitored by Uppaal TRON tool [3] throughout the surgical procedure. While the RTIOCO relation between the already learned model and currently observed timed traces gets violated, TRON signals about that and the learning set will be extended with RTIOCO counter example traces and new learning (model construction) iteration will be triggered online (see figure 1).

The result of each learning iteration will be verified against predefined safety constraints, performance and non-blocking conditions using Uppaal model checker [4]. In case the newly learned model extension is correct the SNR behaviour planning is switched over to the extended model. In case the correctness conditions are violated, the SNR discards the model extensions, recovery procedure is initiated and diagnostic traces passed to human expert for offline revision.

Second major issue of model based control in SNR is action planning in the presence of partial knowledge about the external situation and potentially non-deterministic (although with low probability) reactions of human partners in the surgical scene. To address these issues we have implemented in SNR reactive planning algorithm developed initially for online testing of non-deterministic embedded systems [5]. Encoding of online planning preferences in the robot control model allows natural implementation of UPTA based action planning in Uppaal Tron.

The rest of the paper is structured as follows. In Section II, project background and motivation is presented; Section III introduces UPTA syntax and semantics and discusses the usage of Uppaal Tron for SNR robot control. Section IV outlines model construction related issues and Section V model correctness verification issues.

II. Timed automata and Uppaal

Timed Automata have been extended in Uppaal tool family [4] (www.uppaal.com) by data types parallel composition and synchronization primitives. Due to the extended syntax, and semantics these automata are named Uppaal Timed Automata (UPTA). The Uppaal tools provide a rather convenient GUI for UPTA model definition and a verification engine. Modelling robot control and also planning is an overwhelming task to be done by hand. It grows large and complex fast. Even if the Uppaal tool has greatly improved this area it still remains an issue - an issue that can be addressed with UPTA synthesis using external tools and languages [5]. High-level model synthesis has proven to be effective and justified but doing the same for low-level robot actions mapped onto input-output (I/O) behaviour is on the other hand redundant. Low-level I/O operations are, e.g., actual manipulator control, reading sensor data etc. Defining these operations and related control actions in terms of abstract models needs further interfacing with the actual hardware. The Uppaal main tool does not support interfacing models with real world phenomena. But there are many sub- and side projects and one of them is Uppaal for Testing Real-time systems Online – the TRON tool. Uppaal TRON is a testing tool, based on Uppaal engine, suited for black-box conformance testing of timed systems, mainly targeted for embedded software commonly found in various

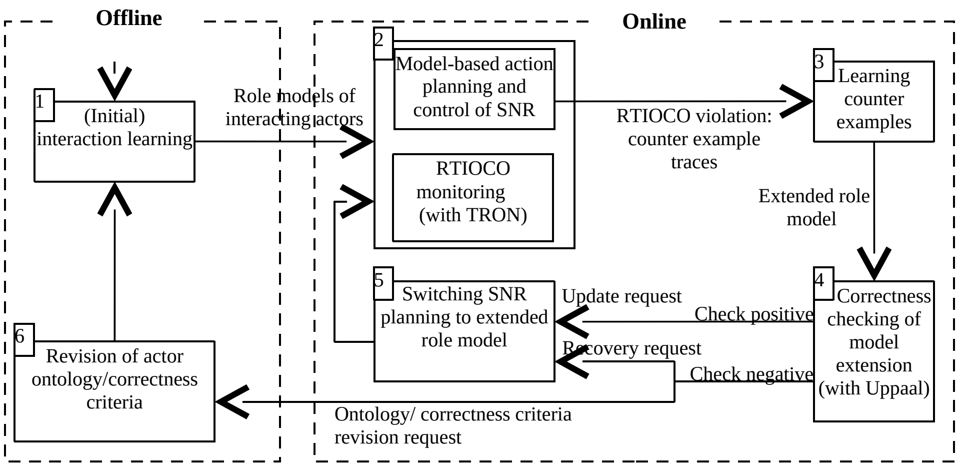

Offline
Online
1 (Initial) interaction learning
Role models of interacting actors
2 Model-based action planning and control of SNR
RTIOCO monitoring (with TRON)
RTIOCO violation: counter example traces
3 Learning counter examples
Extended role model
4 Correctness checking of model extension (with Uppaal)
Update request
Check positive
Recovery request
Check negative
5 Switching SNR planning to extended role model
6 Revision of actor ontology/correctness criteria
Ontology/ correctness criteria revision request

Figure 1. Incremental learning cycle for SNR behavior planning

controllers [3]. Although the TRON is not targeted for model based control, it provides efficient RT-Conformance checking mechanisms between model and real physical process and can issue control commands in the same way as it generates stimuli to the system under test. Thus, TRON could be adapted for robot control with a little effort.

III. Uppaal TRON and TA modelling

TRON follows a client-server programming model. It takes an Uppaal model file as an input and by definition uses this to conformance test the implementation under test (IUT). That IUT in our context is the robot. TRON comes with Adapter classes that are intended for use on the IUT side. These are there for the developers' convenience to "talk" the same protocol as TRON. The Adapters serve the purpose of "bridging" the UPTA with the underlying hardware. In the context of conformance testing the modelling means both formal representation of the IUT and its environment. In figure 2 the SNR tool changer is modelled as the environment and figure 3 shows the controller of the tool changer as IUT. For synchronizing the execution of models with reality synchronization signal called Channels are used in Uppaal. Channels are specified as transition labels in UPTA, e.g., in figure 3 there are channels "toolchanger" and "toolchangerFinished". Channels have a calling side and a receiving side denoted by "!" and "?" respectively. These together form synchronization which by definition makes the bounded transitions to be taken simultaneously while not allowing time to pass.

By means of channels TRON is able to intercept these synchronizations and initiate function calls on the IUT side, i.e., in our example controlling the manipulator. If TRON executes the Toolchanger controller model and initiates a synchronization "toolchanger!" then the corresponding labelled transition in the tool changer model also has to be taken. But TRON intercepts this synchronization and executes also a function call on robot API and physical movement is achieved. The same

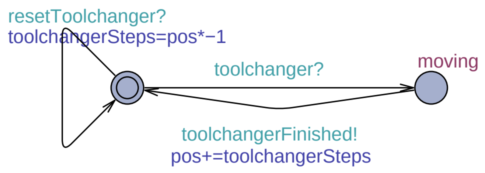


Figure 2. Toolchanger model

synchronization mechanism works in reverse direction. The model execution waits in a state where "toolchangerFinsihed!" has to be called. This will be executed by IUT whenever the movement finishes and TRON intermediates this back to the model enabling it to proceed. While having the ability to complement transitions with additional variable labels this enables sensory or feedback information to be imported into the model. There is one consideration though. Variable labels can only be integers.

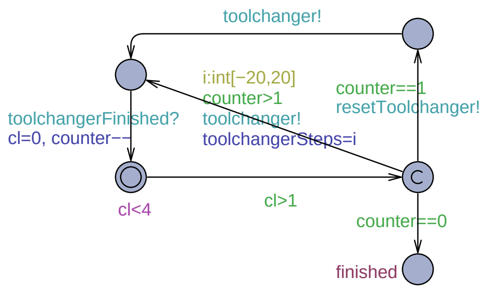


Figure 3. Toolchanger controller

Channels in Uppaal are defined like any other variable with the limitation that they have to be declared in global declarations section. Uppaal models are composed of UPTA

Templates that are instantiated into Processes. Local variables are scoped into one Template/Process and Global variables are shared across them. Since Channels are a mechanism to bind two (or more) transitions in the corresponding Templates they have to be in Global scope. Variable declarations while using TRON on the other hand need special attention. Variable labels used by TRON have to be declared globally. And it is important to understand that every time intercepted synchronization takes place TRON checks the values of all Global variables registered for. (On the start of every TRON session you register for variables you want to read/write). If there is a difference between the variable states of the TRON and IUT side – session will fail with non-conformed result. Figure 4 shows a component model of how the architecture fits together. Note that TRON distribution also provides Adapters written in C.

IV. Model synthesis

Model synthesis is an essential part of SNR adaptability to changing environment. Using the tools discussed makes this rather convenient. The control model is constructed in two phases: at first the model skeleton is created by hand and verified with TCTL (Timed Computation Tree Logic, [6][7][8]) queries. Then - iteratively evolving while checking if the TCTL queried properties remain satisfied. If this succeeds the iteration steps can be programmatically atomized. After the model has been synthesized and before deployed with TRON the model is verified for the last time.

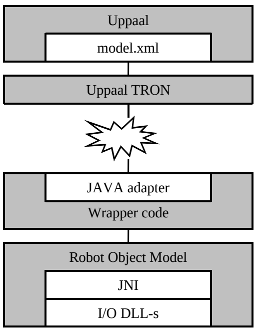


Figure 4. Architecture overview

V. Verification

Uppaal tool comes with a verification engine and therefore enables proving of the correctness of controller model. The query language of Uppaal, used to specify properties to be checked, is a subset of TCTL. For example one could query the model if there is a way to deadlock: $\exists\Diamond deadlock$. Consider the model of a robot shoulder joint shown in figure 5. It is rather simple but denotes a critical part of the system. The shoulder height h is bounded to $0 \leq h \leq 50$. Given the robot the join will otherwise break.

The same constraints as in modelling the controller and the environment (figure 2 and figure 4) apply here. In figure 6 only the model of the joint is depicted not the controller. Controller model can be generated automatically from the model of the joint. Furthermore, it could be modified on-the-fly while the joint model changes. The important part is that after the integration of both models we could query the model weather the invariant will always hold: $\forall\Box pos \geq 0 \wedge pos \leq 50$. Although it is rather obvious given the example at hand it may grow beyond that easy comprehension fast.

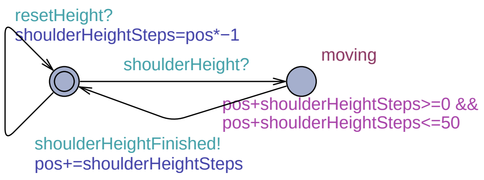

resetHeight?
shoulderHeightSteps=pos*−1
shoulderHeight?
moving
pos+shoulderHeightSteps>=0 &&
pos+shoulderHeightSteps<=50
shoulderHeightFinished!
pos+=shoulderHeightSteps

Figure 5. Shoulder joint height invariant

VI. Time and real-time

Uppaal has the notion of clocks and time. Therefore the name “timed automata”. This means clock constraints can be assign to transitions as guards and to locations as invariants. Note that UPTA have special type of locations called “urgent” and “committed” which both denote a situation where time (clocks) is not allowed to pass. Dealing with clocks in TRON introduces nontrivial semantics. Consider the model depicted in figure 6. If $n = 0$ then this means the interval is atomic and the model has to advance exactly at $cl = 10$. Since the underlying operating system TRON is running on is (usually) not a real-time operating system (OS) – this is not guaranteed to happen. Although TRON handles this internally to some extent by extending this interval TRON execution will still terminate resulting with IUT not conforming to model denoting clock constraint violation.

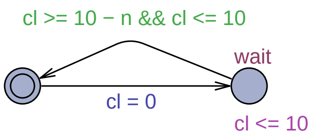


Figure 6. Interval clock constraint

There are a couple of ways to overcome this. The most elegant way is to rewrite the underlying TRON algorithm to better fit real life use, but this option is available to TRON developers only. An alternative way is consider this while modelling and using sensible time intervals instead of time points, e.g., in figure 6 the variable n is incremented to the appropriate value. Third possibility is tuning the parameter Timeunit. Uppaal clocks don’t have default real-world interpretation. In order to create the relation between virtual clock time and real time one can define clock unit to be some specific value in nanoseconds. This is done in configuration upon TRON start-up. Using considerably large time intervals instead of time points and appropriate small timeunits lead to a solution where the interpretation of time remain the same but conformance test will not fail.

VII. Conclusions

This paper presented an approach to implement provably correct robot control using Uppaal timed automata and the TRON tool. We briefly presented the framework project where this is deployed and key architectural components that take advantage of it. The robot project was designed to heavily depend on timed automata and the approach at hand has proven to be sufficient to support this demand. During the deployment we experienced some shortfalls of Uppaal and Tron when it came to actual implementation. These were presented with resolutions used to address them. This included TA modelling guidelines and Tron configuration considerations. Future work includes the integration of TA based robotic control with various other architectural components of the SNR project such as reactive planning [5] and hybrid learning methods.

References

[1] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, and J. Vain. Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery. *Industrial Electronics, IEEE Transactions on*, 52(5):1227 – 1235, October 2005.

[2] Moez Krichen and Stavros Tripakis. Interesting properties of the Real-Time conformance relation. In Kamel Barkaoui, Ana Cavalcanti, and Antonio Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2006.

[3] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, page 77–117. 2008.

[4] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[5] J. Vain, F. Miyawaki, S. Nomm, T. Totskaya, and A. Anier. Human-robot interaction learning using timed automata. In *ICCAS-SICE, 2009*, pages 2037 –2042, August 2009.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, LNCS, page 200–236. Springer–Verlag, September 2004.

[7] 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*, pages 414 –425, June 1990.

[8] T. A Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 394 –406, June 1992.

Appendix II: Human-Robot Interaction Learning Using Timed Automata

ICROS-SICE International Joint Conference 2009
August 18-21, 2009, Fukuoka International Congress Center, Japan

Human-Robot Interaction Learning Using Timed Automata

J. Vain[1], F. Miyawaki[2], S. Nõmm[3], T. Totskaya[1] and A.Anier[1]

[1] Department of Computer Science, Tallinn University of Technology, Tallinn, Estonia
(Tel : +37-2-620-4190; E-mail: vain@ioc.ee)
[2] Miyawaki Lab, Tokyo Denki University, Tokyo, Japan
(E-mail: miyawaki@b.dendai.ac.jp)
[3] Department of Control Systems, Institute of Cybernetics, Tallinn, Estonia

Abstract: A new unsupervised learning algorithm of human-robot interaction for behavior planning unit of a scrub nurse robot is proposed in this paper. The algorithm constructs a composition of timed IO automata where each automaton represents behavior of an interaction party. The learning architecture of the scrub nurse robot and its incremental learning cycle are discussed. The automatic compilation of the interaction model is guided parametrically in the learning process that allows generating models of different level of abstraction and profile. The approach is illustrated with an example of learning a fragment of laparoscopic surgical procedure.

Keywords: unsupervised behavior learning, model construction, timed IO automata, real-time input-output conformance relation.

1. INTRODUCTION

Studies on human–robot interaction (HRI) are roughly classified into two categories. The first category is related to communication with verbal or nonverbal aids, and the second is related to physical task accomplishment by cooperation, e.g. for spaceship inspection and repairing [1]. In this paper, the HRI learning problems are studied in the context of cooperative surgical task accomplishment by Scrub Nurse Robot (SNR) [2] and a human surgeon. The main challenge in SNR control and its adaptation to human surgeon is learning proper reactions of human scrub nurse assisting in surgical procedures. Therefore, imitational learning seems to be more relevant approach in given context than "classical" optimal planning of manipulator trajectory. Although possibly more efficient (in terms of time, energy consumption etc.) the synthetic optimized behavior of the SNR may feel unnatural and distract the surgeon's attention during critical phases of surgery.

In our approach SNR is supposed to learn the basic skills initially by observing the surgical procedure passively and later, when involved in real surgery, it improves its world model incrementally.

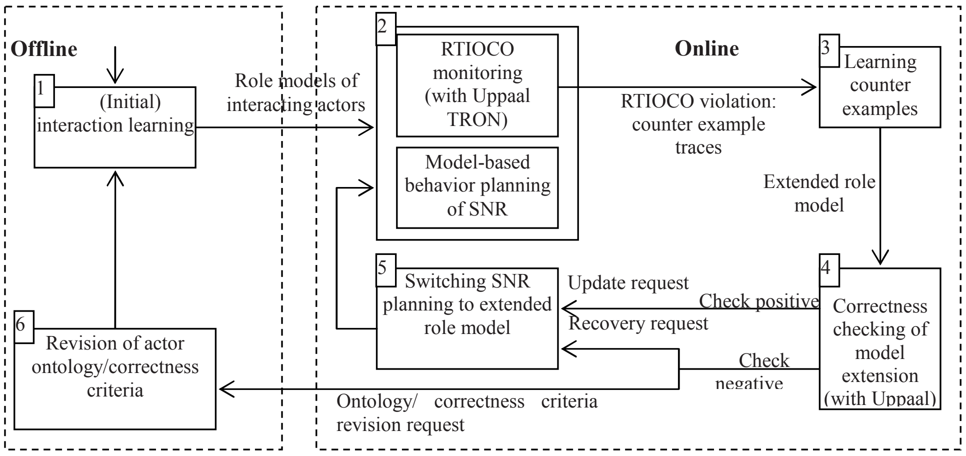
The SNR learning architecture highlighted in this paper is layered into low-level gesture learning and high-level behavior learning as defined in [3]. The stratified leaning architecture provides a flexible infrastructure to combine advantages of short-term gesture learning techniques [4, 5, 6] with a long-term behavior learning method being the focus of this paper.

In the supervised-unsupervised learning scale the SNR architecture implements the hybrid learning method. Low-level supervised learning is applied off-line for recognition of a reference set of gesture patterns. The high-level unsupervised behavior learning is applied in off-line mode when constructing the initial model of the participants' interaction in the surgical procedure, and later, on-line by updating the model incrementally with new behaviors.

The actors participating in the interactions to be learned are "black boxes" in the sense that the learning system does not have a reference on their inner motives or reasoning. Thus, we can rely on interaction related observable IO behavior only. The automata class we use for model learning from IO behavior is non-deterministic timed automata with inputs and outputs − TAIO [7]. More precisely, since we limit ourselves with observation sequences of motion switching events, being produced as symbolic outputs of the SNR low-level gesture recognition module, the class of automata constructed by the algorithm is event recording non-blocking and input complete timed IO automata [8].

Since SNR must ensure provably correct behavior during surgical procedures, the learning quality is of critical importance. For characterizing the quality of behavior learning we propose the real-time input-output conformance relation (*RTIOCO* [9]). The HRI behavior is monitored by Uppaal TRON tool [10] throughout the surgical procedure. While the *RTIOCO* relation between the already learned model and currently observed timed traces gets violated, the learning set will be extended with *RTIOCO* counter example traces and new learning (model construction) iteration will be triggered online (see Fig.1).

The result of each learning iteration will be verified against a preset safety constraints and non-blocking conditions using Uppaal model checker [10]. In case the newly learned model extension is correct the SNR behavior planning is switched over to the extended model. In case the correctness conditions are violated, the SNR discards the model extensions, recovery procedure is initiated and diagnostic traces passed to human expert for offline revision. Full learning cycle is depicted in Fig. 1.



Offline
1
(Initial) interaction learning
Role models of interacting actors
Online
2
RTIOCO monitoring (with Uppaal TRON)
Model-based behavior planning of SNR
RTIOCO violation: counter example traces
3
Learning counter examples
Extended role model
4
Correctness checking of model extension (with Uppaal)
Check positive
Update request
Recovery request
Check negative
5
Switching SNR planning to extended role model
6
Revision of actor ontology/correctness criteria
Ontology/ correctness criteria revision request

Fig. 1 Incremental learning cycle for SNR behavior planning

The key features of the TAIO automata learning algorithm proposed in the paper (the algorithm is applied in steps 1 and 3 of Fig.1) are following:

- The learning is un-supervised. There is no explicit selection mechanism involved that would put the preference of some learned behavior over others.
- We assume that the symbolic event sequences being a learning set for our algorithm are representative, i.e., they correspond to real motion switching samples recognized by low-level gesture recognition module. Filtering and labeling of video data is done in motion recognition module and highlighted in [2, 4, 5, 6].

The model constructed by learning is abstract in the sense that instead of recording each individual motion switching as a separate model element (state or transition) we define the equivalence classes of model elements that provides an abstract and more compact representation of the observed IO behavior.

The equivalence classes are parameterized using equivalence class size. Class size determines the intervals of parameter values considered to be equivalent. Our current experiments indicate that the models constructed by learning laparoscopic surgical procedures are tractable and relevant to SNR action planning and verification. The approach will be illustrated with motion samples of laparoscopic surgery in Section 4.

2. TIMED AUTOMATA LEARNING: THE STATE OF THE ART

The construction of models from observations of system behavior can be seen as a learning problem. For finite-state reactive systems, the active learning means usually constructing a (deterministic) finite automaton from the answers to a finite set of *membership queries*, each of which asks whether a certain sequence of input symbols (observed events) is accepted by the automaton or not. There are several techniques (see, e.g., [11, 12] for overview) which use the same basic principles; they differ in how membership queries may be chosen and in how an automaton is constructed from the answers. The techniques guarantee that a correct automaton will be constructed if sufficient information is obtained. In order to check the sufficiency of learning sets, the *equivalence queries* are used [11] that ask whether a hypothesized automaton accepts the correct sequences of symbols. Such a query is answered either by *yes* or by a counterexample on which the hypothesis and the correct language disagree.

In [8] the learning algorithm of Angluin [11] is extended to the setting of timed systems, namely, to event recording timed automata. This automata class is restricted to be event-deterministic in the sense that each state has at most one outgoing transition per action (i.e., the automaton obtained by removing the clock constraints are deterministic). Under this restriction, timing constraints for the occurrence of an action depend only on the past sequence of actions, and not on their relative timing.

As an alternative to the active learning method of [8] we present a passive learning algorithm that does not have a possibility to ask equivalence or membership queries. Instead, relative (to observation history) completeness of the non-deterministic TAIO [7] learned is guaranteed by construction. Secondly, instead of a single automaton our learning algorithm constructs the composition of interacting TAIO automata that communicate over IO variables and synchronize by means of clock guards. Because of the incrementality of our learning approach we can not assume the completeness of past observations and therefore we cannot conclude the synchrony of transitions via synchronization channels as used in Uppaal automata [9]. Any observation in the future can violate the synchrony between component automata regardless the fact that synchrony would have been observed up to the given moment of time.

Third extension introduced in our algorithm is the usage of predicate abstraction in clock and state variable conditions. Nondeterministic constraints in the TAIO transition guards and location invariants are constructed in the form of linear interval constraints. The possibility to determine the interval size parametrically in the algorithm keeps the balance between the model complexity and precision.

3. UNSUPERVISED LEARNING OF TIMED IO AUTOMATA

3.1 Timed IO Automata

A timed automaton [7] is a state machine whose states are divided into *variables*, and that has a set of discrete actions, some of which may be internal and some external. The state of a timed automaton may change in two ways: by *discrete transitions*, which change the state atomically, and by trajectories, which describe the evolution of the state over intervals of time. The discrete transitions are labeled with actions; this will allow us to synchronize the transitions of different timed automata when we compose them in parallel. The evolution described by a trajectory may be described by continuous or discontinuous functions.

Definition (*timed automaton - TA*) $A = (X; Q; \Theta; E; H; D; T)$ consists of: a set X of *internal variables*; a set $Q \subseteq val(X)$ of (*control*) *states* or *locations*; a nonempty set $\Theta \subseteq Q$ of *start states* (*locations*); a set E of *external actions* and a set H of *internal actions* disjoint from each other; a set $D \subseteq Q \times A \times Q$ of *discrete transitions*, where $A \triangleq E \cup H$ (we use $x \rightarrow_a x'$ as shorthand for $(x, a, x') \in D$. We say that a is *enabled* in x if $x \rightarrow_a x'$ for some x'. We say that a set $A_1 \subseteq A$ of actions is enabled in a state x if some action in A_1 is enabled in x); a set T of trajectories. Given a trajectory $\tau \in T$ we denote the first state of τ by $\tau.fstate$, and, if τ is closed, we denote the last state of τ by $\tau.lstate$. When $\tau.fstate = x$ and $\tau.lstate = x'$, we write $x \rightarrow_\tau x'$. We require that the following axioms hold:

T0 (*Existence of point trajectories*):

$x \in Q \Rightarrow \tau(x) \in T$.

T1 (*Prefix closure*):

$\forall \tau, \tau'.\ \tau \in T$ and $\tau' \leq \tau \Rightarrow \tau' \in T$.

T2 (*Suffix closure*):

$\forall \tau \in T$ and $\forall t \in dom(T)$, $\tau \trianglerighteq t \in T$.

T3 (*Concatenation closure*):

Let $\tau_0 \tau_1 \tau_2 \ldots$ be a sequence of trajectories in T such that, for each nonfinal index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$, then $\tau_0; \tau_1; \tau_2; \ldots \in T$.

Definition (*Timed I/O Automaton*)

A *timed I/O automaton* (*TIOA*) A is a tuple (B, I, O) where $B = (X; Q; \Theta; E; H; D; T)$ is a TA, I and O partition E into *input* and *output actions*, respectively. Actions in $L \triangleq H \cup O$ are called *locally controlled*; as before we write $A \triangleq E \cup H$.

The following additional axioms are satisfied:

E1 (*Input action enabling*):

$\forall x \in Q, \forall a \in I, \exists x' \in Q$ s.t. $x \rightarrow_a x'$.

E2 (*Time-passage enabling*):

$\forall x \in Q, \exists \tau \in T$ s. t. $\tau.fstate = x$ and either

1. $\tau.ltime = \infty$, or
2. τ is closed and some $l \in L$ is enabled in $\tau.lstate$.

Input action enabling is the input enabling condition of ordinary I/O automata [7]; it says that a TIOA is able to perform an input action at any time. The time-passage enabling condition says that a TIOA either allows time to advance forever, or it allows time to advance for a while, up to a point where it is prepared to react with some locally controlled action. The condition ensures that whenever time progress stops there exists at least one enabled transition.

3.1 The timed automata learning algorithm

In a nutshell, our model learning algorithm consists of a generation of TAIO locations and transitions that represent single motions and switchings between those motions respectively. Full specification of the algorithm explained in this section is presented in Appendix A.

As pointed above, the TAIO constructed as a result of learning is abstract, i.e., fine deviations of motion switching parameter values are abstracted away using interval representation.

To define the equivalence relation of motions we introduce the robustness parameter (that defines the granularity of the state space) R_i for each i-th state dimension and one for time. R_i defines the maximum distance any two points in the equivalence class can differ in ith dimension.

For convenience of defining the learning algorithm we partition the transition relation D by locations of TAIO into n subsets (n – number of locations in the model), i.e., $D = \cap_{i \in [1,n]} D_i$, where D_i corresponds to the location l_i all the transitions of D_i are departing from. By definition $\forall\ i,j \leq n,\ i \neq j \wedge D_i \cup D_j = \varnothing$.

Note that D_i are multisets, because there may be many transitions between locations l_i and l_k but we require that all the transitions are distinguishable by their guard conditions and to avoid non-determinism the interpretation sets of their guards do not intersect. To distinguish individual transitions between a pair of locations $\langle l_i, l_j \rangle$ we introduce an additional index k as third parameter in transition notation - $t(l_i, l_j, k)$.

At first, we define the inputs, outputs and parameters of the algorithm.

Inputs: The sequence E of observed motion switching events. An example fragment of E is depicted in Table 1. Each element $e_i \in E$ (a line in the Table 1) is described as a triple $e_i = \langle id_i, ts_i, \boldsymbol{X}_i \rangle$, where id_i identifies the motion phase of an actor starting with i-th event, ts_i is the timestamp of i-th switching event and $\boldsymbol{X}_i$ is the valuation of observable state variables at time instant ts_i. Only those variables being relevant to the actors' interaction model are presented in $\boldsymbol{X}_i$. The relevance of $\boldsymbol{X}_i$ is determined by actors' IO configuration model, e.g.

in Fig. 2 both actors have two inputs and two outputs. E is implemented as FIFO buffer where *get*-operation returns the oldest unread element of E. The emptiness of E is checked without shifting the pointer of E.

Output: The model of observed behaviour defined as TAIO M = < B, *I*, *O*>. An example of the automaton learned from observations of Table 1 is represented in Fig. 3 (here Uppaal graphical syntax of TAIO is used).

Parameters: To reduce the model state space and to select only the state variables having importance from the SNR control point of view we use the vector P_r that lists the observable states of importance, i.e., P_r defines the subset X_P of state variables X the current valuation $\mathbf{X}_i$ is projected on, e.g., for getting valuation of controllable variables X_c we define $P_r = X_c$.

The observation robustness R allows defining equivalence classes used as atomic propositions in TAIO conjunctive transition guards.

Parameter *RS* denotes the rescaling vector that consists of scaling functions, one for each state variable. Rescaling is necessary for keeping the model in a compact non-negative integer domain.

The algorithm comprises following basic steps:

<u>*Step 1*</u>: Unless the buffer of event sequence E is not empty read the motion switching event from buffer E and interpret it as a transition with known source location and possibly new target location. The source location is supposed to be known from the previous event read from E or it is an initial location l_0 when the first event is taken. If the buffer E is empty go to Step 3.

Table 1 Sample of $\mathcal{E}$ (motion switching events)

Event		***TS***	***Surgeon***				***Nurse***			
Sur	***Nur***		X_s	$\tilde{X}_s$	Y_s	$\tilde{Y}_s$	X_n	$\tilde{X}_n$	Y_n	$\tilde{Y}_n$
a^S_0	a^N_0	1	123	22	52	0	214	23	64	11
-	a^N_1	17	-	-	76	-	237	26	34	-
-	a^N_2	42	-	-	93	-	222	24	85	-
-	a^N_3	48	-	-	57	-	191	21	55	-
a^S_1	a^N_4	70	81	8	123	24	212	23	46	11
-	a^N_6	78	-	-	132	-	245	27	72	-
a^S_2	-	79	118	20	85	6	-	-	26	23
a^S_3	-	86	116	19	73	-	-	-	85	-
-	a^N_0	88	-	-	73	-	202	22	66	-
a^S_5	-	107	121	21	59	-	-	-	44	-
-	a^N_7	109	-	-	77	-	244	27	88	-
-	a^N_5	122	-	-	86	-	259	29	35	-
a^S_6	a^N_8	124	59	0	116	22	199	22	63	18
a^S_4	a^N_9	130	92	11	139	30	211	23	93	30
-	a^N_{10}	134	-	-	75	-	194	21	55	-
-	a^N_2	137	-	-	104	-	201	22	33	-
a^S_1	a^N_4	142	92	11	110	20	201	22	26	2
a^S_2	a^N_5	150	133	25	68	6	230	25	76	23
a^S_3	-	158	121	21	76	-	-	-	55	-
a^S_5	-	171	146	30	63	-	-	-	27	-
a^S_6	a^N_8	177	138	27	105	18	170	18	22	1
a^S_0	a^N_6	180	147	30	66	5	169	18	62	17
-	a^N_{10}	184	-	-	124	-	268	30	90	-
-	a^N_0	186	-	-	73	-	20	0	20	-

***Notations of Table 1*:**

$\mathbf{X}_s$, $\mathbf{X}_n$, $\mathbf{Y}_s$, $\mathbf{Y}_n$ – *x*- and *y*-coordinates of surgeon's and nurse's wrists

$\tilde{\mathbf{X}}_s$, $\tilde{\mathbf{Y}}_s$, $\tilde{\mathbf{X}}_n$, $\tilde{\mathbf{Y}}_n$ - normalized in interval [0,30] coordinates $\mathbf{X}_s$, $\mathbf{Y}_s$, $\mathbf{X}_n$, $\mathbf{Y}_n$

TS – switching event time stamp

Switching events of Nurse's Gestures:

a^N_0 – idle
a^N_1 – prepare instrument
a^N_2 – picking up an instrument
a^N_3 – holding the instrument & waiting
a^N_4 – passing the instrument
a^N_5 – wait returning
a^N_6 – withdrawing hand
a^N_7 – stretching hand
a^N_8 – receiving
a^N_9 – moving back
a^N_{10} – putting on the tray.

Switching events of Surgeon's gestures:

a^S_0 – idle,
a^S_1 – receiving an instrument;
a^S_2 – inserting instrument;
a^S_3 – working;
a^S_4 – waiting for an instrument;
a^S_5 – extracting from trocal cannula;
a^S_6 - returning the instrument.

<u>*Step 2:*</u> Check the inclusion of the transition and location representing red from E an event *e* in any already existing model element equivalence class. If the inclusion is established the algorithm returns to Step 1. If the model element is not in any existing equivalence class the new equivalence class is created and the algorithm returns to Step 1.

<u>*Step 3*</u> (*Model reduction*): Model reduction minimizes the set of state variables necessary for specifying transition guards of the TAIO model. Reduction must preserve determinism of the model. It means that for each location l_i its outgoing transitions' guards have to satisfy the condition that there exists at least one state variable x_k for each pair of transitions $\langle t(l_i, l_j .), t(l_i, l_k, .)\rangle$ where $j \neq k$ s.t. their guards are mutually exclusive, i.e., $\forall l_i \in Q$, $\forall t(l_i,l_j.)$, $t(l_i,l_k,.) \in$ D: $\neg(g(t(l_i,l_j.)) \wedge (g(t(l_i,l_k,.)))$.

<u>*Step 4*</u>: Construction of location invariants. For each location l_i the invariant $I(l_i)$ is constructed from the guards of incoming and outgoing transitions: $I(l_i) \equiv \wedge_k g(t(l_k,l_i.)) \wedge \neg \vee_j g(t(l_i,l_j.))$.

4. CASE STUDY: LAPAROSCOPIC SURGICAL PROCEDURE

Given:

- Observation sequence E (a sample fragment of the sequence E is specified in Table 1).
- System configuration as depicted in Fig 2.
- Rescaling operator RS_i with region [0,30] for all state variables $x_i \in X$.
- Robustness $R_i = 2$, for all $x_i \in X$.

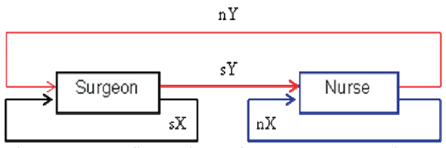


Fig. 2 IO configuration of Surgeon-Nurse interaction (self-loops denote self-dependencies)

Result:

The parallel composition of Surgeon's and Nurse's TAIO-s learned by their interaction during a laparoscopic surgery (IO observation sequence E of Table 1) is represented in Fig. 3.

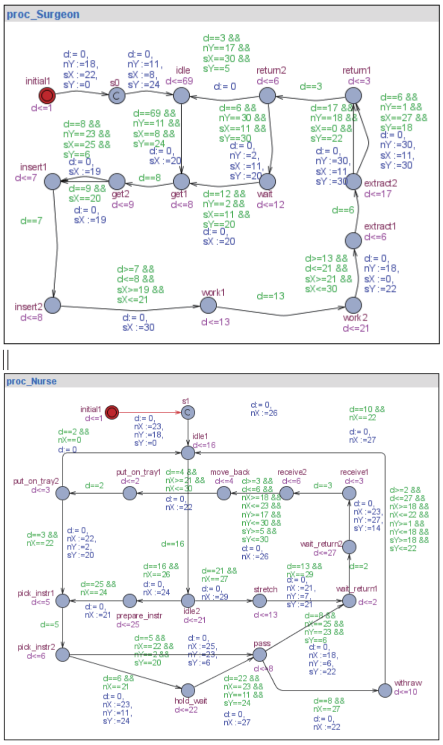


Fig. 3 Parallel composition of Surgeon's and Nurse's TAIO learned from Table 1.

5. CONCLUSION

We have proposed a timed IO automata model learning approach that makes the scrub nurse robot high level action planner synthesis on-line feasible. The advantages of proposed unsupervised learning approach are following: (*i*) learning is incremental, i.e., pre-existing knowledge about human scrub nurse behaviour can be re-used; (*ii*) in the presence of predefined scenario models the functional correctness of learning results and the efficiency of the robot action can be verified by model checking before used for actual planning; (*iii*) the learning algorithm can be easily tuned by choosing different value sets for its robustness and projection parameters; the last allows generating families of models with different level of abstraction and of different profile.

ACKNOWLEDGEMENTS

The work was financially supported by the Estonian Science Foundation Grants 6884, 7667, by research funding BF99 of Tallinn University of Technology, and also financially supported by a Grant-in-Aid for Scientific Research (B) (grant number19300186) from the Ministry of Education, Culture,Sports, Science and Technology of the Japanese Government.

REFERENCES

[1] M.Ogino, H. Toichi, Y. Yoshikawa, M. Asada. Interaction rule learning with a human partner based on an imitation faculty with a simple visuo-motor mapping. Robotics and Autonomous Systems 54, 2006. 414–418.

[2] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, J. Vain, Scrub nurse robot system - intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery. *IEEE Trans. on Industrial Electronics*, *52*(5), 2005, 1227-1235.

[3] M.J. Mataric. Learning in Behavior-Based Multi-Robot Systems: Policies, Models, and Other Agents. Cognitive Systems Research, 2(1), Apr 2001, 81-93.

[4] J. Jakubiak, S. Nõmm, J. Vain, F. Miyawaki. Polynomial based approach in analysis and detection of surgeon's motions. In: ICARCV 2008 10th international Conference on Control, Automation, Robotics & Vision: 2008, Hanoi, Vietnam: N.J.: IEEE, 2008, 611 - 616.

[5] S. Nõmm, E. Petlenkov, J. Vain, J. Belikov, F. Miyawaki, K. Yoshimitsu. Recognition of the surgeon's motions during endoscopic operation by statistics based algorithm and neural networks based ANARX models. IFAC 17th World Congress, Seoul, Korea, 2008, 14773 - 14778.

[6] E. Petlenkov, S. Nõmm, J. Vain, F. Miyawaki. Application of self organizing Kohonen Map to detection of surgeon motions during endoscopic surgery. International Joint Conference on Neural Networks (IJCNN2008), Hong Kong, 2008, 2807 - 2812.

[7] D. Kaynar, N. A. Lynch. Decomposing verification of timed I/O automata. In Y. Lakhnech and S. Yovine, editors, Lecture Notes in Computer Science 3253, 2004, 84-101.

[8] O. Grinchtein, B. Jonsson, M. Leucker. Learning of Event-Recording Automata. Y.Lakhnech and S.Yovine (Eds.): FORMATS/ FTRTFT 2004, LNCS 3253, 379–395.

[9] M. Krichen, S. Tripakis. Interesting Properties of the Real-Time Conformance Relation TIOCO. K. Barkaoui, *et al* (Eds.): ICTAC 2006, LNCS 4281, 317–331.

[10] http://www.uppaal.com

[11] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[12] M.Kearns, U.Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

APPENDIX A

```
1: while $E \neq \emptyset$ do
2:     $e \leftarrow get(E)$ % get event $e$ from buffer E
3:     $h' \leftarrow h$, $h \leftarrow l$
4:     $l \leftarrow e[1]$, $cl \leftarrow (e[2] - h_{cl})$, $\boldsymbol{X} \leftarrow e[3]$
5:     if $l \notin L$ then                  % if new event type
6:         $L \leftarrow L \cup \{l\}$,
7:         $T \leftarrow T \cup \{t(h,l,1)\}$      % add transition
8:         $g\_cl(h,l,1) \leftarrow [cl, cl]$ % add cl. reset
9:         for all $\boldsymbol{x_i} \in \boldsymbol{X}$ do
10:        $g\_x(h,l,1,x_i) \leftarrow [\boldsymbol{x_i}, \boldsymbol{x_i}]$ % add grd
11:        end for
12:    else                  % if $e$ in existing eqv. class
13:        if $\exists k \in [1,|t(h,l,.)|]$, $\forall \boldsymbol{x_i} \in \boldsymbol{X}$,: $\boldsymbol{x_i} \in$
           $g\_x(h,l,k,x_i) \wedge cl \in g\_cl(h,l,k)$ then
14:            goto 34
15:        else       % if $e$ extends the eqv. class
16:            if $\exists k \in [1,|t(h,l,.)|]$, $\forall \boldsymbol{x_i} \in \boldsymbol{X}$:   $\boldsymbol{x_i} \in$
        $g\_x(h,l,k,x_i)^{\updownarrow R_i} \wedge cl \in g\_cl(h,l,k)^{\updownarrow Rcl}$      (*)
17:            then
18:                if cl $< g\_cl(h,l,k)^-$  then
     $g\_cl(h,l,k) \leftarrow [cl, g\_cl(h,l,k)^+]$ end if
19:                if cl $> g\_cl(h,l,k)^+$ then
     $g\_cl(h,l,k) \leftarrow [g\_cl(h,l,k)^-, cl]$   end if
20:                for all $\boldsymbol{x_i} \in \boldsymbol{X}$   do
21:              if $\boldsymbol{x_i} < g\_x(h,l,k,x_i)^-$     then
     $g\_x(h,l,k,x_i) \leftarrow [\boldsymbol{x_i}, g\_x(h,l,k,x_i)^+]$ end if
22:              if $\boldsymbol{x_i} > g\_x(h,l,k,x_i)^+$    then
     $g\_x(h,l,k,x_i) \leftarrow [g\_x(h,l,k,x_i)^-, \boldsymbol{x_i}]$ end if
23:                end for
24:               else          % if $e$ not in eqv. class
25:                  $k \leftarrow |t(h,l,.)| + 1$
26:                  $T \leftarrow T \cup \{t(h,l,k)\}$ % add trn
27:                  $g\_cl(h,l,k) \leftarrow [cl,cl]$ % cl reset
28:                  for all $\boldsymbol{x_i} \in \boldsymbol{X}$ do    % add grd
29:                      $g\_x(h,l,k,x_i) \leftarrow [\boldsymbol{x_i}, \boldsymbol{x_i}]$
30:                  end for
31:        end if
32:    end if
33:    $a(h',h,k') \leftarrow a(h',h,k') \cup \boldsymbol{X_c}$      % add asg
34: end while
```

(*) - *Interval extension operators* $^{\updownarrow R}$: $[x^-, x^+]^{\updownarrow R} = [x^- - \delta, x^+ + \delta]$, where $\delta = R - (x^+ - x^-)$

Appendix III: Supervised Training of Voting Automata for the Surgeon-s Motion

2009 IEEE International Conference on Control and Automation
Christchurch, New Zealand, December 9-11, 2009

Supervised Training of Voting Automata for the Surgeon's Motion Recognition During Laparoscope Surgery

Jüri Vain, Sven Nõmm, Aivo Anier, Fujio Miyawaki and Tatiana Totskaya

Abstract— **Supervised learning of voting automata for the surgeon's right hand motion recognition constitutes the main result reported in the present paper. Within the framework of the project, aiming the design of scrub nurse robot a number of methods for recognizing the current stage of the surgery has been developed. Obviously no one of the methods separately can guarantee hundred percent correct recognition. Therefore, the voting automaton is employed to choose the best result. In this paper main attention is paid to the design of such voting automata using supervised learning techniques.**

I. INTRODUCTION

Cooperation between human and robotic assistant requires the robot to model human behavior in order to make a decision about timing and type of the assisting action. On one side, recognition of of human behavior relies on the context, i.e. the world model consisting of behaviour models of interacting parties and the reference scenario of the procedure to be accomplished, and on the other side, precise information about the current state of action. In general, the online information about the surgery status can be obtained by means of different communication channels like video, voice or sensor information. Within the framework of present contribution we will relay only on the video data transformed into the stream of coordinates describing position of the surgeon's hands in real time.

The results of the present contribution lie in the area where the robot is designed to replace a human scrub nurse (specially trained nurse, who assists directly the operating surgeon) during laparoscope surgery [4]. Laparoscope surgery was chosen as a starting point due to its relatively simple operation scenario, which excludes a lot of uncertainties allowing to concentrate efforts on development of basic solution. Within the framework of the project a number of recognition techniques has been developed, namely, Neural Networks based (NN-based) [6], statistic probabilistic technique [6], self organizing map based technique [8] and a technique based on trajectory parametrization [1]. Since all those methods have their own strength and weaknesses in [7] a hybrid detection technique was proposed to improve the quality of detection results. The voting automaton proposed in [7] based its decision (if next stage of the surgery has begun or not) on the analysis of outputs of different recognition functions. Such approach allowed to improve the quality of recognition by choosing the output of the recognition function which demonstrated the best performance in detection of expected stage of the surgery. Since the voting automaton in [7] was designed "manually" just to use three detection techniques (NN-based, statistic and self-organizing maps based), any change in the number of detection techniques, or their parameters would require redesign of voting automaton which is time and effort consuming process. This led to the idea to automatize the voting automata construction process. In the present paper, supervised learning techniques for finite state machine would be employed for this purpose.

The paper is organized as follows. Section 2 provides general overview of the operating environment and robot construction from the dataflow point of view. In order to make this paper self-sufficient overview of the detection techniques (NN-based, statistic-based and self-organizing maps based) is given in the section 3. Section 4 is devoted to the presentation of main results. Section 5 contains comparison results where performance of entire hybrid detection system is compared to the performance of each technique alone. Conclusions are drawn in the last section.

II. GENERAL OVERVIEW OF THE PROJECT

Scrub nurse robot (SNR) is a specialized medical robot which was designed to perform assisting actions of a scrub nurse (see Figure 1). Detailed description of the SNR design has been presented in [9].

In order to provide greater vision coverage area and reduce the possibility of eclipsing the robot video system four cameras were positioned outside the SNR in fixed positions of the surgical room. Since SNR is designed to function only within the surgical room, such design does not limit robot abilities. Main functional modules are depicted in Figure 2.

From the viewpoint of a human scrub nurse cognition the surgeon's hand motions during laparoscope surgery consists of the six basic gestures or *motions*. Well trained human scrub nurse can distinguish those gestures based on the knowledge of operation scenario and by observing the operating surgeon's hand movements.

This work was partially supported by Estonian Science Foundation grants # 7667 # 6884

J. Vain is with the Institute of Computer Science Tallinn University of Technology, Raja 1, 12618, Tallinn, Estonia vain@cc.ioc.ee

S. Nõmm is with Institute of Cybernetics, Tallinn University of Technology, Akadeemia tee 21, 12618, Tallinn, Estonia sven@cc.ioc.ee

A. Anier is with the Institute of Computer Science Tallinn University of Technology, Raja 1, 12618, Tallinn, Estonia aivo.anier@gmail.com

F. Miyawaki is with Grad. School of Advanced Sc. & Tech. TDU, Ishizaka, Hatoyama-machi, Hiki-gun, Saitama, 350-0394, Japan miyawaki@b.dendai.ac.jp

T. Totskaya is with the Institute of Computer Science Tallinn University of Technology, Raja 1, 12618, Tallinn, Estonia totskaya.tatiana@gmail.com

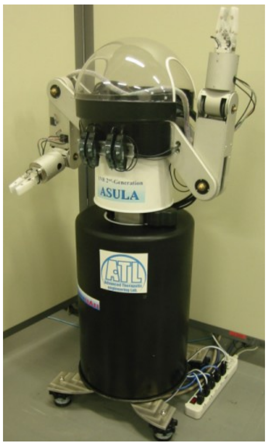


Fig. 1. Scrub nurse robot

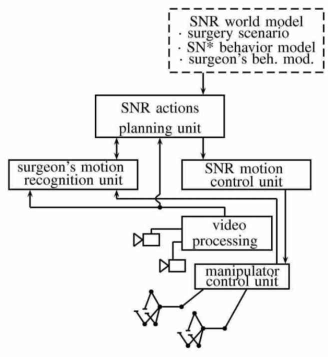


Fig. 2. Functional model of the scrub nurse robot

1) 'inserting' is defined as the motion observed from the moment that instrument is received by surgeon's hand up to the moment when the surgeon inserted it into the abdominal cavity through the cannula (cylindrical tube);
2) 'working' is defined as the motion observed while the surgeon was conducting some kind of surgical procedure with a surgical instrument;
3) 'extracting' is defined as the motion which starts from the moment when the surgeon began to draw the surgical instrument out of the operative field and which lasts until the moment the tip of the instrument comes out of the inlet of a trocar cannula penetrating the abdominal wall;
4) 'passing' is defined as the motion which lasts from the moment that the tip of instrument came out of the trocar cannula to the moment the surgeon releases the instrument by passing it to the scrub nurse or returning it to a surgical tray by himself;
5) 'waiting' is defined as the motion continuing from the moment the surgeon releases the instrument to the moment when he receives the next instrument from the scrub nurse;
6) 'get' is defined as the motion observed while surgeon takes the instrument from the nurse. Compared to other motions '*get*' takes place during extremely short period of time. In spite of that, "get" will be called motion in the sense that it corresponds to the very important stage of the surgery);

The necessity of surgeon's movements segmentation into six motions is caused by the human cognition, since in human-human communication those six motions are the basis for successful cooperation between surgeon and scrub nurse. The robot replacing a human scrub nurse should closely follow this behavioral pattern and, therefore, is required to be able to recognize those six motions.

The video system of the robot is tracking the small markers attached to the surgeon chest, elbow and wrist. With interval $1/60$ of a second it returns 3D coordinates of each marker. In other words, video system provides a stream of coordinates describing position of surgeon's right hand in real-time. Observing this stream the *SNR motions recognition unit* detects the switching events between motions. Once the change in surgeon's right hand motion is detected by the recognition unit corresponding information is sent to the SNR actions planning unit which is implemented in the form of timed automata [4]. Based on the SNR world model and surgery (operation) scenario it generates orders for the motion control unit, which controls manipulators directly.

III. Motion Recognition Techniques

In [7] the hybrid detection system that combines three motion recognition techniques and a voting automaton was presented. In addition to the fact that the mathematical nature of NN-based, the statistics-based and Kohonen-map based techniques differs: the statistics-based technique is designed to detect the points in time when one motion ends and the next one begins, while NN-based and Kohonen-map techniques detect the type of the motion. Within the framework of this project such points will be referred as *switching points* or simply *switchings*. First two techniques require supervision, namely, a set of human made segmentations for training purposes. Producing such human made segmentation is time and effort consuming process where human should review a video recording frame by frame and based on his/her knowledge place the flag in time-line where switching from one motion to another took place.

A. Neural Networks based technique

Proposed in [6] technique is based on training a neural network to recognize the type of the motion which is taking place at the current moment. The model represents a restricted connectivity neural network, can be considered as a generalization of the NN-based ANARX model class [3] for the MIMO case.

$$\Big[m(t)\Big]^T = \sum_{i=1}^{3} C_1 f_i \left(W_i \Big[\alpha(t-i), \beta(t-i), \gamma(t-i)\Big]^T \right). \quad (1)$$

Here α, β and γ are vectors containing coordinates of the markers attached to the chest, elbow and wrist of the operating surgeon at time t, $m(t)$ is a vector composed of six elements. Neural network, corresponding to the model (1) was trained to return vector $m(t)$ with 1 in the position corresponding to the number of motion taking place in the current moment and zeros in all other positions. Of course, the number of human-made segmentations is required for training. For example vector $\{0, 1, 0, 0, 0, 0\}$ would correspond to the motion number 2 or *working*. Such vectors are created on the basis of switching flags positioned by human in time-line. Obviously, the trained network would not return vector of just zeros and one. The values of the output vector of trained network would belong to certain neighborhood of zero and one, which can be rounded to the closest value.

B. Statistic-Probabilistic approach

Positions of the operating surgeon and scrub nurse are fixed in relation to the operation table and each other therefore their hands trajectories in each experiment should follow approximately the same path. This leads to the idea that switching points of the same type should be somehow grouped in certain locations (or inside certain convex sets in the operating room [5]). If the surgeon's wrist trajectory passes through such convex set, motion switching can be declared. On the basis of sufficiently large number of human made segmentations one can find "average switching points" for each pair of consequent motions. Assuming that switching points are normally distributed around corresponding average switching points, and taking into account that ellipsoids represent equidensity contours for three dimensional normal distribution one can consider those ellipsoids as as a convex sets containing corresponding switching points. Obviously the majority of trajectories would not pass through the average switching point but in certain proximity of it. For each point of trajectory approaching certain switching one can find a properly oriented ellipsoid such that this point would belong to its surface and therefore determine a probability that by this time switching is already occurred. On the basis of sufficiently large number of experiments one can determine liminal probability values for declaring switching for each pair of consequent motions, in other words, finding radiuses of ellipsoids which should be crossed by trajectory. Such setting will lead to the segmentation of working room or working area by ellipsoids around switching points, depicted in Figure 3.

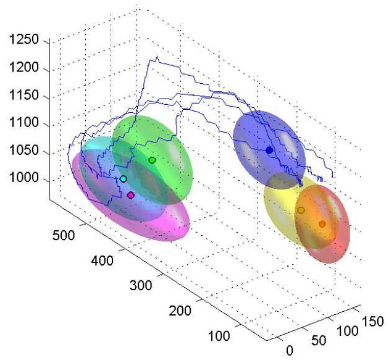


Fig. 3. Segmentation of working space induced by average switching points and liminal switching probabilities

C. Self-Organizing Map-Based Technique

Unlike the two previous techniques the method proposed in [8] does not require human made segmentation for training. Kohonen-map [2] based technique classifies data into given number of classes and calculates so called "reference vectors" for each class (denoted as $W_{class\quad number}$). The only human intervention is required to define which class corresponds to which motion. Depicted in Figure 4 is

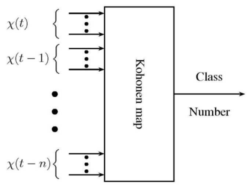


Fig. 4. Schematic diagram of the Kohonen map

the input vector $\Xi(t) = \{\chi(t), \chi(t-1), \chi(t-2)\}$ at each time step composed of the coordinates of chest, elbow and wrist collected during last 3 time steps. At each time step following norm is calculated for each reference vector W_i

$$\|\Xi(t) - W_i\| \quad (2)$$

Vector $\Xi(t)$ is declared to belong to the class which reference vector will result in minimum value of (2). Once the system detects that following vectors do not belong to the same class as previous one, the motion switching is declared.

D. Hybrid Detection Approach

Each of above mentioned techniques has its own advantages and drawbacks. NN-based approach provides accurate recognition of motions which took place for longer periods of time. For motions lasting shorter periods of time its performance less accurate. Statistics-based method demonstrates quite robust performance but the average error (early or late detection) is high. Kohonen-map based technique demonstrates good results dealing with motions lasting for shorter periods of time while ”longer” motions can pose a problem. In order to combine the advantages of all techniques and to improve the detection quality, hybrid detection method was proposed in [7]. Simulink implementation of the hybrid detection approach is depicted in Figure 5.

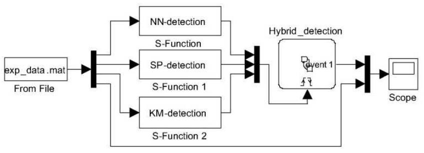


Fig. 5. SIMULINK scheme implementing hybrid motion recognition system

Implementation of each detection technique sends a signal to the voting automaton in Figure 6 once it detects a switching. If no switching is detected recognition functions are silent, generate no output. Such input signal is considered by voting automata as an event which will drive it from state ”idle” to the state where next possible switching event triggers the following transition.

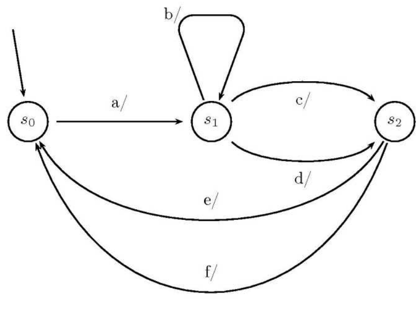


Fig. 6. Voting automata

Interpretation of the events, and states is described in the Table I.

IV. Construction of Learning Automata

When integrating just three techniques introduced above one can construct the voting automata like one in Figure 6

TABLE I

EVENTS AND STATES OF THE VOTING AUTOMATA

event	interpretation
a	one of the functions detected switching
b	time limit or other detection
c	full information obtained
d	time-out
e	positive feed-back (correct detection)
f	failed detection
state	interpretation
s_0	idle, waiting for the detection flag
s_1	collecting information during certain period of time
s_2	voting and generating output signal

manually. Obviously, given approach does not allow neither easy modifications when the recognition method changes nor adding new methods. In order to overcome this problem an auction based motion switching detection mechanism is suggested that supports easy reconfigurability and extendability of the methods pool used in recognition.

A. Auction Based Motion Switching Detection

The auction based motion switching detection system in Figure 7 consists of detection methods each of which is interfaced with the Arbiter (an auctioner) through a Proxy automaton.

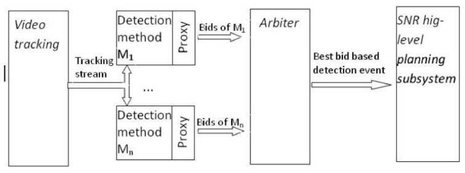


Fig. 7. Auction based switching detection system

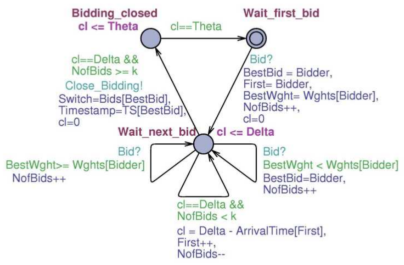


Fig. 8. Bidding Arbiter automaton

The Arbiter automaton depicted in Figure 8 collects the bids from Proxies depicted in Figure 9 and chooses the best bid to be forward to the high-level action planning subsystem.

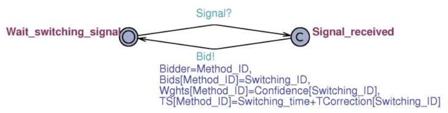

Signal?
Wait_switching_signal
Signal_received
Bid!
Bidder=Method_ID,
Bids[Method_ID]=Switching_ID,
Wghts[Method_ID]=Confidence[Switching_ID],
TS[Method_ID]=Switching_time+TCorrection[Switching_ID]

Fig. 9. Method Proxy automaton

The bidding mechanism implemented by means of timed automata composition depicted in Figure 10 performs as follows. Detection algorithms $M_1, \dots, M_n$ are observing the stream of coordinates received from video tracking subsystem. When detecting the motion switching events the proxy automata $A_1, \dots, A_n$ *listening* to methods $M_1, \dots, M_n$ respectively, communicate the detection event of a method as a bid to the Arbiter automaton. After receiving the first bid on some switching e_i the Arbiter waits Δ time units collecting bids from other methods on e_i and after Δ closes the bidding. Note that Δ is determined by the reaction time requested from the switching detection system. Thereafter, Arbiter chooses the bid with the highest weight received during Δ and forwards it to SNR high-level behavior planning subsystem. To suppress the fault bids arrived after Δ time Arbiter keeps the auction closed at least ς time units. To avoid too early bids or false bidding start the following Δ-window sliding procedure is used. The bidding is inconclusive if the number of bids is not representative (less than k) during Δ and in that case the bidding is not closed after Δ. Instead, the beginning of time window Δ is shifted from the first triggering bid to the next one received. Shift of Δ is repeated until at lest critical number k of bids has been registered within Δ. Parameters Δ and ς strongly depend on the concrete gesture sequencing pattern to be detected and detection response time constraints.

B. Learning Configuration and Procedure

The Proxy automata in the auction based decision making system described in Subsection IV-A are constructed as a result of supervised learning procedure. The learning configuration shown in Figure 10 includes Supervisor automaton SA, depicted in Figure 11 that guides the learning process and proxy automata A_i, depicted in Figure 12 that mediate the learning dialogue between the detection method M_i and Supervisor. The Supervisor automaton and a number of proxy automata together constitute the automata composition for tuning motion switching recognition methods. For evaluating and adjusting the motion switching recognition capabilities of methods $M_1, \dots, M_n$ the Supervisor presents to detection algorithms a test trajectory. We assume that Supervisor has a priori knowledge about the motions segmentation, i.e. reference events $\underline{e_i}$ of type E_k and time instances $TS(\underline{e_i})$ of their occurrence are defined by human expert.

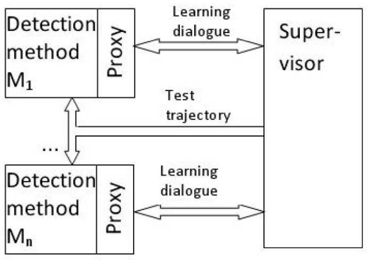


Fig. 10. Configuration of the learning system

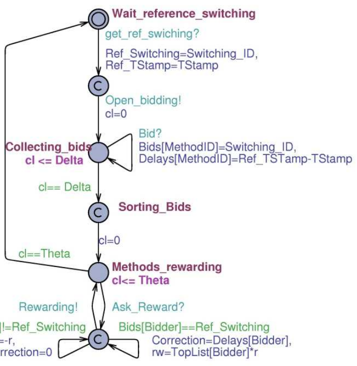


Fig. 11. Supervisor automaton of the learning configuration

Based on the method's M_j response e^i_j to the segmentation query (bidding start) on event e_i and the type E_k of reference event e_i, the Supervisor orders the bids made by methods $M_1, \dots, M_n$ by their time error and calculates reward rw to each Bidding automaton according to the formula (3).

$$rw = \begin{cases} (n - p^j_i + 1)r, & if \quad e^j_i = \underline{e_i} \\ -r, & \text{otherwise} \end{cases} \tag{3}$$

where n-total number of bids in the bidding session on e_i, p^j_i-position of the bid made by A_j on e_i, r-unit reward.

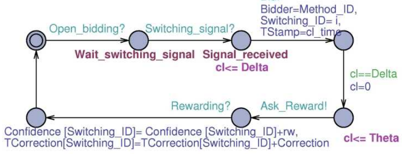
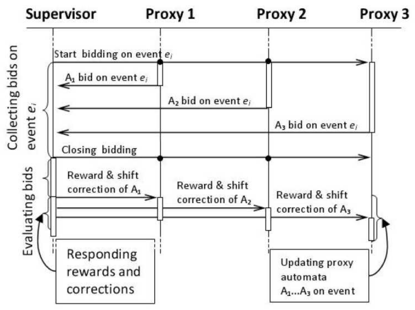


Fig. 12. Proxy automaton of the learning configuration

The detection time error $\delta(e^j_i) = TS(\underline{e_i}) - TS(e^j_i)$ and a reward rw^j_i for bid e^j_i are returned to the bidding automaton



Supervisor
Proxy 1
Proxy 2
Proxy 3
Collecting bids on event e_i
Start bidding on event e_i
A_1 bid on event e_i
A_2 bid on event e_i
A_3 bid on event e_i
Closing bidding
Evaluating bids
Reward & shift correction of A_1
Reward & shift correction of A_2
Reward & shift correction of A_3
Responding rewards and corrections
Updating proxy automata A_1...A_3 on event

Fig. 13. Message sequence chart of the learning dialogue on event e_i

A_j. The automaton A_j receiving the reward and time error on bid e_i^j makes the correction of the method's M_j weight according to the formula (4)

$$W_k^j := W_k^j + rw \tag{4}$$

and time error correction according to the formula (5).

$$\delta(TS) := i^{-1} * (TS * (i-1) + \delta(e_i^j)) \tag{5}$$

Thus, the occurrence of each reference event e_i of type E_i meant to train the proxy automata in the teaching process triggers the learning dialogue between the Supervisor and the method's Proxy automaton as depicted in Figure 13

V. Discussion

Simple example below illustrates simulation of hybrid detection system, which consists of auction based switching and three detection techniques. It can be seen in Figure 14 that the hybrid detection system detects switchings between motions in much more accurate way than the methods alone, there is still chance that for certain switching non of the methods can provide necessary quality. One of the possible ways to overcome this problem is to select methods in such a way that they will "cover" weaknesses of each other. Obviously more detailed research to answer the question what causes failure of certain methods is required.

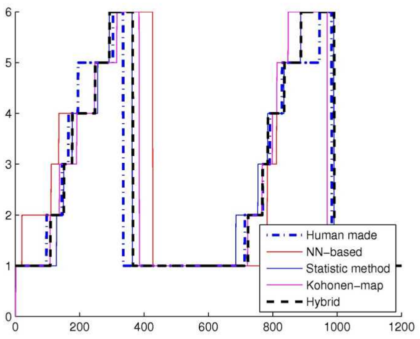


Fig. 14. Comparison of different detection techniques

VI. Conclusions

Main result of present contribution is a way of automatizing the construction of the automata playing a role of arbiter in the hybrid motion recognition system that is meant to recognize different stages of the laparoscope surgery by observing hand movements of the operating surgeon. Such automatization allows to adjust parameters of hybrid detection system or even include new detection techniques in an easy and efficient way. Such automatization ability has a crucial importance not only in developing system but in using it, as it will have to adopt its parameters to operating surgeon. Future research will be pointed towards formalization of notion "motion" and performance improvement of the existing techniques.

References

[1] J. Jakubiak, S. Nõmm, J. Vain, and F. Miyawaki. Polynomial based approach in analysis and detection of surgeon's motions. In *In: Proc of the ICARCV 2008 10th IEEE International Conference on Control, Automation, Robotics & Vision*, pages 611–616. N.J.: IEEE, 2008, Hanoi, Vietnam, 2008.

[2] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, (78):1464–1480, 1990.

[3] Ü. Kotta, F. Chowdhury, and S. Nõmm. On realizability of neural networks-based input-output models in the classical state space form. *Automatica*, 42(6):1211–1216, 2006.

[4] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, and J. Vain. Scrub nurse robot system - intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery. *IEEE Industrial Electronics Transaction on*, 5(52):1227–1235, 2005.

[5] S. Nõmm, E. Petlenkov, J. Vain, F. Miyawaki, and K. Yoshimitsu. Recognition of the surgeon's motions during endoscopic operation by statistics based algorithm and neural networks based anarx models. In *Proceedings of the 17th IFAC World Congress*, pages 14773–14778. Elsiver, Soul, Korea, July 2008.

[6] S. Nõmm, E. Petlenkov, J. Vain, K. Yoshimitsu, K. Ohnuma, T. Sadahiro, and F. Miyawaki. Nn-based anarx model of the surgeon's hand for the motion recognition. In *Proceedings of the 4th COE Workshop on Human Adaptive Mechatronics (HAM)*, pages 19–24. Tokyo Denki University, Tokyo, Japan, March 2007.

[7] S. Nõmm, J. Vain, E. Petlenkov, F. Miyawaki, and K. Yoshimitsu. Hybrid approach to detection of the surgeon's hand motions during endoscope surgery. In *In: Proc of the he 4th IEEE Conference on Industrial Electronics and Applications (ICIEA 2009)*, page Accepted. Xi'an, Peoples Republic of China, 2009.

[8] E. Petlenkov, S. Nõmm, J. Vain, and F. Miyawaki. Application of self organizing kohonen map to detection of surgeon motions during endoscopic surgery. In *Proc: 2008 IEEE World Congress on Computational Intelligence (WCCI 2008)*, pages 2806–2811. Hong-Kong, 2008.

[9] K. Yoshimitsu, F.Miyawaki, T. Sadahiro, K. Ohnuma, Y. Fukui, D. Hashimoto, and K. Masamune. Development and evaluation of the second version of scrub nurse robot (snr) for endoscopic and laparoscopic surgery. In *In Proc: Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, number 10.1109/IROS.2007.4399359, pages 2288–2294. San-Diego, California, USA, October 2007.

Appendix IV: Model based continual planning and contro for assistive robots

MODEL BASED CONTINUAL PLANNING AND CONTROL FOR ASSISTIVE ROBOTS

A. Anier and J. Vain
Tallinn University of Technology, Ehitajate tee 5, 19086 Tallinn, Estonia

Keywords: Model based control, Continual planning, Cognitive architecture, Online safety monitoring.

Abstract: The paper presents a model-based robot planning and control framework for human assistive robots - namely for Scrub Nurse Robots. We focus on endoscopic surgery as one of the most relevant surgery type for applying robot assistants. We demonstrate that our framework provides means for seamless integration of sensor data capture, cognitive functions for interpretation of sensor data, model based continual planning and actuation control. The novel component of the architecture is a distributed continual planning system implemented based on the Uppaal timed automata model-based verification and control tool suite. The distributed and modular architecture of the framework enables flexible online reconfiguration and easy adaptability to various application contexts. Online learning and safety monitoring functions ensure timely and safe updates of software components on-the-fly.

1 INTRODUCTION

The assistive robotics sets high standards to cognitive capabilities, autonomy and movement precision for robots. Functionally, it means understanding human intention and providing adequate reaction to it. Technically it means human-in-the-loop collaborative action control, fusion of various sensor information, high accuracy actuation and reliable software implementation. Action and trajectory planning safety issues become critical in the conditions where the robot shares user's working envelope to achieve required physical interaction.

This paper presents a software integration framework for *Scrub Nurse Robot* (SNR)(Miyawaki et al., 2005) focusing on distributed model based continual planning and control issues. The goal of a SNR is to learn the interactions between a surgeon and a scrub nurse during a laparoscopic surgery and to replace the (human) nurse on demand. The key aspect for incorporating the SNR in the collaborative action (e.g. when the human scrub nurse has to deal with unexpected emergencies) is to avoid the need for the surgeon to re-adapt to the changed partnerwhile still preserving the "original feel" and the accustomed workflow. A physical scene of a SNR example deployment is shown in Fig.1.

A scrub nurse must hand a surgical instrument to a surgeon as soon as it is requested. If the scrub nurse has to spend time searching for the instrument after the request the procedure is interrupted, valuable time is lost and an unnecessary burden is placed on the surgeon. That possibly reduces the quality and effectiveness of the operation. The scrub nurse must be fully attentive to the activity in the operative field and anticipate accurately what a surgeon will need to avoid delays. For this to be possible the scrub nurse not only needs to know the surgical procedure as well as the surgeon does, but must also be highly disciplined. The "ideal" scrub nurse (if one exists) is able to pass a surgeon whatever is needed without any verbal order at the moment that the surgeon's hand is extended to receive it.

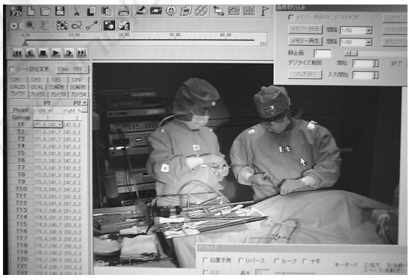


Figure 1: SNR intraoperative scene(Miyawaki et al., 2005).

The goal of the SNR software project is to develop a human-adaptive SNR capable of adapting to

Anier A. and Vain J. (2012).
MODEL BASED CONTINUAL PLANNING AND CONTROL FOR ASSISTIVE ROBOTS.
In *Proceedings of the International Conference on Health Informatics*, pages 382-385
DOI: 10.5220/0003783503820385
Copyright © SciTePress

surgeons with various levels of skill and experience, also to different personalities and moods . In other words, the SNR ought to function as an “ideal” scrub nurse. Highly developed cognitive faculties such as machine vision and speech recognition as well as adaptive robotic arm path planning and targeting are required to attain this ideal.

In conventional surgical operations a scrub nurse frequently has to handle an array of different instruments. It is difficult to make the SNR adaptive to such busy operations. Therefore, the SNR prototype has been designed for endoscopic surgery which only needs limited types of surgical instruments. The adaptivity of the SNR requires unsupervised learning by observing skilled nurses’ interactions and behavior during surgical operations.

Online recognition and anticipation of surgeon’s motions while operating is essential to classify which motions are common to all surgeons and which are specific to individuals. This in turn will aid in anticipating a surgeon’s needs and in adapting to the changes of procedure. On the other hand - the results of the investigation of intraoperative behavior have to be abstracted and memorized in the form of mathematical and/or formal models in order to reproduce the variety of motion trajectories that can be expected from various combinations of surgical procedures and varying external factors. The model of a nurse’s behaviors as he or she reacts to other surgical staff (surgeon, assistant and others) serves as a high-level behavior specification for the SNR action planning.

The SNR’s control architecture depicted in Fig.2 comprises of the following components: 3D position tracking system that is capable of measuring the position-tracking marker’s coordinates with precision more than 1 mm with sampling rate up to 200 fps. The surgeon’s hand movement sampling data is passed to gesture recognition module that uses multiple recognition methods in parallel. These methods of detecting operator’s current motions and the voting mechanism(Vain et al., 2009) maximize the confidence of the recognition. The identified motion and its parameters are inputs for reactive motion planning that compares the observed movement of surgeon’s hand with that of predicted by surgeon’s behavior model and surgery scenario model.

Such online conformance monitoring allows to correct the current model state with precision of minimum sampling error. By the corrected state information and surgery scenario model the next SNR action is planned and the resulting control parameters are transferred to the actuation control unit of SNR. The information about surgeon’s possible reactions predicted by the surgeon’s model is returned to the motion recognition module for discrimination of the decisions space when new movement is being recognized.

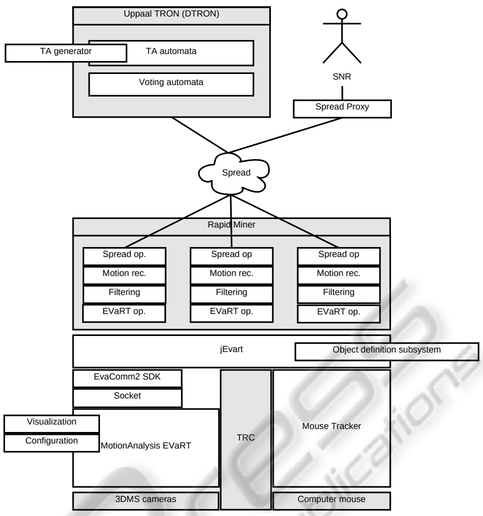


Figure 2: Architecture model.

The control architecture described above is implemented based on the open middle-ware platform discussed more thoroughly in the following sections.

2 SOFTWARE ARCHITECTURE

2.1 Data Acquisition

SNR doesn’t have integrated vision. Instead, the visual feedback control is implemented by means of external MotionAnalysis Hawk near-infrared active 3D measurement system (3DMS). 3DMS is not the only source of information. There are various sensors to monitor the state of the robot and peripheral interfaces that contribute to the overall situation and context awareness. For instance, the position data of surgical instruments is backed by RFID readings of ceramic RFID tag positions that are attached to the instruments. Abdominal video imaging from laparoscopic camera provides more accurate information about the course of surgery.

Middle-ware *jEvart* unifies 3DMS data with other data acquisition sources and passes to data analysis and cognitive modules implemented by means of Rapid Miner tool - www.rapid-i.com.

2.2 Data Analysis and Cognitive Functions

The robot control framework and middle-ware provide a common platform for integration of data acquisition and cognitive functions.

Data analysis and cognitive functions are implemented by means of data mining toolkit Rapid Miner. It includes hundreds of algorithms ranging from filtering and clustering to machine learning packaged into an integrated development environment. Rapid Miner is inspired by WEKA machine learning toolkit(Hall et al., 2009) improved with extensive data visualization and analysis automation tools.

To make the Rapid Miner fit the SNR overall control architecture some custom plug-ins are implemented. Specifically, it concerns the data acquisition components to capture the data available for analysis and visualization, but also the DTRON plug-in that bridges cognitive functions to deliberative control level functions. The deliberative control is based on provably correct timed automata models executed symbolically by DTRON tool.

3 DISTRIBUTED TRON

The SNR timed automata based action planning and control make use of Uppaal tool suite(Behrmann et al., 2004). Uppaal editor allows manual construction of timed automata in a way of visual programming paradigm. Limited functionality of various elements of the automata can be encoded using C-like functions. Although those functions make it somewhat easy to specify state transitions, their usage is prone to state space explosion. The Uppaal tool-suite includes an extension for *Testing Real-time systems Online* (*TRON*)(Hessel et al., 2008). Although TRON was originally developed for conformance testing, it also supports the functionality relevant to model-based discrete control. To interface the *TRON* model-based control module with controllable object requires “*adapters*” on the object side. Adapters intermediate and interpret the signals trafficking between the Uppaal automata and the control object. TRON was originally designed for single *tester-testee* pair and does not scale well with $n > 1$ *testers* and $m > 1$ *testees*. So it does not easily scale to distributed control applications. The main limitation of TRON usage is that it requires an extensive effort for adapter coding between controllers and control objects. When the adapter-controller pairs are tightly coupled every change in configuration requires re-wiring on both adapter ends.

Distributed TRON (*DTRON*) proposed in this paper is a framework built around the TRON tool to support multicast messaging between TRON instances running in parallel. In the ISO OSI networking architecture sense it implements the *whiteboard pattern* where *publishers* publish data and *subscribers* get notified about this. On the other hand, it embraces *the dependency injection* programming paradigm to make the *controller*-controllable object pairs *loosely coupled* for much better scaling.

To multicast is to send a message not to one recipient but to *n* recipients. *DTRON* is able to intercept the designated transitions within one control agent (model) and inform the other control agents of interests about it. The designation is defined by predicate on a *synchronized transition of the* controlling agent model. The synchronization and communication between agents is implemented by means of multicast message passing that allows the agents (dynamically) to join and leave a multicast whenever they want without the need to re-configure existing infrastructure. It only requires an agreement or protocol how messages are defined and what data they carry when they traverse the multicast.

4 CONTINUAL PLANNING AND CONTROL

Continual planning(DesJardins et al., 1999) denotes a planning strategy where the interactions between the controller and controllable object cannot be planned deterministically up front. The control signals have to be chosen depending on the situation as it emerges. The controller “knows” the state of the control object it tries to reach, but has limited control over stimuli or limited observation power of the control object behavior. The continual planning controller stimulates the object by limited set of stimuli step-by-step driving it towards the control goal by adjusting the stimuli to the control object responses.

Timed automata based planning and control suits for continual control due to its non-deterministic nature. Observations are mapped to automata structure and transition guards that encode the selection of stimuli to guide the (possibly) non-deterministic moves of the controllable object.

Uppaal comes with a formal verification engine that is used to establish weather a “plan” always drives the object to a desired state, provided the object responses are (at least partially) known. An extreme case would be a fully non-deterministic object that implies that it cannot be guaranteed or estimated which conditions should hold in order to reach the target state. This sets practical limits to the controllability for the SNR. If major deviations from prespecified scenario model occur the SNR would safely disengage human interaction from the working envelope and switches to manual override.

5 REACTIVE PLANNER

For continual planning and control the SNR actions in nondeterminstic situations are synthesized on-the-fly. The synthesis is based on the interaction model the SNR has learned by observing and recording Scrub Nurse and Surgeon's interactive behavior. The timed automata model learning algorithm used for that has been introduced in (Vain et al., 2009). The synthesis of reactive planning controller(Vain et al., 2011), that guides the SNR action when being active is based on the interaction model learned. The intended control goal of the SNR operation is encoded in the scenario automaton that specifies the sub-goals of the control, their temporal order and timing constraints. Whenever one of the sub-goals has been reached it triggers resets on guard conditions of the interaction model and activates driving conditions to reach the subsequent goal or one of the alternatives if multiple equal goals are reachable. In case of violating timing constraints or blocking an exception handling procedure or reset is activated and diagnostics recorded. Special care has been taken to address the safety precautions in SNR control. An independent safety monitoring process is running to check if all safety invariants are satisfied. Whenever safety violation is detected the disengagement procedure from continual planning unit is activated.

6 CONCLUSIONS

The cognitive robot architecture framework described in this paper supports several innovative aspects needed for implementing assisting robots in different applications. Our experience is based on the Scrub Nurse robot control architecture and software platform development exercise. We demonstrated that DTRON model-based distributed control framework provides flexible infrastructure for interfacing data acquisition and cognitive functions with the ones of deliberative control level planning and decision making. The architecture also incorporates a module for learning human interactions and model construction with reactive planning controller generator and runtime execution engine. The timed automata based interaction model learning, on-the-fly reactive planning, controller synthesis and online safety monitoring are steps towards the concept of provably correct robot design of *cognitive assisting robots*.

ACKNOWLEDGMENTS

This work was partially supported by the Estonian Science Foundation under grant No. 7667 and by Centre of Research Excellence in Dependable Embedded Systems - CREDES.

We want to thank Fuji Miyawaki for the productive discussions and suggestions on this subject.

REFERENCES

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. In Bernardo, M. and Corradini, F., editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, LNCS, page 200–236. Springer–Verlag.

DesJardins, M. E., Durfee, E. H., Ortiz Jr, C. L., and Wolverton, M. J. (1999). A survey of research in distributed, continual planning. *AI Magazine*, 20(4):13.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18.

Hessel, A., Larsen, K., Mikucionis, M., Nielsen, B., Pettersson, P., and Skou, A. (2008). Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, page 77–117.

Miyawaki, F., Masamune, K., Suzuki, S., Yoshimitsu, K., and Vain, J. (2005). Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery. *Industrial Electronics, IEEE Transactions on*, 52(5):1227 – 1235.

Vain, J., Kull, A., Kääramees, M., Maili, M., and Raiend, K. (2011). Reactive testing of nondeterministic systems by test purpose directed tester. In *Model-Based Testing for Embedded Systems.*, Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 425–452. CRC Press - Taylor & Francis Group, Massachusetts, USA.

Vain, J., Miyawaki, F., Nomm, S., Totskaya, T., and Anier, A. (2009). Human-robot interaction learning using timed automata. In *ICCAS-SICE, 2009*, pages 2037 –2042.

Appendix V: Provably Correct Test Generation for Online Testing of Timed Systems

Provably Correct Test Development for Timed Systems

Jüri VAIN [a,1], Aivo ANIER [a] and Evelin HALLING [a]

[a] *Department of Computer Science, Tallinn University of Technology, Estonia*

Abstract. Automated software testing is an increasing trend for improving the productivity and quality of software development processes. That, in turn, rises issues of trustability and conclusiveness of automatically generated tests and testing procedures. The main contribution of this paper is the methodology of proving the correctness of tests for remote testing of systems with time constraints. To demonstrate the feasibility of the approach we show how the abstract conformance tests are generated, verified and made practically executable on distributed model-based testing platform dTron.

Keywords. model-based testing, provably correct test generation, timed automata, verification by model checking

Introduction

The growing competition in software market forces manufacturers to release new products within shorter time frame and with lower cost. That imposes hard pressure to software quality. Extensive use of semi-automated testing approaches is an attempt to improve the quality of software and related development processes in industry. Although a wide spectrum of commercial and academic tools are available, the testing process still involves strong human factor and remains prone to human errors. Even fully automated approaches cannot guarantee trustable and conclusive testing unless the test automation is correct by construction or exhaustively covered with correctness checks. Test automation and test correctness are the main subjects of study in model based testing (MBT). Generally, MBT process comprises following steps: modelling the system under test, referred as Implementation-Under-Test (IUT), specifying the test purposes, generating the tests and executing them against IUT.

In this paper we study how the correctness of test derivation steps can be ensured and how to make the test results trustable throughout the testing process. In particular, we focus on model-based online testing of software systems with timing constraints capitalizing on the correctness of the test suite through test development and execution process. In case of conformance testing the IUT is considered as a black-box, i.e., only the inputs and outputs of the system are externally controllable and observable respectively. The aim of black-box conformance testing [1] is to check if the behaviour observable on sys-

[1]Corresponding Author: Jüri Vain; Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, 12618 Tallinn, Estonia; E-mail: juri.vain@ttu.ee

tem interface conforms to a given requirements specification. During testing a tester executes selected test cases on an IUT and emits a test verdict (pass, fail, inconclusive). The verdict is computed according to the specification and a input-output conformance relation (IOCO) between IUT and the specification. The behaviour of a IOCO-correct implementation should respect after some observations following restrictions: (i) the outputs produced by IUT should be the same as allowed in the specification; (ii) if a quiescent state (a situation where the system can not evolve without an input from the environment [2]) is reached in IUT, this should also be the case in the specification; (iii) any time an input is possible in the specification, this should also be the case in the implementation.

The set of tests that forms a test suite is structured into test cases, each addressing some specific test purpose. In MBT, the test cases are generated from formal models that specify the expected behaviour of the IUT and from the coverage criteria that constrain the behaviour defined in IUT model with only those addressed by the test purpose. In our approach Uppaal Timed Automata (UPTA) [3] are used as a formalism for modelling IUT behaviour. This choice is motivated by the need to test the IUT with timing constraints so that the impact of propagation delays between the IUT and the tester can be taken into account when the test cases are generated and executed against remote real-time systems. Another important aspect that needs to be addressed in remote testing is functional non-determinism of the IUT behaviour with respect to test inputs. For non-deterministic systems only online testing (generating test stimuli on-the fly) is applicable in contrast to that of deterministic systems where test sequences can be generated off-line. Second source of non-determinism in remote testing of real-time systems is communication latency between the tester and the IUT that may lead to interleaving of inputs and outputs. This affects the generation of inputs for the IUT and the observation of outputs that may trigger a wrong test verdict. This problem has been described in [4], where the Δ-testability criterion (Δ describes the communication latency) has been proposed. The Δ-testability criterion ensures that input/output interleaving never occurs.

1. Preliminaries

1.1. Uppaal Timed Automata

Uppaal Timed Automata (UPTA) [3] are widely used as one of the main modelling formalism for representing time constraints of software intensive systems. Before delving into test construction we shortly introduce the syntax and semantics of UPTA.

A timed automaton is given as a tuple $(L;E;V;Cl;Init;Inv;T_L)$. L is a finite set of locations, E is the set of edges defined by $E \in L \times G(Cl,V) \times Sync \times Act \times L$, where $G(Cl,V)$ is the set of transition enabling conditions - guards. *Sync* is a set of synchronization actions over channels. In the graphical notation, the locations are denoted by circles and transitions by arrows. An action *send* over a channel h is denoted by $h!$ and its co-action *receive* is denoted by $h?$. *Act* is a set of sequences of assignment actions with integer and boolean expressions as well as with clock resets. V denotes the set of integer and boolean variables. Cl denotes the set of real-valued clocks ($Cl \cap V = \emptyset$).

$Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks. $Inv : L \to I(Cl,V)$ is a function that assigns an invariant to each location, $I(Cl,V)$ is the set of invariants over clocks Cl and variables V. $T_L \to \{ordinary, urgent, committed\}$ is the function that assigns the type to each location of the automaton.

We can now define the semantics of UPTA in the way presented in [3]. A clock valuation is a function $val_c l : Cl \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. A variable valuation is a function $val_v : V \to \mathbb{Z} \cup BOOL$ from the set of variables to integers and booleans. Let $\mathbb{R}^{Cl}$ and $(\mathbb{Z} \cup BOOL)^V$ be the sets of all clock and variable valuations, respectively. The semantics of an UPTA is defined as a LTS $(\sum, init, \to)$, where $\sum \subseteq L \times \mathbb{R}^{Cl} \times (\mathbb{Z} \cup BOOL)^V$ is the set of states, the initial state $init = Init(cl, v)$ for all $cl \in Cl$ and for all $v \in V$, with $cl = 0$, and $\to \subseteq \sum \times \{\mathbb{R}_{\leq 0} \cup Act\} \times \sum$ is the transition relation such that:

$(l, val_{cl}, val_v) \to (l, val_{cl} + d, val_v)$ if $\forall d' : 0 \leq d' \leq d \Rightarrow val_{cl} + d \models Inv(l)$,
$(l, val_{cl}, val_v) \to (l', val'_{cl}, val'_v)$ if $\exists e = (l, act, G(cl, v), r, l') \in E$ i.e.
$val_{cl}, val_v \models G(cl, v), val'_{cl} = [re \to 0]val_{cl}$, and $val'_{cl}, val'_v \models Inv(l')$,

where for delay $d \in \mathbb{R}_{\geq 0}, val_{cl} + d$ maps each clock cl in Cl to the value $val_{cl} + d$, and $[re \to 0]val_{cl}$ denotes the clock valuation which maps (resets) each clock in re to 0 and agrees with val_{cl} over $Cl \backslash re$.

1.2. Test Generation for On-line Testing

Reactive on-line testing means that the tester program has to react to observed outputs of the IUT and to possible changes in the test goals on-the-fly. The rationale behind the reactive planning method proposed in [5] lies in combining computationally hard offline planning with time bounded online planning phases. Off-line phase is meant to shift the computationally hard planning as much as possible in the test preparation phase. Here the static analysis results of IUT model and the test goal are recorded in the format of compact planning rules that are easy to apply later in the on-line phase. The on-line planning rules synthesized must ensure close to optimal test runs and termination of the test case when a prescribed test purpose is satisfied.

The RPT synthesis algorithm introduced in [5] assumes that the IUT model is an output observable non-deterministic state machine ([6]). Test purpose (or goal) is a specific objective or a property of the IUT that the tester is set out to test. Test purpose is specified in terms of test coverage items. We focus on test purposes that can be defined as a set of boolean "*trap*" variables associated with the transitions of the IUT model ([7]). The goal of the tester is to drive the test so that all traps are visited at least once during the test run.

The tester synthesis method outputs tester model as UPTA where the rules for online planning are encoded in the transition guards called gain guards. The gain guard evaluates *true* or *false* at the time of execution of the tester determining if the transition can be taken from the current state or not. The value *true* means that taking the transition with the highest gain is the best possible choice to reach unvisited traps from current state. The decision rules for on-the-fly planning are derived by performing reachability analysis from the current state to all trap-equipped transitions by constructing the shortest path trees. Since at each test execution step only the guards associated with the outgoing transitions of the current state are evaluated, the number of guard conditions to be evaluated at one planning step is relatively small (equal to the location-local branching factor in the worst case). To implement such a gain guided model traversal, the gain guard is constructed using (model and goal specific) gain functions and the standard function max that return the maximum of those gain values that characterize alternative test paths.

Technically, the gain function of a transition returns a value that depends on the distance-weighted reachability of the unvisited traps from the given transition. The gain guard of the transition is *true* if and only if that transition is a prefix of the test sequence with highest gain among those that depart from the current state. If the gain functions of several enabled transitions evaluate to same maximum value the tester selects one of these transitions using either random selection or “least visited first” principle. Each transition in the model is considered to have a weight and the gain of test case is proportional to the length and the sum of weights of whole test sequence.

The RPT synthesis comprises three main steps (Figure 1):

1. extraction of the RPT control structure,
2. constructing gain guards,
3. reduction of gain guards according to the parameter *“planning horizon”* that defines the pruning depth of the planning tree.

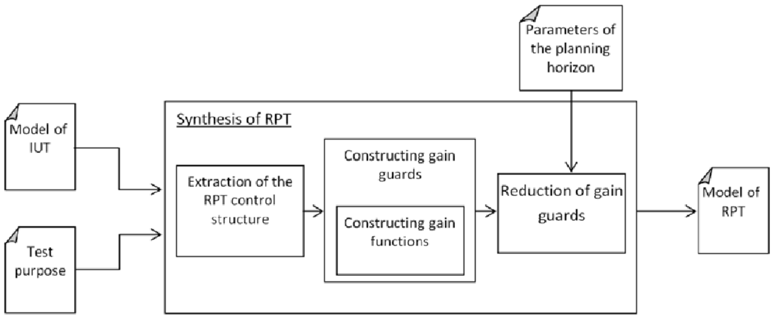

Parameters of the planning horizon
Model of IUT
Test purpose
Synthesis of RPT
Extraction of the RPT control structure
Constructing gain guards
Constructing gain functions
Reduction of gain guards
Model of RPT

Figure 1. RPT synthesis workflow

In the first step, the RPT synthesiser analyses the structure of the IUT model and generates the RPT control structure. In the second step, the synthesizer finds possibly successful IUT runs for reaching the test goal.

Last step of the synthesis reduces the gain functions pruning the planning tree up to some predefined depth that is given by parameter *“planning horizon”*. Since the RPT planning tree has the longest branch proportional to the length of Euler’s contour in the IUT model control graph the gain function’s recurrent structure may be very complex and for practical purposes needs to be bounded by some planning horizon. Traps being beyond the planning horizon still contribute in the gain function value but their distance is just ignored. Thus, for deep branches of planning tree the gain function returns an approximation of the gain value.

2. Correctness of IUT Models

2.1. Modelling Timing Aspects of IUT

For automated testing of input-output conformance of systems with time constraints we restrict ourselves with a subset of UPTA that simplifies IUT model construction. Namely, we use a subset where the data variables, their updates and transition guards on data variables are abstracted away. We use the clock variables only and the conditions expressed by clocks and synchronization labels. An elementary modelling pattern for representing IUT behaviour and timing constraints is Action pattern (or simply Action) depicted in Figure 2.

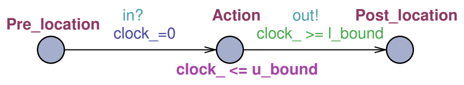


Figure 2. Elementary modelling fragment "Action"

An Action models a program fragment execution on a given level of abstraction as one atomic step. The Action is triggered by input event and it responds with output event within some bounded time interval (response time). The IUT input events (stimuli in testing context) are generated by Tester, and the output events (IUT responses) are to make the reactions of IUT observable to Tester. In UPTA, the interaction between IUT and Tester is modelled by synchronous channels that mediate input/output events. Receiving an input event from channel *in* is denoted by *in*? and sending an output event via channel *out* is denoted by *out*!.

The major timing constraint we represent in IUT model is *duration* of the Action. To make the specification of durations more realistic we represent it as a closed interval $[l\_bound, u\_bound]$, where *l_bound* denotes a lower bound and *u_bound* an upper bound of the interval. Duration interval $[l\_bound, u\_bound]$ can be expressed in UPTA as shown in Figure 2. Clock reset "$clock = 0$" on the edge "$Pre\_location \rightarrow Action$" makes the time constraint specification local to the Action and independent from current value at earlier execution steps. An invariant "$clock \leq u\_bound$" of location "$Action$" forces the Action to terminate latest at time instant *u_bound* after the clock reset and guard "$clock \geq l\_bound$" of the edge "$Action \rightarrow Post\_location$" defines the earliest time instant w.r.t. clock reset when the outgoing transition of Action can be executed.

From tester's point of view IUT has two types of locations: passive and active. In passive locations IUT is waiting for test stimuli and in active locations IUT chooses its next moves, i.e. presumably it can stay in that location as long as specified by location invariant. The location can be left when the guard of outgoing transition $Action \rightarrow Post\_location$ evaluates to $true$. In Figure 2, the locations $Pre\_location$ and $Post\_location$ are passive while $Action$ is an active location.

We compose IUT models from Action pattern using two types of composition rules: *sequential* and *alternative composition*.

Definition 1. Composition of Action patterns.

Let F_i and F_j be UPTA fragments composed of Action patterns (incl. elementary Action) with pre-locations l_i^{pre},l_j^{pre} and post-locations l_i^{post},l_j^{post} respectively, their composition is the union of elements of both fragments satisfying following conditions:

- sequential composition $F_i;F_j$ is UPTA fragment where $l_i^{post} = l_j^{pre}$;
- alternative composition $F_i + F_j$ is UPTA fragment where $l_i^{pre} = l_j^{pre}$ and $l_i^{post} = l_j^{post}$.

The test generation method we highlighted in Section 1.2 relies on the notion of well-formedness of the IUT model according to the following inductive definition.

Definition 2. Well-formedness (wf) of IUT models

- atomic Action pattern is well-formed;
- sequential composition of well-formed patterns is well-formed;
- alternative composition of well-formed patterns is well-formed if the output labels are distinguishable;

Proposition 1. Any UPTA model M with non-negative time constraints and synchronization labels that do not include state variables can be transformed to bi-similar to it well-formed representation $wf(M)$.

Note without the detailed proof that for those locations and edges of UPTA that do not match with the Definition 2, the well-formedness needs adding auxiliary pre-, and post-locations and ε-transition, that do not violate the i/o behaviour of original model. For representing internal actions that are not triggered by external events (their incoming edge is ε-labelled) we restrict the class of pre-locations with type "committed". In fact, the subclass of models transformable to well-formed is broader than given by Definition 2, including also UPTA that have data variable updates, but in general *wf*-form does not extend to models that include guards on data variables.

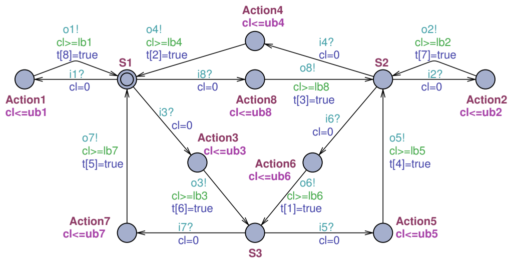


Figure 3. Simple example of well-formed IUT model

In the rest of paper, we assume for test generation that M^{IUT} is well-formed and denote this fact by $wf(M^{IUT})$. An example of such an IUT model we use throughout the paper is depicted in Figure 3.

2.2. *Correctness of IUT Models*

The test generation method introduced in [5] and developed further for EFSM models in [8] assumes that the IUT model is connected, input enabled, output observable and strongly responsive. In the following we demonstrate how the validity of these properties usually formulated for IOTS (Input-Output Transition System) models can be ensured for well-formed UPTA models (see Definition 2).

2.2.1. *Connected Control Structure and Output Observability*

We say that UPTA model is connected in the sense that there is an executable path from any location to any other location. Since the IUT model represents an open system that is interacting with its environment we need for verification by model checking a non-restrictive environment model. According to [9] such an environment model has the role of canonical tester. Canonical tester provides test stimuli and receives test responses in any possible order the IUT model can interact with its environment. A canonical tester can be easily generated for well-formed models according to the pattern depicted in Figure 4b (this is canonical tester for the model shown in Figure 3).

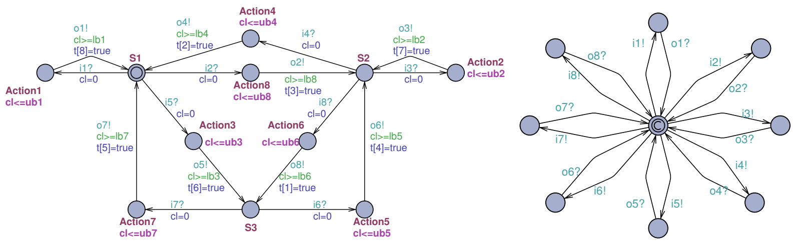


Figure 4. Synchronous parallel composition of a) IUT and b) canonical tester models

The canonical tester implements the "random walk" test strategy that is useful in endurance testing but it is very inefficient when functionally or structurally constrained test cases need to be generated for large systems.

Having synchronous parallel composition of IUT and the canonical tester (shown in Figure 4) the connectedness of IUT can be model checked with query (1) that expresses the absence of deadlocks in interactions between IUT and canonical tester.

$$A[]\, not\, deadlock \tag{1}$$

The output observability condition means that all state transitions of the IUT model are observable and identifiable by the outputs generated by these transitions. Observability is ensured by the definition of well-formedness of the IUT model where each input event and Action location is followed by the edge that generates a locally (w.r.t. source location) unique output event.

2.2.2. *Input Enabledness*

Input enabledness assumption means that blocking due to irrelevant test input during test execution is avoided. Naive way of implementing this assumption in IUT models presumes introducing self-looping transitions with input labels that are not represented on other transitions that share the same source state. That makes IUT modelling tedious and leads to the exponential increase of the M^{IUT} size. Alternatively, when relying on the notion of observational equivalence one can approximate the input enabledness in UPTA by exploiting the semantics of synchronizing channels and encoding input symbols as boolean variables $I_1 ... I_n \in \Sigma$. Then the pre-location of the Action pattern (see Figure 2) needs to be modified by applying the Transformation 1.

2.2.3. *Transformation 1*

- assume there are k outgoing edges from pre-location l_i^{pre} of $Action_i$, each of these transitions is labeled with some input $I_1 ... I_k \in \Sigma^i(Action_i) \subseteq \Sigma$;
- we add a self-looping edge $l_i^{pre} \rightarrow l_i^{pre}$ that models acceptance of all inputs in Σ except those in Σ^i. Because of that we specify the guard of edge $l_i^{pre} \rightarrow l_i^{pre}$ as boolean expression: $not(I_1 \vee ... \vee I_k)$.

Provided the outgoing branching factor $\mathscr{B}_i^{out}$ of l_i^{pre} is, as a rule, substantially smaller than $|\Sigma|$ we can save $|\Sigma| - \mathscr{B}_i^{out} - 1$ edges at each pre-location of Action patterns. Note that by wf-construction rules the number of pre-locations never exceeds the number of actions in the model. That is due to alternative composition that merges pre-locations of the composition. A fragment of alternative composition accepting inputs in Σ^i with described additional edge for accepting symbols in $\Sigma \setminus \Sigma^i(Action_i)$ is depicted in Figure 5 (time constraints are ignored here, I_1 and I_2 in the figure denote predicates $Input == i_1$ and $Input == i_2$ respectively).

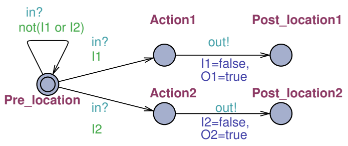


Figure 5. Input enabled fragment

2.2.4. Strong Responsiveness

Strong responsiveness (*SR*) means that there is no reachable livelock (a loop that includes only ε-transitions) in the IUT model M^{IUT}. In other words, M^{IUT} should always enter the quiescent state after finite number of steps. Since transforming M^{IUT} to $wf(M^{IUT})$ does not eliminate ε-transitions there is no guarantee that $wf(M^{IUT})$ is strongly responsive by default. To verify the *SR* propety of M^{IUT} we apply Algorithm 1.

2.2.5. Algorithm 1

1. According to the Action pattern of Figure 5 the information of i/o events is specified using synchronization channel *in* and a boolean variable that represents receiving an input symbol I_i. Since Uppaal model checker is state based we need recording the occurrence of input events in states. Therefore, the boolean variable representing an input event is kept true in the destination location of the edge that is labelled with given event and reset $false$ immediately after leaving this location. For same reason the ε-transitions are labelled with update $EPS = true$ and following output edge with update $EPS = false$.
2. Next, we reduce the model by removing all the edges and locations that are not involved in the traces of model checking query: $l_0 \models E\Box EPS$, where l_0 denotes initial location of M^{IUT}. The query checks if any ε-transition is reachable from l_0 (necessary condition for violating *SR*-property).
3. Further, we remove all non ε-transitions and locations that remain isolated thereafter.
4. Remove recursively all locations that do not have incoming edges (their outgoing edges will be deleted with them).
5. After reaching the fixed point of recursion of step 4 we check if the remaining part of model is empty. If yes then we can conclude that M^{IUT} is strongly responsive, otherwise it is not.

It is easy to show that all steps except step 2 are of linear complexity in the size of the M^{IUT}.

3. Correctness of RPT Tests

3.1. Functional Correctness of Generated Tests

The tester program generated based on IUT model can be characterized using some test coverage criteria it is designed for. As shown in Section 1.2, the RPT generating algorithm is aimed at structural coverage of IUT model elements and can be expressed by means of boolean "trap" variables. To recall, the traps are assignment expressions of boolean trap variables and the valuation of traps indicates the status of the test run. For instance, one can observe if the edges labeled with them are already covered or not in the course of test run. Thus, the relevant correctness criterion for the tester generated is its ability to cover traps.

Definition 3. Coverage correctness of the test.

We say that the RPT tester is coverage correct if the test run covers all the transitions that are labelled with traps in IUT model.

Definition 4. Optimality of the test.

We say that the test is length (time) optimal if there is no shorter (accordingly faster) test runs among all those being coverage correct.

We can show that the RPT method generates tests that are coverage correct (and in general, close to optimal) by construction, if the planning horizon of gain function is greater or equal to the depth of reduced reachability tree of M^{IUT}. Though, the practical limit of planning depth is set by Uppaal tool where the largest integer value of type '*long*' is 2^{31}. That allows distinctive encoding of gain function co-domain for test paths up to depth 31. It means that if the IUT is fully connected and deterministic RPT provides a test path that covers all traps length-optimally. In non-deterministic case it provides the best strategy against any legal strategy the IUT chooses (legal in this context means that any behaviour of IUT either conforms to its specification or is detectably violating it).

While the reachability tree exceeds given by the horizon depth limit the gain function becomes stochastic (insensible to reachability tree structure deeper than the horizon). It is distinctive on the number of deeper traps only, but it is not distinctive on their co-reachability. Even though, the planning method with cross horizon depth has shown to be statistically efficient by providing close to optimal test paths in large examples there is threat of choosing infeasible paths if the model is not well-formed and/or not connected.

Instead of going into details of the proof (by structural induction) of RPT tester generation correctness and optimality we provide ad-hoc verification procedure in terms of model checking queries and model construction constraints.

Direct way of verifying the coverage correctness of the tester is to run a model checking procedure with query:

$$A \diamond \forall (i : int[1,n])\, t[i]\,, \qquad (2)$$

where $t[i]$ denotes i-th element of the array of traps. The model the query is running on is synchronous parallel composition of IUT and Tester automata. For instance, the RPT automation for IUT modelled in Figure 3 is depicted in Figure 6.

3.2. Invariance of Tests with Respect to Changing Time Constraints of IUT

In section 2.2 the coverage correctness of RPT tests was discussed without explicit reference to M^{IUT} time constraints. The length-optimality of test sequences can be proven

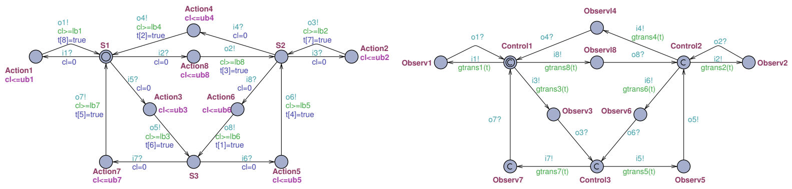


Figure 6. Synchronous parallel composition of IUT and RPT models

in Uppaal when for each $Action_i$ both the duration lower and upper bounds lb_i and ub_i equal to one, i.e., $lb_i = ub_i = 1$ for all $i \in 1,...,|Action|$. Then the length of the test sequence and its duration in time are numerically equal. For instance, having some integer valued (time horizon) parameter TH as an upper bound to the test sequence length the following model checking query proves the coverage of n traps with a test sequence of length at most TH stimuli and responses:

$$A \diamond \forall (i : int[1,n])\, t[i] \wedge TimePass \leq TH \tag{3}$$

where $TimePass$ is Uppaal clock that represents global time of the model.

Generalizing this result for IUT models with arbitrary time constraints we assume that all edges of M^{IUT} are attributed with time constraints as described in Section 2.1. Since not all the transitions of model M^{IUT} need to be labelled with traps (and thus covered by test) we apply compaction procedure to M^{IUT} to abstract away from the excess of information and derive precise estimates of test duration lower and upper bounds. With compaction we aggregate consecutive trapless transitions with one trap-labelled transition the trapless ones are neighbours to. Now, the aggregate Action becomes like atomic Action of Figure 2 that copies the trap of the only trap labelled transition included in the aggregate. The first transition of the aggregate contributes its input event and the last transition its output event. The other I/O events of the aggregate will be hidden because all internal transitions and locations are substituted with one aggregate location we call composite Action. Further, we compute the lower and upper bounds for the composite action. The lower bound is the sum of lower bounds of the shortest path in the aggregate and the upper bound is the sum of upper bounds of the longest path of the aggregate plus the longest upper bound (the later is needed to compute the test termination condition). After compaction of deterministic and timed IUT model it can be proved that the duration TH of a coverage correct tests have length that satisfies following condition:

$$\sum_i lb_i \leq TH \leq \sum_i ub_i + \max_i(ub_i), \tag{4}$$

where index i ranges from 1 to n (n - number of traps in M^{IUT}).

In case of non-deterministic IUT models, for showing length- and time-optimality of generated tests the bounded fairness of M^{IUT} needs to be assumed. We say that a model M is $k-fair$ iff the difference in the number of executions of alternative transitions of non-deterministic choices never exceeds the bound k. This assumption excludes unbounded "starvation" and "conspiracy" behaviour in non-deterministic models. During the test run our test execution environment dTron [10] is monitoring k-fairness and reporting error message "violation of IUT k-fairness assumption" when this constraint is broken. Due to k-fairness monitoring by dTron the safe estimate of the test length upper bound in case of non-deterministic models can be found for the worst case by multiplying the deterministic upper bound by factor k. The lower bound still remains $\sum_i lb_i$.

Proposition 2. Assuming a trap labelled UPTA model M^{IUT} is well-formed in the sense of Definition 2 and compactified, the RPT that is generated based on M^{IUT} remains invariant with respect to variations of the time constraints specified in M^{IUT}.

The practical implication of Proposition 2 is that a RPT once generated for a timed trap labeled UPTA model M^{IUT}, one can use it for any syntactically and semantically feasible modification of M^{IUT} where only timing parameters and initial values of traps have been changed. Invariance does not extend to structural changes of M^{IUT}.

Due to the limited space we sketch the proof in two steps by showing that (i) the control decisions of M^{RPT} do not depend on the timing of M^{IUT} and (ii) the M^{RPT} behaviour does not influence the timing on controllable transitions of M^{IUT}.

(i) The behaviour of M^{RPT} depends on the gain guards of its controllable edges and responses (output events) of M^{IUT}, not on the time instances when these responses are generated. Same applies to the gain guards. They are boolean functions defined on the structure of M^{IUT} and the valuation vector of traps. Thus the timing constraints specified in M^{IUT} do not influence the behaviour of M^{RPT}.

(ii) In the synchronous parallel composition $M^{IUT} ||_{sync} M^{RPT}$ the actions of M^{IUT} and M^{RPT} take the effect over progress of time alternatively. Though the communication of input and output events is synchronous, it is due to the semantics of UPTA, that execution of transitions is instantaneous, and does not pose any constraint on the delay between earlier or later event. Since the planning time of M^{RPT} is assumed to be negligible comparing to the response time of M^{IUT} we model the control locations in M^{RPT} always as committed locations (denoted by "c" in Figure 6) and there is no additional waiting in obsevation locations of M^{RPT} either. Thus, M^{RPT} does not set any restriction to the time invariants $inv(Action_i)$and transition guards $grd(Action_i \rightarrow PostLocation_i)$ of M^{IUT} actions.

4. Test Execution Environment dTron

Uppaal TRON is a testing tool, based on Uppaal [3] engine, suited for black-box conformance testing of timed systems [11]. dTron [12] extends this enabling distributed execution. It incorporates Network Time Protocol (NTP) based real-time clock corrections to give a global timestamp (t_1) to events at IUT adapter(s). These events are then globally serialized and published for other subscribers with a Spread toolkit [13]. Subscribers can be other SUT adapters, as well as dTron instances. NTP based global time aware subscribers also timestamp the event received message (t_2) to compute and possibly compensate for the overhead time it takes for messaging overhead $\Delta = t_2 - t_1$.

Δ is essential in real-timed executions to compensate for messaging delays that may lead to false-negative non-conformance results for the test-runs. Messaging overhead caused by elongated event timings may also result in messages published in on order, but revived by subscribers in another. Δ can then also be used to re-order the messages in a given buffered time-window t_Δ. Due to the online monitoring capability dTron supports the functionality of evaluating upper and lower bounds of message propagation delays by allowing the inspection of message timings. While having such a realistic network latency monitoring capability in dTron our test correctness verification workflow takes into account theses delays. For verfication of the deployed test configuration we make corresponding time parameter adjustments in the IUT model. By Proposition 2 the RPT tester generated is invariant to time parameter variations. Thus final verification against the query 3 is proving that the test is feasible as well in the presence of realistic configuration constraints of the testing framework dTron.

5. Web Testing Case-study

We describe street light control system (SLCS) to show the applicability of the proposed testing workflow. The SLCS has a central server and multiple controllers each controlling one or more streetlight. The controllers have programmable high-power relays (contactors) to manipulate the actual lights, but also have various sensor and communication extensions to provide supplementary capabilities like dimming and following more complex lighting programs.

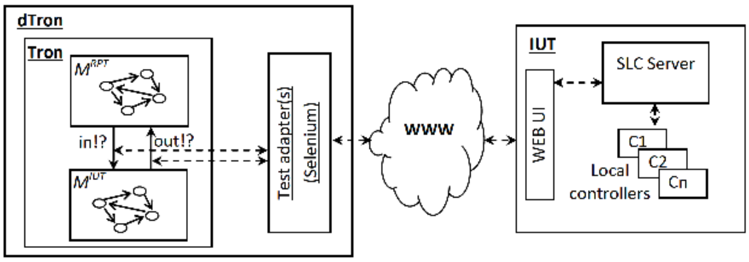


Figure 7. Street light control system test architecture

Light-controllers have minimal memory and do not persistently store their state in the memory. They poll the central server to retrieve their designated state information. This state information is stored in the array of bits, each bit denoting a specific parameter value for the controller. Controller polls the server and the server responds whether it has new state info for the controller. If this is the case, the information is provided with the response. The server holds the state information for each controller. This information can be manipulated by users via an Internet web user interface (UI). Figure 7 shows an abstract view of test architecture. The test purpose is to test if when a user has logged in and tries to turn on a light using the UI, the light will eventually get lit and that is reported back with message lights on.

Figure 8 shows an extract of IUT model M^{IUT} and generated tester M^{RPT}. The test adapters shown in Figure 7 interface symbolic interactions specified by channels in the model with real interface of IUT. These channels are distinguished by name convention. We use names *in* and *out* in the model and they are intercepted by dTron and executed by *adapters*. Adapters translate synchronizations in the model in to actions against the actual system and feed information back to the model.

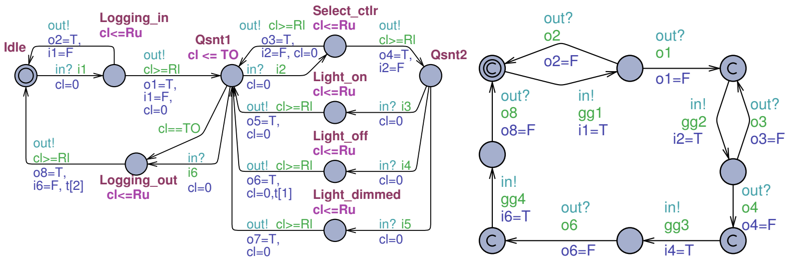


Figure 8. IUT and RPT models

Table 1. Tester input and output variables.

Input		**Output**	
Variable	**Meaning**	**Variable**	**Meaning**
i1	login	o1	login sucessful
i2	select controller (for setting)	o2	login failed
i3	set light on	o3	empty selection of controllers
i4	set light off	o4	mode setting menu for chosen controllers
i5	dimming the light	o5	status report “light on”
i6	logout	o6	status report “light of”
		o7	status report “light dimmed”
		o8	log out completed

Table 2. Pre-execution correctness checks of tests.

Correctness condition	**Verification method**
Output observability of M^{IUT}	Static analysis of test stimulus - response pairs
Connected control structure of M^{IUT}	Generating canonical tester and running query 1
Input enabledness of M^{IUT}	Transformation 1 (see Section 2.2)
Strong responsivness of M^{IUT}	Algorithm 1 (see Section 2.2)
Coverage correctness of M^{RPT}	Model checking query 2
Time-bound checks of tests	Compaction procedure (Section 3.2), calculate 4

The tester is controlling that the test run will cover traps $t[1]$ and $t[2]$. The inputs and outputs of M^{IUT} are explained in the table 1.

The timing constraints of IUT are specified in M^{IUT} as follows:

- TO denotes the time-out to log off after being logged in if there is no activity over UI during TO time units
- All actions controllable and observable over UI have pre-specified duration interval $[Rl, Ru]$. If the responses to IUT inputs do not conform with given interval the timing conformance test fail is reported. Implicitly $[Rl, Ru]$ includes also parameter Δ. The estimate $\widehat{\Delta}$ of Δ is generated by dTron as the result of monitoring the traffic logs at the planned test interface

Before running the executable test dTron performs a sequence of test model verifications. Table 2 illustrates the verification tasks available with current version of dTron.

6. Conclusion

We have proposed a MBT testing workflow that incorporates steps of IUT modelling, test specification, generation, and execution that are alternating with their correctness verification steps. The online testing approach of timed systems proposed relies on Reactive Planning Tester (RPT) synthesis algorithm and distributed test execution environment dTron. As shown in the paper the behaviour of generated RPT tester model does not set extra timing constraints to controllable input/output of IUT and the on-line decisions of the tester do not depend on the timing of IUT. dTron provides support to estimate time delays in real test configuration and allows to take them into account while verifying the test correctness properties with real environment delay constraints. This is a first practical step towards provably correct automated test generation for Δ-testing outlined as a new MBT challenge in [4].

Acknowledgements

This research is partially supported by ELIKO and the European Union through the European Regional Development Fund and by the Tiger University Program of the Information Technology Foundation for Education.

References

[1] Tretmans, Jan. Test Generation with Inputs, Outputs and Repetitive Quiescence In: Software - Concepts and Tools, 1996, 17 (3), 103 -120.

[2] Roberto Segala. Quiescence, Fairness, Testing, and the Notion of Implementation. In: Inf. Comput., 1997, 138 (2), 194-210.

[3] Behrmann, G., David, A., Larsen, K. A tutorial on uppaal. In: Bernardo, M., Corradini, F. (ed.) *Formal Methods for the Design of Real-Time Systems*. Springer, Berlin Heidelberg, 2004. 200 – 236.

[4] Alexandre David, Kim G. Larsen, Marius Mikucionis, Omer L. Nguena Timo, Antoine Rollet. Remote Testing of Timed Specifications. Springer, 2013, 65-81. (Lecture Note in Computer Science, 8254).

[5] Vain, J., Raiend, K., Kull, A., and Ernits, J. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In: 22nd IEEE/ACM Int. Conf. on Automated Software Engineering. ACM Press, 2007, 363 – 372.

[6] Luo, G., von Bochmann, G., & Petrenko, A. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. IEEE Transactions in Software Engineering, 1994, 20 (2), 149 – 162.

[7] Hamon, G., de Moura, L., & Rushby, J. Generating efficient test sets with a model checker. In: SEFM 2004: Proceedings of the Software Engineering and Formal Methods, Second International Conference. IEEE Computer Society, 2004, 261 – 270.

[8] Kääramees, M. A Symbolic Approach to Model-based Online Testing [dissertation]. Tallinn: TUT Press, 2012.

[9] Brinksma, Ed., Alderen, R., Lngerak, R., Lagemaat, J.d.v., Tretmans, J., A Formal approach to conformance testing. 2nd Workshop on Protocol Test Systems. Berlin, October 1989.

[10] A.Anier, J.Vain. Model based continual planning and control for assistive robots. HealthInf 2012. Vilamoura, Portugal. 1-4 Feb, 2012.

[11] UPPAAL TRON. [WWW] http://people.cs.aau.dk/˜marius/tron/ (accessed 20.04.2014)

[12] DTRON home page. [WWW] http://dijkstra.cs.ttu.ee/˜aivo/dtron/ (accessed 20.04.2014)

[13] The spread toolkit. [WWW] http://spread.org/ (accessed 20.04.2014)

References

[1] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing,” 2006.

[2] M. S. Herring, B. D. Owens, N. Leveson, M. Ingham, and K. A. Weiss, “A Safety-driven, Model-based System Engineering Methodology, Part I,” tech. rep., MIT Technical Report, December 2007.< http://sunnyday. mit. edu/papers. html# system-safety, 2007.

[3] M. Mikucionis, “UPPAAL TRON: Testing Real-time systems Online,” 2007.

[4] C. Baier, J.-P. Katoen, and others, *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[5] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on uppaal,” in *Formal methods for the design of real-time systems*, pp. 200–236, Springer, 2004.

[6] “Home · eishay/jvm-serializers wiki · GitHub.”

[7] K. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a nutshell,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, p. 134–152, 1997.

[8] F. Miyawaki, K. Masamune, S. Suzuki, K. Yoshimitsu, and J. Vain, “Scrub nurse robot system-intraoperative motion analysis of a scrub nurse and timed-automata-based model for surgery,” *Industrial Electronics, IEEE Transactions on*, vol. 52, p. 1227 – 1235, Oct. 2005.

[9] S. Khaitan and J. McCalley, “Design Techniques and Applications of Cyberphysical Systems: A Survey,” *Systems Journal, IEEE*, vol. 9, pp. 350–365, June 2015.

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

[11] E. Lee and others, “Cyber physical systems: Design challenges,” in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pp. 363–369, IEEE, 2008.

[12] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, “Model-integrated development of embedded software,” *Proceedings of the IEEE*, vol. 91, no. 1, p. 145–164, 2003.

[13] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, "Contracts for system design," 2012.

[14] A. Schiele, "Towards a unified model-based control framework for rapid (space) robotics developments," May 2013.

[15] W. Emmerich, "Distributed system principles," *Course notes, University College London*, 1997.

[16] M. Schleipen, "OPC UA supporting the automated engineering of production monitoring and control systems," in *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pp. 640–647, IEEE, 2008.

[17] K. Kim, "Real-time software framework for distributed control systems," *system*, vol. 3, no. 6, p. 7–8.

[18] J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Distributed real-time software for cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 45–59, 2012.

[19] T. Hoare, "The ideal of verified software," in *Computer Aided Verification*, p. 5–16, 2006.

[20] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977.

[21] "projects/jpf-symbc – Java Path Finder."

[22] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-based testing for embedded systems.* CRC press, 2011.

[23] P. Lasota, S. Nikolaidis, and J. Shah, "Developing an Adaptive Robotic Assistant for Close Proximity Human–Robot Collaboration in Space," _ *AIAA Infotech@ Aerospace* _, 2013.

[24] W. Hara, S. G. Soltys, and I. C. Gibbs, "CyberKnife® Robotic Radiosurgery system for tumor treatment," 2007.

[25] "Model-based design," July 2015. Page Version ID: 671636882.

[26] G. Nicolescu and P. J. Mosterman, *Model-Based Design for Embedded Systems.* CRC Press, 2009.

[27] R. Jeffords, 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, 2009.

[28] J.-R. Abrial, "Formal methods in industry: achievements, problems, future," in *Proceedings of the 28th international conference on Software engineering*, pp. 761–768, ACM, 2006.

[29] T. B. Sheridan, *Telerobotics, automation, and human supervisory control.* MIT press, 1992.

[30] J. Vain, F. Miyawaki, S. Nõmm, T. Totskaya, and A. Anier, "Human-robot interaction learning using timed automata," in *ICCAS-SICE, 2009*, pp. 2037–2042, IEEE, 2009.

[31] D. K. Kaynar and N. Lynch, "Decomposing verification of timed I/O automata," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pp. 84–101, Springer, 2004.

[32] O. Grinchtein, B. Jonsson, and M. Leucker, "Learning of event-recording automata," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pp. 379–395, Springer, 2004.

[33] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using UPPAAL," *Formal methods and testing*, p. 77–117, 2008.

[34] A. David, K. G. Larsen, M. Mikučionis, O. L. N. Timo, and A. Rollet, "Remote testing of timed specifications," in *Testing Software and Systems*, pp. 65–81, Springer, 2013.

[35] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2nd ed., 2002.

[36] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, "Modeling Time in Computing: A Taxonomy and a Comparative Survey," *ACM Comput. Surv.*, vol. 42, pp. 6:1–6:59, Mar. 2010.

[37] C. Baier, J.-P. Katoen, and others, *Principles of model checking*, vol. 26202649. MIT press Cambridge, 2008.

[38] M. Felder and A. Morzenti, "Validating Real-time Systems by History-checking TRIO Specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, pp. 308–339, Oct. 1994.

[39] N. Wirth, "Toward a Discipline of Real-time Programming," *Commun. ACM*, vol. 20, pp. 577–583, Aug. 1977.

[40] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety.* Springer Science & Business Media, 2012.

[41] R. Alur and T. Henzinger, "Logics and models of real time: A survey," in *Real-Time: Theory in Practice* (J. de Bakker, C. Huizing, W. de Roever, and G. Rozenberg, eds.), vol. 600 of *Lecture Notes in Computer Science*, pp. 74–106, Springer Berlin Heidelberg, 1992.

[42] A. Burns and G. Baxter, "Time bands in systems structure," in *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective* (D. Besnard, C. Gacek, and C. Jones, eds.), pp. 74–88, Springer London, 2006.

[43] C. Furia and M. Rossi, "Integrating Discrete- and Continuous-Time Metric Temporal Logics Through Sampling," in *Formal Modeling and Analysis of Timed Systems* (E. Asarin and P. Bouyer, eds.), vol. 4202 of *Lecture Notes in Computer Science*, pp. 215–229, Springer Berlin Heidelberg, 2006.

[44] C. Furia, M. Pradella, and M. Rossi, "Automated Verification of Dense-Time MTL Specifications Via Discrete-Time Approximation," in *FM 2008: Formal Methods* (J. Cuellar, T. Maibaum, and K. Sere, eds.), vol. 5014 of *Lecture Notes in Computer Science*, pp. 132–147, Springer Berlin Heidelberg, 2008.

[45] N. Francez, *Fairness. Texts and monographs in computer science.* Springer-Verlag, 1986.

[46] G. Booch, D. L. Bryan, and C. G. Petersen, *Software engineering with Ada.* Addison-Wesley Professional, 1994.

[47] E. Mendelson, *Introduction to mathematical logic.* CRC press, 2009.

[48] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and computation*, vol. 104, no. 1, pp. 2–34, 1993.

[49] M. Timmer, H. Brinksma, and M. Stoelinga, "Model-based testing," 2011.

[50] G. J. Tretmans, "A formal approach to conformance testing," 1992.

[51] M. Van Der Bijl, A. Rensink, and J. Tretmans, "Compositional testing with ioco," in *Formal Approaches to Software Testing*, pp. 86–100, Springer, 2003.

[52] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, p. 1–38, Springer, 2008.

[53] M. Timmer, E. Brinksma, and M. Stoelinga, "Model-Based Testing," in *Software and Systems Safety - Specification and Verification*, pp. 1–32, 2011.

[54] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software—Concepts and Tools*, no. TR-CTIT-96-26, 1996.

[55] R. De Nicola and M. C. Hennessy, "Testing equivalences for processes," *Theoretical Computer Science*, vol. 34, no. 1, p. 83–133, 1984.

[56] E. Brinksma, *A theory for the derivation of tests.* No. VIII in Protocol Specification, Testing, and Verification, University of Twente, Department of Computer Science, 1988.

[57] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," 1988.

[58] R. Segala, "Quiescence, fairness, testing, and the notion of implementation," in *CONCUR'93*, pp. 324–338, Springer, 1993.

[59] J. Vain, K. Raiend, A. Kull, and J. P. Ernits, "Synthesis of test purpose directed reactive planning tester for nondeterministic systems," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, (New York, NY, USA), p. 363–372, ACM, 2007.

[60] M. Kääramees, *A Symbolic Approach to Model-based Online Testing.* PhD thesis, Tallinn University of Technology, Tallinn, Nov. 2012.

[61] E. Brinksma, "Formal approach to conformance testing," in *Proc. Int. Workshop on Protocol Test Systems*, pp. 311–325, North-Holland, 1989.

[62] G. Hamon, L. De Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pp. 261–270, IEEE, 2004.

[63] A. Anier and J. Vain, "Model based continual planning and control for assistive robots.," *HealthInf 2012*, no. Proceedings of the International Conference on Health Informatics, pp. 382–385, 2012.

[64] J. Vain, A. Anier, and E. Halling, "Provably Correct Test Development for Timed Systems," in *Databases and Information Systems VIII: Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014*, vol. 270, p. 289, IOS Press, 2014.

[65] A. C. Dias 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*, pp. 31–36, ACM, 2007.

[66] S. Nomm, E. Petlenkov, J. Vain, J. Belikov, F. Miyawaki, and K. Yoshimitsu, "Recognition of the Surgeon's Motions During Endoscopic Operation by Statistics Based Algorithm and N," in *World Congress*, vol. 17, pp. 14773–14778, 2008.

[67] J. Jakubiak, S. Nõmm, J. Vain, and F. Miyawaki, "Polynomial based approach in analysis and detection of surgeon's motions," in *Control, Automation, Robotics and Vision, 2008. ICARCV 2008. 10th International Conference on*, pp. 611–616, IEEE, 2008.

[68] E. Petlenkov, S. Nõmm, J. Vain, and F. Miyawaki, "Application of self organizing Kohonen map to detection of surgeon motions during endoscopic surgery," in *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, pp. 2806–2811, IEEE, 2008.

[69] J. Shaw, "Web application performance testing—a case study of an on-line learning application," *BT Technology Journal*, vol. 18, no. 2, pp. 79–86, 2000.

[70] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, "Model-Based Performance Testing of Web Services Using Probabilistic Timed Automata," in *Proceedings of the 9th International Conference on Web Information Systems and Technologies* (K.-H. Krempels and A. Stocker, eds.), pp. 99–104, Webist, 2013.

[71] N. Francez, C. Hoare, D. J. Lehmann, and W. P. De Roever, "Semantics of nondeterminism, concurrency, and communication," *Journal of Computer and System Sciences*, vol. 19, no. 3, pp. 290–308, 1979.

[72] R. Morin, "Semantics of deterministic shared-memory systems," in *CONCUR 2008-Concurrency Theory*, pp. 36–51, Springer, 2008.

[73] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie, "Efficient Learning of Typical Finite Automata from Random Walks," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, (New York, NY, USA), pp. 315–324, ACM, 1993.

[74] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[75] M. J. Kearns and U. V. Vazirani, *An introduction to computational learning theory.* MIT press, 1994.

[76] M. Ade, P. GHRIET, P. Deshmukh, and A. SCOE&T, "Methods for incremental learning: A survey," *International Journal of Data Mining & Knowledge Management Process*, vol. 3, no. 4, pp. 119–125, 2013.

[77] "UPPAAL TRON."

[78] "Maven - welcome to apache maven."

[79] T. Parr, *The definitive ANTLR reference: Building domain-specific languages.* Pragmatic Bookshelf, 2007.

[80] "The spread toolkit."

[81] R. M. Hierons, "Oracles for distributed testing," *Software Engineering, IEEE Transactions on*, vol. 38, no. 3, p. 629–641, 2012.

[82] H. Zimmermann, "OSI reference model–The ISO model of architecture for open systems interconnection," *Communications, IEEE Transactions on*, vol. 28, no. 4, p. 425–432, 1980.

[83] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre, "D-ITG distributed internet traffic generator," in *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, p. 316–317, 2004.

[84] J. Nagle, "Congestion control in IP/TCP internetworks," 1984.

[85] A. Anier, "DTRON home - lightcontroller case study," May 2014.

[86] T. A. Henzinger, R. Majumdar, and J.-F. Raskin, "A classification of symbolic transition systems," *ACM Transactions on Computational Logic (TOCL)*, vol. 6, no. 1, pp. 1–32, 2005.

[87] S. Quinton and S. Graf, "Contract-based verification of hierarchical systems of components," in *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pp. 377–381, IEEE, 2008.

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.

111. **Syed Saif Abrar**. Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC. 2016.

112. **Arvo Kaldmäe**. Advanced Design of Nonlinear Discrete-time and Delayed Systems. 2016.

113. **Mairo Leier**. Scalable Open Platform for Reliable Medical Sensorics. 2016.

114. **Georgios Giannoukos**. Mathematical and physical modelling of dynamic electrical impedance. 2016.