TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
17/2019

Scenario Oriented Model-Based Testing

EVELIN HALLING

TTÜ
PRESS

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

The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Informatics on April 2, 2019.

Defence of the thesis: May 10, 2019, Tallinn

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

Evelin Halling ______________________
signature

European Union
European Regional
Development Fund

Investing
in your future

Copyright: Evelin Halling, 2019
ISSN 2585-6898 (publication)
ISBN 978-9949-83-416-7 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-417-4 (PDF)

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
17/2019

Stsenaariumjuhitud mudelipõhine testimine

EVELIN HALLING

TTÜ KIRJASTUS

Contents

List of Publications

Publication I: E. Halling, J. Vain, A. Boyarchuk, and O. Illiashenko. Test scenario specification language for model-based testing. *International Journal of Computing*, 2019

Publication II: J. Vain, A. Anier, and E. Halling. Provably correct test development for timed systems. In H. Haav, A. Kalja, and T. Robal, editors, *Databases and Information Systems VIII - Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014, 8-11 June 2014, Tallinn, Estonia*, volume 270 of *Frontiers in Artificial Intelligence and Applications*, pages 289–302. IOS Press, 2014

Publication III: J. P. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ROS packages: A mobile robot case study. In *2015 European Conference on Mobile Robots, ECMR 2015, Lincoln, United Kingdom, September 2-4, 2015*, pages 1–7. IEEE, 2015

Publication IV: J. Vain, E. Halling, G. Kanter, A. Anier, and D. Pal. Model-based testing of real-time distributed systems. In G. Arnicans, V. Arnicane, J. Borzovs, and L. Niedrite, editors, *Databases and Information Systems - 12th International Baltic Conference, DB&IS 2016, Riga, Latvia, July 4-6, 2016, Proceedings*, volume 615 of *Communications in Computer and Information Science*, pages 272–286. Springer, 2016

Publication V: J. Vain and E. Halling. Constraint-based testing scenario description language. *Proceedings of 13th Biennial Baltic Electronics Conference, BEC 2012*, IEEE: 89–92, 2012

Author’s Contributions to the Publications

I In Publication I, I was the main author, carried out the simulations and the analysis of the results, prepared the figures, and wrote the manuscript.

II In Publication II, I conducted the experiments and simulations, analysed the results, prepared the figures, and wrote the manuscript.

III In Publication III, I conducted the experiments and simulations, analysed the results, prepared the figures, and wrote the manuscript.

IV In Publication IV, I prepared the example models and conducted validation experiments.

V In Publication V, I conducted the experiments and simulations, prepared the figures, and wrote the manuscript.

Overview

In this thesis, the research is focused on model-based testing, specifically, on the test purpose specification and test generation techniques to address the test coverage and faults back-traceability problems.

The structure of the thesis is the following:

- Chapter 1 gives the motivation why this research is needed in the critical systems software engineering practice, defines the research problems to be solved and their scope, outlines the related work, and introduces the methodology, problem specific hypothesis and goals to be addressed in the thesis.
- Chapter 2 presents the theoretical and methodological preliminaries the results of the thesis are built upon. The foundations of the thesis are model-based conformance testing theory, Uppaal timed automata and TCTL model checking theory.
- Chapter 3 introduces the provably correct test development workflow and verification conditions necessary to assure the correctness of development increments. The test purpose descriptions and test generation correctness are verified with respect to these conditions.
- Chapter 4 defines the test purpose specification language TDL^{TP}, its syntax and semantics.
- In Chapter 5, the practical usability of the test purpose specification language TDL^{TP} and provably correct test development method are validated on the TUT100 satellite software case study.

The results of the thesis are concluded and future research perspectives outlined in the thesis conclusion.

Abbreviations

CCDL	Check Case Definition Language
CPS	Cyber/Physical Systems
DTRON	Distributed Testing Realtime systems Online
EFSM	Extended Finite State Machine
IOCO	Input-Output Conformance Relation
IOTS	Input-Output Transition System
ISO	International Organization for Standardization
ITU	International Telecommunication Union
IUT	Implementation Under Test
LEOP	Launch and Early Orbit Phase
LTL	Linear Temporal Logic
LTS	Labelled Transition System
MBT	Model-Based Testing
MCS	Mission Critical Systems
MSC	Message Sequence Chart
NTA	Network of Timed Automata
OTX	Open Text Sequence Exchange Format
RPT	Reactive Planning Tester
RT-IOCO	Real-time Input-Output Conformance Relation
SUT	System Under Test
SysML	Systems Modelling Language
TA	Timed Automata
TCTL	Timed Computation Tree Logic
TCS	Time Critical Systems
TDL	Test Description Language
TIOTS	Timed Input-Output Transition System
TLTS	Timed Labelled Transition System
TPLan	Test Purpose Language
UML	Unified Modelling Language
V&V	Verification and Validation

1 INTRODUCTION

In this chapter the motivation for studying novel methods of model-based testing is given. The scope of the thesis is determined from three perspectives: application domain, testing technology, and formal framework for test purpose specification, test generation and execution. The goal and main results of the thesis are positioned with respect to the related work of other authors in the field. The chapter concludes by outlining the main hypothesis, methodology, and contribution of the thesis.

1.1 Motivation

Mission Critical Systems[1] (MCS) are systems whose failure might cause catastrophic consequences, such as someone dying, damage to property, severe financial losses, damage to national security and others. There are several well-known failure cases, such as TheracTwentyFive[2] radiation therapy machine malfunctioning caused by undetected software fault, namely race condition. Six known accidents have been listed due to the overdose, several of them fatal. Similarly, Patriot Missile Failure caused by software error in the system's clock, resulted in an accumulated clock drift that led to the improper reaction to Scud attack and death of twenty eight soldiers and over hundred other casualties. Mars Climate Orbiter Crash (NASA lost $125-million) was caused by misinterpreted requirements in software implementation, namely, the thrust impulse command for the thrusters was produced in imperial units, instead of metric units.

In these and many other examples the mission criticality is mixed also with time criticality. The *Time Critical Systems*[3] (TCS) fail if the timing deadlines are not met by the system. So, a well-designed MCS, even in case of unavoidable system's failures, if properly predicted, timely detected and recovered, should be able to operate under severe exploitation conditions without catastrophic consequences.

Detection of software bugs, especially those deeply nested in software loops which manifest sporadically as wrong timing in complex TCS, is a real challenge for current MCS and TCS software engineering methods. The methods of risk mitigation, in particular the provably correct software synthesis, formal verification as well as model-based testing, are powerful but time-wise and computationally expensive which limits their wider application. In [26], it is stated that software verification and testing constitutes up to 50 percent (or even more in mission critical applications) of the total development costs of software.

Authors of [34] report that the root causes of 56 percent of all defects identified in software projects are introduced in the requirements phase. They profess that low software quality is mainly due to the *problematical test coverage and incorrect requirements*. In addition, 50 percent of incorrect requirements are caused by incomplete specification and another 50 percent by *unclear* and *ambiguous* requirements.

Another research report [5] outlines the application domains and development phases with highest risk of failure or delay. In *automotive* and *medical* domain the *system integration level test and verification* cause project delays respectively in 63 percent and in 66,7 percent of the cases. An extreme is the medical domain where system's *middleware development and test* has caused delays in about 75 percent of studied cases. Since both automotive and medical domains are often mission and time critical it gives indication that *the software integration level test* and *verification* are the *main bottlenecks*, thus new

[1]`http://wiki.c2.com/MissionCritical`
[2]`http://wiki.c2.com/TheracTwentyFive`
[3]`http://wiki.c2.com/TimeCritical`

verification and test development methods and their tooling are of key importance. This way, any increase in productivity of testing methods and tools would have strong impact on the productivity of the whole development process and on MCS software assurance in general.

1.2 The scope of the thesis

In this thesis, the research is focused on model-based testing, specifically, on the test purpose specification and test generation techniques to address the test coverage and back-traceability of faults problems capitalized in Section 1.1.

The scope of the thesis is defined from three interrelated perspectives:

- The *application domain* that dictates needs and constraints on the testing approach;
- The *testing technology* applied to meet these needs;
- The *formal framework* used to automatize the test purpose specification and generation procedures.

1.2.1 The application domain perspective

The applications that require extensive test effort are typically systems that integrate many functions onto one while ensuring the safe segregation of functions with different criticality levels. These systems are called mixed-criticality systems. Those mixed-critical MCS are usually based on a small set of core services that can be used to instantiate the system e.g., networked, virtualized multicore computers satisfying both the performance and segregation demands, which are extended depending on the specifics of a mission with relevant application software services [1]. However, the mixed criticality integration challenge also exists for the development process. Here, model-based engineering enables to integrate numerous functions of different criticality levels onto a shared complex hardware / software platform.

For instance, Robotic systems completing critical missions and having extensive degree of autonomy constitute one subclass of MCS. They operate in the dynamic and often under unpredictable conditions which require complex software solutions and sometimes even dynamic reconfigurability. Examples of such systems are surgical robots and assisting robots used in medical treatment procedures as well as spacecrafts having long term autonomous missions.

One of the practical examples of such MCS is the *Scrub Nurse Robot* (SNR) [6] developed at Tokyo Denki University. It was designed to collaborate with a human surgeon to assist at laparoscopic surgeries. Anticipating surgeon's movements, estimating the course of surgery and planning assisting actions (what instrument to pick, when and how to pass it to the surgeon) needs complex decision making methods, high degree of precision in timing and in manipulator movement planning. Exhaustive manual testing of SNR application appeared to be extremely time consuming and error prone due to factors such as variability of surgeon motion characteristics, imperfections of fusing different sensor types (cameras, IMU, etc.), mechanical inertia when moving instruments of different weight and shape, safety precautions of picking and handling instruments to surgeon (instrument must be in steady position and orientated properly when surgeon can safely grasp it), etc.

Due to safety-criticality and complex use cases, such applications as SNR set high requirements on testing. Extensive number of combinations of movement characteristics and variations of behaviour have to be covered by tests. Doubtlessly, exhaustive manual testing or even writing only test scripts for automatic test execution of these test cases remains out of the practical feasibility limits.

Satellite mission control. Launch and early orbit phase (LEOP) are the critical first steps in a spacecraft's life starting after the satellite separates from the launcher's upper-most stage. Mission control on-board software is responsible for activating, monitoring and verifying various subsystems on board the satellite, to ensure that the solar panels have deployed and that they undertake critical orbit and attitude control manoeuvres.

During LEOP the ground station software should provide extra telecommanding 'passes', time slots when the satellite is in view of a station. This provides mission controllers flexibility when complex command stacks must be sent up or additional software must be uploaded to troubleshoot any problems that may be found.

In the later phases of mission, despite the best preparations, unforeseen problems and challenges often arise that must be solved in real-time by mission control software autonomously when the satellite is out of communication range or already too far for timely communication. The onboard mission control operates in synch with core functionalities of the satellite control software including flight control, flight dynamics, telecommanding and data receipt via ground stations, and at the same time, has to take care of high-level functions such as conflict resolution, algorithm generation, event and plan generation.

Common features to be addressed when testing the use cases of satellite mission control are:

- significantly longer communication delays compared to local computation deadlines, e.g., when communicating with a ground station,
- security vulnerabilities due to communicating via open channels,
- functional interference between software components,
- non-determinism regarding events timing,
- varying control and data transmission capabilities (the communication depends on the satellite position in the orbit, atmospheric conditions), etc.

1.2.2 The testing technology perspective (to meet the requirements)

According to standard IEEE-1012-2004 testing is considered to be part of the software verification and validation (V&V) processes. *Verification* focuses on evaluating whether the software matches its specification, the *validation* goal is to assess if the specification matches the customer's requirements. Software testing as a method can be used in both. While the testing of functionality has been used traditionally in non-critical software development approaches, verifying and validating the predictable timing of critical services in the presence of heterogeneous and evolving distributed architectures remains still a challenge [6]. Therefore, validation methods like bench testing and encasing alone, although helpful and widely used, have become insufficient for MCS.

The quality and productivity issues of MCS V&V can be mitigated with model-based techniques and tools that operate on relevant level of abstraction [28]. MBT as one group of such techniques provides the opportunities for test automation and reduces systems V&V effort [39]. MBT suggests the use of a formal model for specifying the expected behaviour of System Under Test (SUT) and the test purpose. For instance, the behaviours or model elements to be covered by tests are subject to test purpose specification. Both, the SUT model and the test purpose specification are pre-requisites for automatic test generation. According to the taxonomy shown in Figure 1 [35] MBT captures the up-right corner of the *Accessibility-Level* plane and extends through all categories along the *Aspect* dimension. Thus, MBT advantages expose most clearly in *Integration* and *System* level testing where the functionality, timing, safety, security and other aspects of MCS are inspected in their most integrated form.

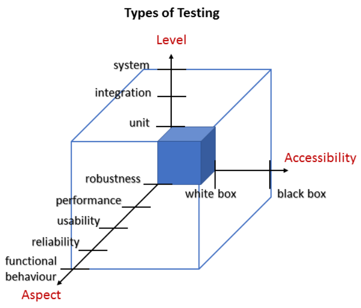


Figure 1 – Taxonomy of testing [40]

From test generation-execution point of view, in MBT the tests are generated either in *offline* or *online* mode. The online testing (test generation) can be divided internally by the methods how the test purpose is defined and how the test stimuli are selected on-the-fly. Online test execution requires more run-time resources for interpreting the SUT outputs and selecting new inputs until the test verdict about the conformance relation can be made. In offline testing, it is required to explore the whole state space of the model of SUT prior to generating the tests and therefore, the computationally expensive state space exploration is not needed during test execution anymore.

MBT focuses on the *conformance testing* where the SUT is considered to be a “black-box”, i.e. only its inputs and outputs are assumed to be externally controllable and observable, respectively. The internal behaviour of the system is abstracted away in a model. The aim of black-box conformance testing, according to [41], is to check if the behaviour observable on the system interfaces conforms to that of given in the system requirements specification. During MBT, a tester executes selected test cases (extracted from the system specification model) by running SUT in the test harness and emits a test verdict (*pass*, *fail*, *inconclusive*). The verdict shows test result in the sense of a conformance relation between SUT and the requirements model. A conformance relation used most often in MBT is Input-Output Conformance (IOCO) introduced by Tretmans [37]. The behaviour of IOCO-correct implementation should respect after some observations the following restrictions:

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

From MBT point of view, the aim of the thesis is to develop an *expressive test purpose specification language* and the *method of extracting complex test cases* from SUT models. The derived tests should satisfy the coverage criteria specified in the test purpose, be *correct*, which means that they should not signal errors in correct implementations, and should be *meaningful*, i.e. erroneous implementations should be detected with high probability [36]. To address the problems of complexity and traceability in MBT the thesis extends the model-based conformance testing with *a scenario based test description language* TDL^{TP} and an automatic *test generation technique* and *tool*.

1.2.3 Formal modelling perspective

MBT relies on formal models. The models are built from the requirements or design specifications in order to describe the expected behaviour of SUT in interaction with its environment. The model should be precise, unambiguous, and presented in a way relevant for correctness verification and test generation.

Another main purpose of using models in MBT is that the models of SUT are used to retrieve a test suite consisting of a set of test cases. The test cases are selected by means of a test case specification. The standard ETSI ES 202 951 v1.1.1 (2011-07) “Requirements for Modelling Notations” is used to define characteristics of MBT [3]. These characteristics concern main phases of the MBT process: SUT and its environment modelling, test purpose specification that defines the test coverage criteria, test generation and test execution steps.

Based on the rigour of semantics, the models used in testing can be classified into formal, semi-formal and informal ones. The models with strict formal semantics provide certainty that if the models represent systems adequately, then all the properties verified, really hold. However, formal models tend to have some practical usability limits for MBT, in particular, the scalability of test generation methods for large industrial systems. Due to high complexity their usage is typically limited with critical software domains such as automotive, medical, military, and critical infrastructure systems. The general purpose software industry uses semiformal modelling languages such as Unified Modelling Language (UML), Systems Modelling Language (SysML) and others which are expressive and intuitive to designers, but lack fully rigorous semantics. Regardless the lack of complete formal semantics they are preferred also due to elaborated graphical representations and tool support. Informal models are used to communicate the main ideas but they lack a clear semantics and are not suitable for development of critical systems. Regardless the wide use of UML, a considerable amount of testing theory has been conducted on formal models, in particular, based on different classes of state machines. An extensive survey on modelling formalisms used in MBT can be found in [30].

From the test goal specification and automated test generation point of view, the modelling formalism to be used for MCS should be expressive enough to specify the features of systems that are required to be tested. The class of systems in the thesis target domain - TCS and MCS can be characterized with the following features: behaviours of TCS have projection in the unbounded and dense time domain featuring effects such as Zeno behaviour and time bounded fairness; explicit reference to metric time constraints; simultaneous behaviours on different time scales and 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 importance (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 also practical usability requirements. Since the theory of timed automata and its extension Uppaal timed automata (Uppaal TA) theory satisfy the criteria listed above, the thesis relies on the underlying theory of Uppaal TA. Related Uppaal[4] tool family, supports modelling, validation and verification of real-time

[4]`http://www.uppaal.org`

systems. Uppaal TA model systems as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels and (or) shared data structures. Typical application areas include systems in which timing aspects are critical. In particular, for online test execution the tool Uppaal Tron and its extension for distributed testing DTron will be exploited in this work.

1.3 Related work

The requirements to the test purpose specification languages for MBT can be summarized as following:

- *intuitivity* to support human comprehension and to make the specification process user-friendly;
- *expressiveness* to capture the features and behaviours under test in a compact and unambiguous form;
- *formal semantics* to make the test purpose specifications verifiable and pertinent for automated test generation;
- *decidability* to make the test generation from test purpose specification algorithmically feasible.

The first two criteria have been capitalized in earlier attempts of designing test purpose specification languages. Check Case Definition Language (CCDL) [31] provides a high-level approach for requirements-based black-box system level testing. Test simulations and expected results specified in human readable form in CCDL can be compiled into executable test scripts. However, due to the lack of standardization, high-level test descriptions in CCDL are heavily tool-dependent and can be embedded only in its proprietary testing process.

High-level keyword-based test languages, using frameworks such as the Robot Framework[5], have also been integrated with MBT [32]. In some domains such as avionics [22] and automotive industry the efforts have been made to address the standardization of testing methods and languages, e.g. creating a meta-model for testing avionics systems [22], and the Automotive TestML [21] focusing on automotive systems. Similarly, the Open Test Sequence Exchange Format (OTX) [9] standardized at the International Organization for Standardization (ISO) provides tool-independent XML-based data exchange format [10] for the formal description and documentation of executable test sequences for automobile diagnostics. These efforts have focused primarily on enabling the exchange of test specifications between involved stakeholders and tools, and do not possess precise semantics. Due to their domain and purpose specialization the applicability of these languages in other domains is limited.

The Message Sequence Chart (MSC) [8] standardized at the International Telecommunication Union (ITU) was one of the first languages for the specification of scenarios, not focusing strictly on testing. In addition, [7] provides a formal specification of the semantics of MSC. Some of the features of MSC are adopted in UML as Sequence Diagram. The loose semantics of UML and various interpretations of sequence diagrams are a limiting factor for its use as a universal and consistent test description language [4].

The Precise UML [14] introduces a subset of UML and OCL for MBT trying to provide strict semantics of different diagrams. This was motivated by the need for behavioural specifications of SUT which are well suited for generating test cases out of SUT models.

[5]`https://robotframework.org`

Though, the experience with approaches that are based on a concrete executable language with strict semantics, such as TTCN-3, are not well suited for review and high-level interpretation of testing results. This is due to the low level of detail and the need to be able to understand the programming language-like syntax [20].

The domain specific and weakly formalized test purpose specification languages referred above share also a common set of disadvantages, either they have imprecise or informal semantics, lack of standardization, lack of comprehensive tool support, or poor interoperability with other development and testing tools.

European Telecommunications Standards Institute ETSI intended to address these shortcomings and develop a new specification language standard by introducing Test Purpose Language (TPLan) that supports the high-level expression of test purposes in prose [2]. Though TPLan provides notation for the standardized specification of test purposes, it leaves a gap between the declarative test purposes and the imperative test cases. Without formal semantics the development of test descriptions by means of different notations and dialects led to significant overhead and frequent inconsistencies that needed to be checked and fixed manually.

As a consequence, ETSI started a new initiative to develop the Test Description Language TDL [20] intended to bridge the gap between declarative test purposes and imperative test cases by offering a standardised approach for the specification of test descriptions. TDL provided a standardised meta-model, that was subsequently enriched with a graphical syntax, exchange format, and a UML profile. By 2015 ETSI succeeded in completing TDL as a common meta-model with well-defined semantics, which can be represented by means of different concrete notations. The main benefits of ETSI TDL outlined in [20] are:

- higher quality tests through better design;
- easier layout to review by non-testing experts;
- better and faster test development;
- seamless integration of methodology and tools.

The development of ETSI TDL was driven by industry where it is used primarily, but not exclusively, for functional testing. To enable the application of TDL in UML based working environments, a UML Profile for TDL (UP4TDL) [4] was developed. Domain-specific concepts are represented in a UML profile by means of *stereotypes*. A stereotype enables the extension of a UML meta-class with additional properties, relations, or constraints in order to address domain-specific concerns.

Though the ETSI TDL features one of the most advanced test purpose description language it has room for improvements including following:

- Automatic mapping of ETSI TDL to TTCN-3, that is needed for generating executable tests from TDL descriptions and re-using the existing TTCN-3 tools and frameworks for test execution, is not fully defined yet.
- Adaptation of TDL to different domains and types of testing in order to determine new language features and extensions, e.g. for security and performance testing, needs to be done.
- Restricted timing semantics. The *Time* package in TDL contains concepts for the specification of time operations, time constraints, and timers. TDL time operations include *Wait* and *Quiescence* and timer operations *Start*, *Stop*, *Timeout*. Since time

in TDL is global and progresses monotonically in discrete quantities there is no way of expressing synchronization conditions between local time events of parallel processes and detecting possible *Zeno computations* that can be analysed in continuous time models. Similarly, *time-divergency* (a path is time-divergent if its execution time is infinite), *timelock-freedom* cannot be *analysed* (a model is timelock-free if no state in the reachability set of the model contains a timelock - a state contains a timelock whenever no time-divergent paths emanate from it).

As one step further towards automatic test generation, the timed games based synthesis of test strategies has been introduced in [17] and implemented in the Uppaal Tiga tool. Timed computation tree logic (TCTL) is used to specify test purpose in this approach. TCTL has high expressive power and formal semantics relevant for expressing quantitative time properties combined with CTL operators such as AG ('*always*'), AF ('*inevitable*'), EG ('*potentially always*'), EF ('*possible*'), and > ('*leads-to*') [16].

Due to the complexity consideration of model checking, the TCTL syntax in Uppaal tool is limited with un-nested operators only, making the TCTL expression with respect to the temporal operators 'flat'. On the other hand, to specify the properties of timed reachability the 'flat' TCTL expressions are not sufficient for specifying complex properties and so called auxiliary property recognizing automata have to be added to the test models. An instance of such auxiliary automata is 'stopwatch' automata that are needed to compensate for that deficiency. Modifying the test model structure by adding property automata is not trivial for non-expert and may be error prone process leading to the unintended changes of model semantics.

As an extension to TCTL based test purpose specification language, the aim of thesis is to build an extra language layer (Test Scenario Definition Language - TDL^{TP}) for test scenario specification that is expressive, free from the limitations of 'flat' TCTL, interpretable in Uppaal TA, and thus, suited for automated test generation.

1.4 Main hypothesis and goals

The research goals of the thesis are based on following *hypothesis*:

- Due to its high expressive power the representation of test scenarios in TDL^{TP} is more compact compared to that of TCTL;
- Formal semantics of TDL^{TP} expressions enables
  - formal correctness verification of test models and test purpose specifications, incl. evaluation of their feasibility, time/space complexity;
  - automated generation of tests from verified models;
  - interpretation of different coverage criteria and back-tracing the root causes of found bugs.
- The TDL^{TP} expressions can be interrelated with other test coverage criteria and coverage metrics like structural coverage, function coverage, requirement coverage etc.

The technical goals of the thesis are following:

- Defining the syntax of test scenario definition language TDL^{TP};
- Defining the interpretation of TDL^{TP} in terms of language generating Uppaal TA;

- Designing and implementing the interpreter of TDL^{TP} and based on that a symbolic test generator;
- Integrating the TDL^{TP} usage into provably correct testing workflow described in Publication II.
- Demonstrating the feasibility of TDL^{TP} usage on a practical non-trivial test purpose specification and test generation case study.

1.5 The methodology used in the thesis

The research methodology applied in the thesis relies on the theory of automata, model checking and model-based testing. To achieve the thesis goals, following concrete techniques and methods are applied:

- TDL^{TP} syntax is defined in textual form that is inspired by temporal logic and process algebra notation;
- To reuse the existing test generation methods designed for Uppaal TA, the semantics of TDL^{TP} operators is interpreted by transforming the terms of TDL^{TP} to fragments of Uppaal TA that constitute the executable test model;
- TDL^{TP} interpretation rules are defined recursively as rewriting rules applicable to the terms of TDL^{TP} expressions;
- the correctness of SUT models and the test models generated from TDL^{TP} is verified using TCTL model checking. The verification follows methodology introduced in [43];
- the usability of TDL^{TP} is validated on a case study where TUT100 satellite software is tested and TDL^{TP} expressions are extracted from the satellite software requirements specification;
- the efficiency evaluation of the developed approach is measured in terms of saved test generation effort, and detected bugs *back-traceability* effort (the bug is *back-traceable* if the term of TDL expression can be referred to which caused the 'test fail').

1.6 Thesis main contributions

The thesis provides the following novelties in the field of model-based testing:

1. Defines a highly expressive test purpose specification language TDL^{TP} for complex test scenario specifications that is needed for MBT in safety and time critical systems.
2. Gives the semantics of TDL^{TP} operators that is defined in terms of model transformation rules that map declarative TDL^{TP} expressions to executable test models represented as Uppaal timed automata.
3. A provably correct test development process model is introduced and correctness conditions specified to be verified when showing correctness of test development steps.
4. Validation of the thesis theoretical results on the TUT100 satellite software case study.

1.7 Chapter summary

This chapter gives the motivation about why further research is needed in the domain of critical software engineering theory and practice. We define the research problems to be solved and their scope from three perspectives, the application domain that dictates the needs and constraints on the testing approach, the testing technology applied to meet these needs and the underlying formal framework to implement the MBT approach. Based on the analysis of the related work on test purpose specification languages it was concluded that extensions towards languages expressive power and semantic rigour are needed to support automatic test generation from coverage specifications. Research methodology how to solve these problems in the thesis was outlined. Finally, problem specific hypothesis and goals were defined to keep the focus and to prove the validity of targeted results.

2 PRELIMINARIES

In this chapter, the basics of testing are introduced in the context of model-based testing process while capitalizing the role of formal test model development and test purpose specification. Uppaal Timed Automata are defined as the modelling formalism for SUT description and test execution environment Uppaal Tron is explained as relevant tool for this class of models. To decide on the test sucess or fail, the Relativized Timed Input Output Conformance (RTIOCO) relation between the model and system under test is defined. Finally, related work on test coverage criteria and test purpose specification languages used in model-based testing are reviewed and the need for a new multi-criterial test scenario specification language is articulated.

2.1 Model-based testing

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

Like other software processes the MBT test development can follow different process models - waterfall, spiral, v-shape, agile etc. Regardless the process model all of them include five principal steps: *modelling of SUT*, *specification of the test purpose*, *test generation*, *deployment*, *and execution*. In this thesis, we exploit as an example the waterfall shape test development process model shown in Figure 2 [43].

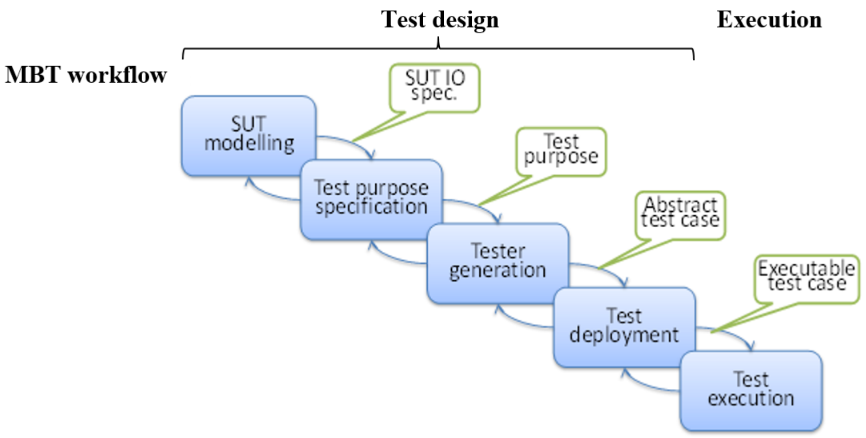


Figure 2 – Waterfall shape MBT workflow

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

The coverage of model structural elements (states and transitions) can be used also as a measure of *thoroughness* for a test suite. Thus, a test purpose is a specific objective (or property) that the tester wants to test, and can be seen as a specification of the test case. It may be expressed in terms of a single coverage item, scenarios, duration of the test run etc. As an example let us consider a requirement *"Test a state change from state s_A to state s_B"* in a model M^{SUT}. For this purpose, a test case should be generated such that, when starting from the initial state s_0 of M^{SUT}, it covers the specific state transition $s_A \to s_B$ of M^{SUT}. This requires that the test drives SUT to the state s_A, then executes $s_A \to s_B$ and the test should terminate in some safe state of M^{SUT} after that.

For non-deterministic systems a single precomputed test sequence may never succeed in reaching the test goal if M^{SUT} is not deterministically test-controllable. Instead of a single test sequence we need here an online testing strategy that is capable of reaching the goal even when SUT provides non-deterministic responses to test stimuli. The issue is addressed in [44] where the reactive planning online tester synthesis method is described.

In the third step, the abstract test suite is generated from the model consisting of SUT and the environment component so that the test purpose can be reached by executing the test suite. The test sequences generated are the intersection of behaviours of SUT and those specified by the test purpose.

The abstract test cases are deployed using the test execution framework. Deployment means transforming abstract tests to executable test scripts or by introducing test adapters which map symbolic model inputs to executable ones and the concrete outputs of SUT back to symbolic form to compare them with ones given in the model. The advantage of separating an abstract test suite and concrete test suite is the *platform* and *language independence* of the abstract test cases so, that the same abstract test case can be executed in different test execution environments.

In the fifth step, the deployed test cases are executed against the SUT. The test execution will result in a report that contains the outcome of the execution of the test cases. After the test execution, the detected bugs are analysed and their root cause backtracked. Hereby, for each test that reports a failure, the cause of the failure is determined and the program (or model) is corrected.

An example of symbolic test execution tool for Uppaal TA is Uppaal-TRON [25] depicted in Figure 3.

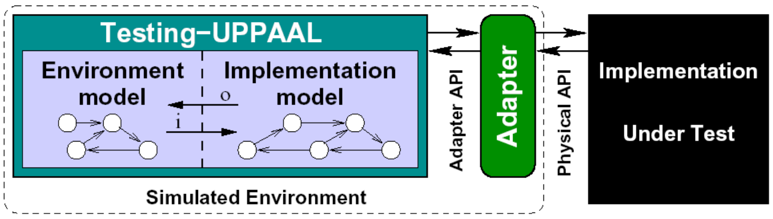


Figure 3 – Online MBT execution architecture: Uppaal-TRON

2.2 Uppaal timed automata

The Uppaal TA are extension of Timed Automata defined in [13] as follows:

Definition 2.1 (Timed Automaton)

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

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

- L is a finite set of locations (or nodes),
- $l_0 \in L$ is the initial location,
- $E \in L \times G(C) \times \Sigma \times 2^C \times L$ is the set of edges and
- $I : L \to G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $x \leq n$ or $x < n, n \in \mathbb{N}^+$. For shorthand we write $l \xrightarrow{g,a,r} l'$, to denote edges.

To define the clock conditions with respect to local events the functions known as clock resets are used. They map C to non-negative naturals $\mathbb{N}$. To keep the analysis tractable we limit the class of timed automata to rectangular timed automata where guard conditions are in conjunctive form with conjuncts of the form $k \sim n$ for $n \in \mathbb{N}$, and $\sim \in \{\geq, \leq, =, >, <\}$.

Definition 2.2 (Operational Semantics of TA)
The semantics of a timed automata are defined as a transition system. There are two types of transitions between states: either delay, or action (action transition).

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

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

- Timed transitions: $\langle l, u\rangle \to_d \langle l, u+d\rangle$ if $u \in C(l)$ and $(u+d) \in C(l)$ for a non-negative real $d \in \mathbb{R}^+$;

- Action transitions: $\langle l, u\rangle \to_a \langle l', u'\rangle$ if $l \xrightarrow{g,a,r} l', u \in g, u' = u[0/r]$ and $u' \in I(l')$.

The graphical representation of a timed automaton is considered as a directed graph, where locations are represented by the vertices and they are connected by edges (see Figure 2.3). Locations are labelled with invariants. Invariants are conjunctive Boolean expressions where the literals consist of clock variables and bound conditions of clock variables, e.g. $Clock1 \leq const1$.

In the graphical representation the edges are annotated with guards, synchronisations and updates. An edge is enabled by a guard in a state if and only if the guard evaluates to true. Processes (parameterized instances of automata templates) can synchronize transitions over channels. The execution of two edges of different automata labelled with a common channel is synchronized. For instance in Figure 4, the edge $WaitingCard \to Idle$ of Customer automaton and the edge $printReceipt \to Idle$ of ATM automaton synchronize over channel $card$. Updates express the change of the system state when the edge is executed, e.g., update $Clock1 = 0$ resets the value of model clock $Clock1$.

To model concurrent systems, TA are extended with parallel composition. A network of TA $N_{TA} = (T_1 || \ldots || T_n)$ is a collection of concurrent TA $T_i (i = 1, ..., n)$ composed using parallel composition. The state of the network is modelled by a configuration $\langle \bar{l}, \bar{c}\rangle$, where the first component is a location vector $\bar{l} = \langle l_1, \ldots, l_n\rangle$ where l_i is the current location of automaton T_i. The second component $\bar{c} \in \mathbb{R}^+$ is the valuation of all clock variables. The initial configuration of the network is $\langle \bar{l}, \bar{c}\rangle$, where all automata in N_{TA} are at initial configuration and the valuation of all clock variables is zero.

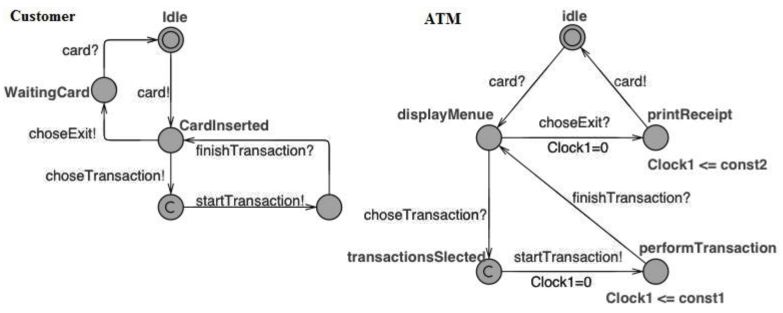


Figure 4 – The TA model of ATM

Similarly to TA network configuration a symbolic state s of a single timed automaton T is a pair $\langle l, c\rangle$, where $l \in L(T)$ is a location and c is the valuation of all clocks in $C(T)$. The valuation c must always satisfy the invariant constraints in the current location of the automaton $l : c \models I(l)$. There are three types of transitions in a TA network. A transition for TA network N_{TA} is defined by:

- *Action transition*: if $l_i \xrightarrow{g,a,r} l'_i$ is an action transition in the i-th automaton T_i with a guard over clock constraint $g(\bar{c}), \bar{c}' \models I(\bar{l}')$, an action $a \in A$, and an action transition $\langle \bar{l}, \bar{c}\rangle \xrightarrow{a} \langle \bar{l}_j, \bar{c}'\rangle$.
- *Synchronized transition*: if $l_i \xrightarrow{g_1,a,r_1} l'_i$ and $l_j \xrightarrow{g_2,\bar{a},r_2} l'_j$ are edges in i-th and j-th ($i \neq j$) automata synchronized via action a and its co-action $\bar{a}$ with $\bar{c} \models (g_i \wedge g_j)$ and $\bar{c}' \models I(\bar{l}')$, then $\langle \bar{l}, \bar{c}\rangle \xrightarrow{\tau} \langle \bar{l}', \bar{c}'\rangle$ is an internal action transition in N_{TA}, where $a, \tau \in A, \bar{l}' = \bar{l}[l'_i/l_i, l'_j/l_j]$, and $\bar{c}' = (r_1 \wedge r_2)(\bar{c})$. Notation $\bar{l}[l'_i/l_i]$ means substitution of all occurrences of l'_i with l_i in $\bar{l}$ [33].
- *Delay transition*: if $\delta \in \mathbb{R}^+$ is a delay with condition $\forall d < \delta : (\bar{c} + d) \models I(\bar{l})$, then $\langle \bar{l}, \bar{c}\rangle \xrightarrow{\delta} \langle \bar{l}', \bar{c} + \delta\rangle$ is a δ-delay transition in N_{TA}.

Uppaal timed automata [12] extend TA also with data types such as bool, integer, arrays of both types but also types of locations such as committed, urgent and normal. The advantage of this extension is that the model has rich enough modelling power to represent real-time and resource constraints and at the same time to be efficiently decidable for reachability analysis.

Definition 2.3 A timed automaton with data variables (TAD) over actions A, clock variables C and data variables V is a tuple (L, l_0, E) where

- L is a finite set of locations,
- l_0 is the initial location,
- $E \subseteq L \times G(C,V) \times A \times P^C \times L$ corresponds to the set of edges, where $G(C,V)$ is set of guard conditions that range over g.
  - g is a constraint in the form: $c \sim n$ or $v \sim n$ for $c \in C, v \in V, \sim \in \{\geq, \leq, =\}$ and n is a natural number.
  - the guards $G(C,V)$ can be divided into two parts: a conjunction of constraints over clock variables in the form $c \sim n$ and conjunction of constraints over data variables in the form $v \sim n$.

2.3 Conformance testing with Uppaal TA

We define the conformance relation using Uppaal TA semantics representation as a timed labelled transition system (TLTS). TLTS is a 4-tuple $\langle S, s_0, Act_{\tau\varepsilon}, \rightarrow\rangle$, where

- S is a non-empty set of states
- $s_0 \in S$ is the initial state
- $Act_{\tau\varepsilon} \overset{def}{=} Act \cup \{\tau\} \cup D$ are the actions Act including the internal action ε and time-passage actions τ; where $D = \{\varepsilon(d) \mid d \in \mathbb{R}^{+}\}$
- $\rightarrow \subseteq (S \times Act_{\tau\varepsilon} \times S)$ is the transition relation with the following consistency constraints:
  - Time Determinism whenever $s \xrightarrow{\varepsilon(d)} s'$ and $s \xrightarrow{\varepsilon(d)} s''$ then $s' = s''$
  - Time Additivity $\forall s, s'' \in S \wedge \forall d_1, d_2 \geq 0 : (\exists s' \in S : s \xrightarrow{\varepsilon(d_1)} s' \xrightarrow{\varepsilon(d_2)} s'')$ iff $s \xrightarrow{\varepsilon(d_1+d_2)} s''$
  - Null Delay $\forall s, s' \in S : s \xrightarrow{\varepsilon(0)} s'$ iff $s = s'$.

The labels in Act_{ε}(where $Act_{\varepsilon} \overset{def}{=} Act \cup D$) represent the observable input-, output actions $(Act = Act_I \cup Act_O)$ of a system, i.e. labelled actions and passage of time; the special label τ represents an unobservable internal action. A transition $(s, \mu, s') \in\rightarrow$ is denoted as $s \xrightarrow{\mu} s'$.

A *computation* is a finite or infinite sequence of transitions: $s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_3} \ldots \xrightarrow{\mu_{n-1}} s_{n-1} \xrightarrow{\mu_n} s_n (\rightarrow \ldots)$

A *timed trace* captures the observable aspects of a computation; it is the sequence of observable actions. The set of all $finite$ sequences of actions over Act_{ε} is denoted by Act_{ε}^{*}, while ε denotes the empty sequence. If $\sigma_1, \sigma_2 \in Act_{\varepsilon}^{*}$ then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2.

We consider normalised traces where actions and delays strictly alternate, starting with a delay. It has been shown that this set characterises the set of all traces [5]. A timed trace is thus a sequence $\sigma \in (\mathbb{R}_{\geq 0} \cdot Act)^{*} \cdot (\mathbb{R}_{\geq 0} + \varepsilon)$ such that $s_0 \xrightarrow{\sigma} s'$ for some $s' \in S$. The set of traces of TLTS S is noted $traces(S)$. The set of states that can be reached from state s via a trace σ is denoted as s after$_t \sigma$.

Definition 2.4 (*$after_t$*). Let $\langle S, Act, s_0, \rightarrow\rangle$ be a TLTS and $\sigma \in (\mathbb{R}_{\geq 0} \cdot Act)^{*} \cdot (\mathbb{R}_{\geq 0} + \varepsilon)$. Then s after$_t \sigma =_{def} \{s' \mid s \xrightarrow{\sigma} s'\}$.

Crucial for the definition of *tioco* is the set of delay and output labels of the outgoing transitions of a state.

Definition 2.5 (*$elapse(s)$* and *$out_t(s)$*). We define $elapse(s) =_{def} \{d \mid s \xrightarrow{d}\}$, and $out_t(s) =_{def} \{o \in Act_U \mid s \xrightarrow{o}\} \cup elapse(s)$. For $S' \subseteq S, out_t(S') =_{def} \cup_{s \in S'} out_t(s)$.

As in the *ioco* theory, we assume that implementations under test are input-enabled.

Definition 2.6 (Input-enabled *$TLTS$*). A $TLTS$ $\langle S, Act, s_0, \rightarrow\rangle$ is called input-enabled if and only if for all $s \in S$ and all $i \in Act_I : s \xrightarrow{i}$.

With the introduced concepts we can define the family of implementation relations $tioco_F$.

Definition 2.7 (*$tioco_F$*). Let P be an input-enabled $TLTS$, S a $TLTS$, and $F \subseteq traces(S)$. Then P conforms to S w.r.t. $tioco_F$ (written P tioco$_F S$) if and only if the following holds: $\forall \sigma \in F : out_t(P$ after$_t \sigma) \subseteq out_t(S$ after$_t \sigma)$.

Definition 2.8 Relativized timed input/output conformance (RTIOCO)
Let TTr_i be projection of $traces(S,E)$ on input alphabet Σ^I, i.e.

$$TTr_i(S,E) = traces(S,E)_{|\Sigma^I}$$

and let TTr_o be projection of $traces(S,E)$ on output alphabet Σ^O, i.e.

$$TTr_o(S,E) = traces(S,E)_{|\Sigma^O}$$

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

I rtioco S iff $\forall \sigma \in TTr_i(E)$:

$$TTr_o((I,E),\sigma) \sqsubseteq TTr_o((S,E),\sigma) \quad (1)$$

The timed traces generated by Upaal verifier provide symbolic test sequences to be executed by Uppaal Tron. For each test event, symbolic state in which the event occurred is specified in terms of clock constraints, variable valuation, list of next available states, and list of input/output actions. In the test sequence, when executed by TRON, a new test event occurs at a specific time, the clock constraints are updated, a transition to a new symbolic state occurs and the list of the next available states is updated.

2.4 Test coverage criteria

Test selection criteria are based on what hypothesis about the SUT behavior have to be covered by test cases. Like the code coverage measures what percentage of code is covered by tests, MBT coverge criteria measure how well the generated test suite covers the model elements. Many model coverage criteria have been adopted from the field of code coverage, e.g. statement coverage, decision/condition coverage, etc. In the MBT taxonomy of [40] following coverage based test selection criteria groups are described:

- Structural model coverage
- Data coverage
- Requirements coverage
- Test case specification coverage
- Random & stochastic coverage
- Fault based coverage

Structural coverage criteria deal with coverage of the control-flow through the model, based on analogy of control flow through programs. Structural model coverage criteria are derived from the key concepts of the modeling paradigms used for model-based testing. For example, transition-based notations have given rise to a family of coverage criteria that includes all-states, all-transitions, and all-transition pairs.

Pre/post notations refer to predicate coverage of guards and invariants in the model. For state transition models their control graph related notions can be used such as *all* *states*, *all transitions*, *all cycles*. Also data-flow coverage criteria reflect the data dependencies in the model computations, e.g. *all updates of variables that depend on given input values*.

Data coverage deals with the coverage of the input data space of an operation or transition in the model. It presumes choosing test data values from a large input data space. To reduce the number of possible test cases the data space is split into equivalence classes and one representative from each equivalence class has to be chosen to detect potential failures. The partitioning of the value domains into equivalence classes is often complemented by *boundary tests* of the equivalence classes.

Requirements coverage aims to generate a test suite that ensures that all the informal requirements are tested. Traceability of requirements can be automated if the fragments of the model can be associated with requirements of the SUT. For instance, if requirements' ID-s are used to label the transitions or states of a state machine or predicates of the post-conditions of a pre- and post-condition model, then test generation can aim to cover all requirements.

Random & stochastic coverage based generation is a computationally cheap way to generate tests that explore a wide range of system behaviors. These are mostly applicable to environment models, because it is the environment that determines the usage patterns of the SUT. The probabilities of actions are modelled directly or indirectly. The generated tests then follow an expected usage profile ([49] , [29]).

Fault-based coverage is used to find faults in the SUT using mutation coverage. This involves mutating the model, then generating tests that would distinguish between the mutated model and the original model. The assumption is that there is a correlation between faults in the model and in the SUT. The mutants are used for test generation against the implementation in order to check whether the latter allows for unspecified behaviors [33].

Ad-hoc test case specifications. To drive the test on heavily used cases, or to ensure that particular paths will be tested an explicit control of test execution can encoded in the test model. The notation used to express these test objectives may be the same as the notation used for SUT modelling, or it may be a declarative notation interpreted on the model. Notations commonly used for test objectives include UML Sequence diagrams, FSMs, regular expressions, temporal logic formulae, constraints and Markov chains (for expressing intended usage patterns). This family of coverage criteria relates to the scenario-based testing approach (e.g., ([50] , [38])) where test cases are generated from descriptions of abstract scenarios.

The main advantage of explicit test case specifications is that they give precise control over the test run. The drawback is that such specifications can be more labor intensive than choosing some structural model coverage criteria. Therefore, it is recommended to combine test selection criteria. One such strategy is to start generating tests using simple structural model coverage and data coverage criteria and then use test case specifications to enhance the coverage of SUT parts that are accessible only when applying complex test scenarios.

To summarize, a large variety of coverage criteria can be used to configure an automated test generation process. These criteria have different scopes and purposes, but the key is *complementarity of test selection criteria* [40]. To obtain good quality test suites, different coverage criteria should be combined, particularly when testing complex MCS. In other words, these applications need scenario based testing where multiple coverage criteria are seamlessly combined and applied jointly or separately in the steps of test scenarios. To minimize such multi-criterial scenario specification effort, instead of explicit test control scenario descriptions symbolic languages with rich expressive power are needed.

2.5 Chapter summary

Model-based testing of time/mission critical systems presumes relevant formal notation for describing time-dependant complex behaviors of SUT and the test coverage criteria for specifying critical test cases. In this chapter, syntax and semantics of Uppaal Timed Automata which is one of the most relevant formalism for such description, is described. To decide on the test sucess or fail the Relativized Timed Input Output Conformance (RTIOCO) relation between the Uppal TA model and system under test is defined. The taxonomy of test coverage criteria and test purpose specification languages used in model-based testing revieled that the available test specification languages suffer from expressibility and/or automatic test generation support for the class of SUTs of interest in this thesis. As a conclusion the need for a new multi-criterial test scenario specification language for time/mission critical systems is articulated.

3 PROVABLY CORRECT TEST DEVELOPMENT

The provably correct MBT process introduced in Section 2.1 (Figure 2) comprises test development steps (modelling the system under test, specifying the test purposes, generating the tests and executing them against SUT) that alternate with verification steps. In this chapter, the verification of test development steps is described and how the test results are made trustworthy 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.

3.1 The Correctness of SUT Models

3.1.1 Modelling Timing Aspects of SUT

For automated testing of input-output conformance of systems with time constraints we restrict ourselves with a subset of Uppaal TA that simplifies SUT model construction. Namely, we use a subset of Uppaal TA 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 SUT behaviour and timing constraints is *Action pattern* (or simply Action) depicted in Figure 5.

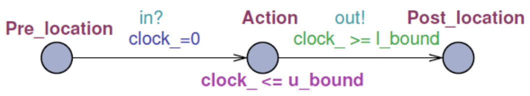


Figure 5 – Elementary modelling fragment

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 SUT input events (*stimuli* in the testing context) are generated by Tester, and the output events (SUT *responses*) are to make the reactions of SUT observable to Tester. In Uppaal TA, the interaction between SUT and Tester is modelled with *channels* that link synchronous input/output events.

The major timing constraint we represent in SUT model is the 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 repectively. The duration interval $[l\_bound, u\_bound]$ can be expressed in Uppaal TA as a pair of predicates on clocks as shown in Figure 5. 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 $clock\_ <= u\_bound$ of location $Action$ forces the $Action$ to terminate latest at time instant u_bound after the clock reset and guard $clock\_ >= l\_bound$ 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 SUT has two types of locations: *passive* and *active*. In passive locations SUT is waiting for test stimuli and in active locations SUT 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 5, the locations $Pre\_location$ and $Post\_location$ are passive while $Action$ is an active location.

We compose the SUT models from Action patterns using *sequential* and *alternative composition*.

Definition 3.1. (Composition of Action patterns). Let F_i and F_j be Uppaal TA 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 of [47] relies on the notion of well-formedness of the SUT model according to the following inductive definition.

Definition 3.2. (Well-formedness - wf of SUT models)

- The atomic Action pattern is well-formed;
- The sequential composition of well-formed patterns is well-formed;
- The alternative composition of well-formed patterns is well-formed if the output labels are distinguishable.

Proposition 1. Any Uppaal TA 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 Uppaal TA elements that do not match with the Definition 3.2, auxiliary pre-, and post-locations and internal ε-transition are added, that do not violate the i/o behaviour of the 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 3.2, including also Uppaal TA that have data variable updates, but in general well-formedness does not extend to models that include guards on data variables.

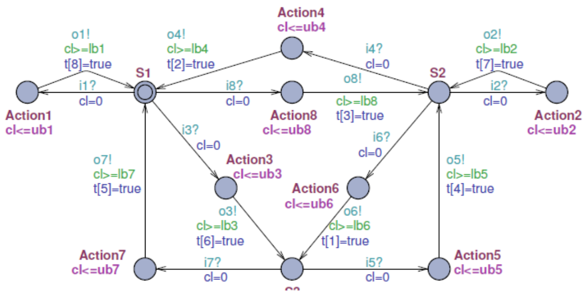


Figure 6 – An example of well-formed SUT model

In the rest of this chapter, we assume for test generation that M^{SUT} is well-formed and denote these models by $wf(M^{SUT})$. An example of a well-formed model we use throughout the chapter is depicted in Figure 6.

3.1.2 Correctness Conditions of SUT Models

The test generation method introduced in [48] and developed further for EFSM models in [27] assumes that the SUT 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 Uppaal TA models (see Definition 3.2).

3.1.2.1 Connected Control Structure and Output Observability We say that Uppaal TA model is $connected$ in the sense that there is an executable path from any location to any other location. Since the SUT model represents an open system that is interacting with its environment we need for verification by model checking a *nonrestrictive environment* model. According to [15] such an environment model has the role of a *canonical tester*. A canonical tester provides test stimuli and receives test responses in any possible order the SUT model can interact with its environment. A canonical tester can be easily generated for well-formed models according to the pattern depicted in Figure 7b (this is a canonical tester for the SUT model shown in Figure 7a).

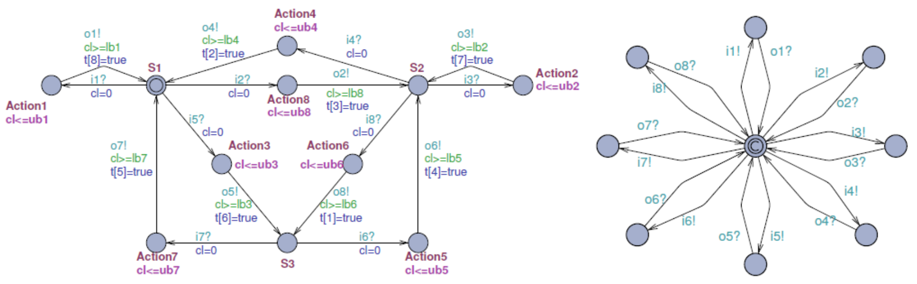


Figure 7 – Synchronous parallel composition of a) SUT and b) canonical tester models

The canonical tester composed with SUT model 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 SUT and the canonical tester (shown in Figure 7) the connectedness of SUT can be model checked with query (2) that expresses the absence of deadlocks in interaction between SUT and canonical tester.

$$A[]\, not\, deadlock \tag{2}$$

The *output observability* condition means that all state transitions of the SUT 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 SUT 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.

3.1.2.2 Input Enabledness. Input enabledness of SUT model means that blocking of SUT due to irrelevant test input never occurs. That presumes implicitly the input completeness of the SUT model. A naive way of implementing input enabledness in SUT models presumes introducing self-looping transitions with input labels that are not represented on other transitions that have the same source state. This makes SUT modelling tedious and leads to the exponential increase of its size in the size of the input alphabet. Alternatively, when relying on the notion of observational equivalence one can approximate the input enabledness in Uppaal TA by exploiting the semantics of synchronizing channels and encode the input symbols as boolean variables $I_1 \dots I_n \in \Sigma^{in}$. Then the pre-location of the Action pattern (see Figure 5) 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 $\Sigma^{in}(M^{SUT})$;
- add a self-loop edge (l_i^{pre},l_i^{pre}) that models acceptance of all inputs in $\Sigma^{in}(M^{SUT})$ except $I_j,\ j = 1,k$. To implement given constraint we specify the guard of the auxiliary edge (l_i^{pre},l_i^{pre}) with boolean expression: $not(\underset{j=1,k}{\vee} I_j)$.

Provided the branching factor $\mathscr{B}$ of the edges that are outgoing from l_i^{pre} is, as a rule, substantially smaller than the size of the input alphabet $|\Sigma^{in}(M^{SUT})|$, we can save $|\Sigma^{in}(M^{SUT})| - \mathscr{B}(l_i^{pre})$ 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 $|\Sigma^{in}(M^{SUT})|$ is depicted in Figure 8 (time constraints are ignored here for clarity). Symbols $I1$ and $I2$ in the figure denote predicates $Input == i1$ and $Input == i2$ respectively.

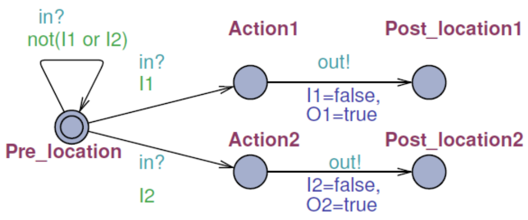


Figure 8 – Input-enabled model fragment

3.1.2.3 Strong Responsiveness Strong responsiveness (SR) means that there is no reachable *livelock* (a loop that includes only ε-transitions) in the SUT model. In other words, the model should always enter the *quiescent state* after finite number of steps. Since transforming M^{SUT} to $wf(M^{SUT})$ does not eliminate ε-transitions there is no guarantee that $wf(M^{SUT})$ is strongly responsive by construction (it is built-in feature of the Action pattern). To verify the SR propety of arbitrary M^{SUT} we apply Algorithm 1.

It is straightforward to infere that all steps except step 2 of Algorithm 1 are of linear complexity in the size of the M^{SUT} .

3.2 Correctness of RPT Tests

3.2.1 Functional Correctness of Tests

The tester designed or generated for given SUT model can be characterized by the test coverage criteria it is designed for. The test generator of [47] for online testing is aimed to cover SUT model structural elements. The structural coverage can be expressed by means of boolean "trap" variables as suggested in [24]. The traps are assignment expressions of

Algorithm 1 Strong responsiveness verification

```
1: According to the Action pattern in Figure 5 the $M^{SUT}$ input events are encoded by
   means of channel in? and a boolean variable $I_i$ that represents the condition that
   input value is $\iota_i$. Since input occurence in Uppaal models can be expressed as state
   property, we have to keep the input value indicating predicate $true$ 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 $\varepsilon$-transitions need to be labeled with
   update $EPS = true$ and following output edge with update $EPS = false$.
2: 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 the initial location
   of $M^{SUT}$. The query checks if any $\varepsilon$-transition is reachable from $l_0$ (that is necessary
   condition for violating SR-property).
3: Remove all non $\varepsilon$-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, check if the remaining part of
   model is empty. If yes then conclude that $M^{SUT}$ is strongly responsive, otherwise it is
   not.
```

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 3.3. *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 the SUT model.

Definition 3.4. *Optimality of the test*
We say that the test is length- (time-) optimal if there is no shorter (resp. faster) test run among those being coverage correct.

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

Direct way of verifying the coverage correctness of the tester is to run the model checking query (3) :

$$A\Diamond\forall(i:int[1,n])t[i]) \tag{3}$$

where $t[i]$ denotes i-th element of the array t of traps. The model for query (3) to be checked is assumed to be the synchronous parallel composition of SUT and Tester automata. For instance, the tester automaton generated using RPT generator [47] for SUT modelled in Figure 6 is depicted in Figure 9.

3.2.2 Invariance of Tests with Respect to Changing Time Constraints of SUT

In the previous section the coverage correctness of RPT tests was discussed without explicit reference to time constraints of the SUT 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., $lb_i = ub_i$ for all $i \in 1,\ldots|Action|$. Then the length of the test sequence and its duration in time are numerically equal. For

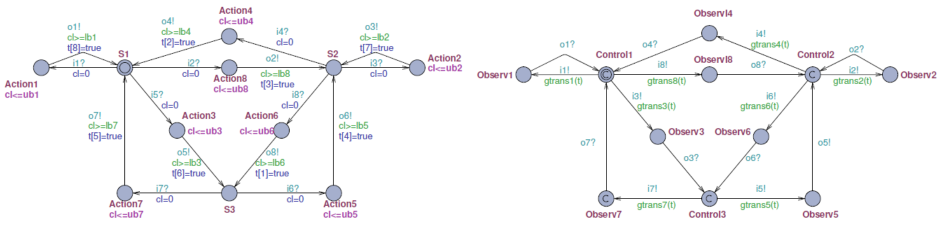


Figure 9 – Synchronous parallel composition of SUT and tester automata

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:

$$A\Diamond\forall(i:int[1,n])t[i])\wedge TimePass\leq TH \tag{4}$$

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

Generalizing this approach for SUT models with arbitrary time constraints we can assume that all edges of SUT model M^{SUT} are attributed with time constraints as described in Section 3.1.1. Since not all edges of M^{SUT} need to be labeled with traps (and thus covered by test) we apply compaction procedure to M^{SUT} 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 one trap-labelled edge the trapless sequence is adjacent to. As the result, the aggregate action becomes an atomic Action that copies the trap of the labelled adjacent edge. 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 to compute the test termination condition). After compaction of deterministic and timed SUT model it can be proved that the duration TH of a coverage correct tests have length that satisfies bound condition:

$$\sum_i lb_i \leq TH \leq \sum_i ub_i + \max_i(ub_i), \tag{5}$$

where index i ranges from 1 to n (n is the number of traps in M^{SUT}). In case of non-deterministic SUT models, for showing the length- and time-optimality of generated tests the bounded fairness assumption of M^{SUT} 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 source 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 DTRON [11] is capable of collecting the traces of monitoring the k-fairness and reporting about its violations. 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 $\sum_i lb_i$.

Proposition 2. (*Invariance of Tests with respect to changes of M^{SUT} timing*) [43].

Assume a trap labelled well-formed model $wf(M^{SUT})$ is compactified, the tester automaton M^{TST} generated using approach of [47] is invariant with respect to the variations of time constraints specified in M^{SUT}.

The proof consists of two steps, showing that

(*i*) Provided the reaction time of tester is negligible with respect to that of SUT, the control flow of the tester M^{TST} does not depend on the timing of M^{SUT} and

(*ii*) the M^{TST} behaviour does not influence the timing of controllable transitions of the M^{SUT}.

The practical implication of Proposition 2 is that a tester, once generated, can be used also for syntactic modification of M^{SUT} provided only timing parameters and initial values of traps have been changed. Note that invariance does not extend to structural modifications of M^{SUT}.

3.3 Correctness of test deployment

Practical execution of generated tests presumes the deployment of test adapters that map symbolic input alphabet used in the test model $M^{SUT} \; || M^{TST}$ to executable inputs. Similarly, real outputs from SUT need to be transformed back to symbolic outputs. This mapping performed by test adapters may introduce additional delays that are not reflected neither in SUT nor tester models. Also, distributed test configurations may add extra delays and propagation time to test execution, that can alter the ordering of test stimuli and responses specified in the model. By applying network monitors one can measure the latency of form $\triangle = [\delta^l, \delta^u]$ at each test input and output adapter. To verify the feasibility of the executable test suit, 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 $\Delta - testability$ introduced in [18]. Parameter $\triangle$ shows the delay between consecutive test stimuli necessary to maintain the ordering of input-output events at test 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. For any input symbol $a \in \Sigma^{in}(M^{SUT})$ and any output symbol $b \in \Sigma^{out}(M^{SUT})$ get the interval estimates of its total latency (including delay caused by adapters and propagation delays): $\Delta_a = [\delta^l_a, \delta^u_a]$ and $\Delta_b = [\delta^l_b, \delta^u_b]$ respectively.

Step 2: modify the timed guards Grd and invariants Inv of each action of $wf(M^{SUT})$ as follows:

- $Inv \cong cl \leq ub \longmapsto Inv' \cong cl \leq ub + \delta^u_a + \delta^u_b$
- $Grd \cong cl \geq lb \longmapsto Grd' \cong cl \geq lb + \delta^l_a + \delta^l_b$

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

3.4 Chapter summary

This chapter has been motivated by the need to increase the trust on testing results and to avoid running infeasible tests or tests that could lead to incorrect conclusions. Secondly, to reduce the test development time and to detect the test development faults in earliest possible phases the proposed approach enables verifying each intermediate test development product as soon as it is available, not just waiting for the final executable test product. The verification conditions and technique provided are relatively independent from the specifics of development method. This makes the verification approach customizable to different development process models and modeling formalisms. Another advantage of the approach is that it does not focus on functional properties only but correctness verification steps enable to prove also the correctness aspects of distributed SUT where timing constraints are substantial.

4 TEST PURPOSE SPECIFICATION LANGUAGE

In this chapter the *test purpose specification language* (TDL^{TP}) introduced in [45] is described. At first, a general overview of the language is given. Then, the syntax and semantics are defined. Based on the formal definition of the semantics a mapping of TDL^{TP} operators to Uppaal TA constructs is defined. The mapping serves the purpose of constructing the test model from the model of SUT and the test purpose expression in TDL^{TP}. We consider the usage of TDL^{TP} in the context of provably correct test development workflow and present the usage steps of TDL^{TP} starting from test purpose definition and following through test generation and execution steps until verdict (supplied with test diagnostics vector in case of test fail) formation.

4.1 Overview of the test purpose specification language

Test description in MBT relies typically on two formal description components: System Under Test (SUT) modelling language and the test purpose specification language. In our approach, Uppaal TA serves as a SUT specification language.

For the test purpose specification to be concise and still expressive it must be more abstract than SUT modeling language and not necessarily self-contained in the sense that its expressions are interpreted in the context of the SUT model only. It means that the terms of test purpose specification language TDL^{TP} refer to the SUT model structural elements of interest, they are called *test coverage items* (TCIs). The test purpose specification language TDL^{TP} allows expressing multiple coverage criteria in terms of TCIs, including test scenario constraints such as iteration, next, leads to, and structural coverage criteria such as *selected states*, *selected transitions*, *transition pairs*, and timing constraints, such as *time bounded leads to*.

Generating the test model based on the SUT model and TDL^{TP} coverage expression includes two phases. In the first phase, the TCIs have to be labelled with Boolean trap variables in the SUT model to make these coverage items referable in the TDL^{TP} expression. In case of a non-deterministic SUT model, the coverage of those elementary TCIs is ensured by *reactive planning tester* (RPT) *automata*, one automaton for each set of TCIs (see [47] for further details of RPT generation). In the second phase, a test *supervisor model* M^{SVR} is constructed from the TDL^{TP} expression, in order to trigger the RPT automata according to the test scenario so that the temporal and logical coverage constraints stated in TDL^{TP} specification would be satisfied. Here only sub-optimality of traces can be achieved due to SUT model non-determinism.

In case of deterministic SUT model, the RPT automata can be discarded since the Uppaal model checker solely is enough to generate the coverage witness traces from the parallel composition of SUT model and the test supervisor model M^{SVR}. Due to the fact that deterministic SUT models are deterministically controllable, these witness traces are sufficient to ensure the coverage of intended test purposes. The optimality of these traces is granted by the Uppaal model checker options *fastest trace* or *shortest trace*. In the rest of this chapter, we mainly focus on the deterministic case.

4.2 TDL^{TP} syntax

The ground terms in TDL^{TP} are sets of assignments to auxiliary variables called *trap variables* or simply *traps* added to the SUT model for test purpose specification. A trap is a Boolean variable assignment that labels a test coverage item, in case of Uppaal TA - an edge of the SUT model M^{SUT}. The value of all traps is initially set to *false*. When the edge of M^{SUT} labelled with a trap is visited during test execution the trap update function is executed and the trap value is set to *true*. We say that a trap *tr* is an *elementary trap* if its update function is unconditional, i.e. of shape $tr := true$.

Generally, we assume that the trap names are unique, trap update functions are non-recursive and their arguments have definite values whenever the edge labelled with that trap is executed. If the trap is *conditional*, i.e. the trap tr update condition is a Boolean expression instead of simple boolean assignment (we call it also update constraint) the arguments of which range over the sets of variables and constants of M^{SUT} and over the auxiliary constants and variables occurring in the test purpose specification in TDL^{TP}, e.g. references to other traps, event counters and the time bounds of model clocks.

Although we deal with finite sets of traps and their value domains the quantifiers are introduced in TDL^{TP} for notational convenience. To refer to the situations where many traps have to be true or false at once, we group these traps to sets called $trapsets$ (denoted by TS) and prefix them with trapset quantifiers - A for universal and E for existential quantification. $A(TS)$ means that all traps and $E(TS)$ means that at least one trap of the set TS has to be true for $A(TS)$ and $E(TS)$ to be true. To represent a trapset in Uppaal TA syntax we encode them as one-dimensional trap arrays and refer to individual traps in the array by array index value, e.g. i-th trap in TS is referred to as $TS[i]$.

In the following table we give the syntax of TDL^{TP} expressions in BNF (Algorithm 2).

Algorithm 2 Syntax of TDL^{TP} expressions in BNF

```
<Expression> ::=
  '(' <Expression> ')'
  | 'A' <TrapsetExpression>
  | 'E' <TrapsetExpression>
  | <UnaryOp> <Expression>
  | <Expression> <BinaryOp> <Expression>
  | <Expression> ~> <Expression>
  | <Expression> ~> '[' <RelOp> <NUM> ']' <Expression>
  | <Expression> <RelOp> <NUM>

<TrapsetExpression> ::=
  '(' <TrapsetExpression> ')'
  | '!' <ID>
  | <ID> '\' <ID>
  | <ID> ';' <ID>

<UnaryOp> ::= 'not'
<BinaryOp> ::= '&' | 'or' | '=>' | '<=>'
<RelOp> ::= '<' | '=' | '>' | '<=' | '>='
<ID> ::= ('TR') <NUM>
<NUM> ::= ('0' .. '9')+
```

4.3 TDL^{TP} semantics

To define the semantics of TDL^{TP} we assume there are given:

- a Uppaal TA model M,

- Trapset TS which is possibly a union of member trapsets $TS = \cup_{i=1,m} TS_i$, where the cardinality of each TS_i is n_i.

- $L : TS \to E(M)$, the labelling function that maps traps in TS to edges in $E(M)$, where $E(M)$ denotes the set of edges of the model M. We assume the *uniqueness* of the labeling within a trapset, i.e. there is at most one edge labelled with a trap from the given trapset but an edge can be labelled with many traps if each of them is from the different trapsets.

4.3.1 Atomic labelling function

The atomic labelling function is non-surjective and injective-only mapping between TS and $E(M)$, i.e. each element of TS is mapped to a unique edge of $E(M)$:

$$L : TS \to E(M),$$
$$\text{s.t } \forall e \in E(M) : TS_k[i] \in L(e) \wedge TS_l[j] \in L(e) \Rightarrow k \neq l \quad (6)$$

4.3.2 Derived labelling operations (trapset operations)

The formulas with a trapset operation symbol and trapset(s) identifiers being its argument(s) are called TDL^{TP} trapset formulas.

Relative complement of trapsets ($TS_1 \backslash TS_2$)

Only those edges labelled with traps of TS_1 and not with traps of TS_2 are in the relative complement trapset $TS_1 \backslash TS_2$:

$$[\![TS_1 \backslash TS_2]\!] \text{ iff}$$
$$\forall i \in [0, n_1], j \in [0, n_2], \exists e \in E(M) : TS_1[i] \in L(e) \wedge TS_2[j] \notin L(e) \quad (7)$$

Absolute complement of a trapset ($!TS$)

All edges that are not labelled with traps of TS are in the absolute complement trapset $!TS$:

$$[\![!TS]\!] \text{ iff } \forall i \in [0, n], \exists e \in E(M) : TS[i] \notin L(e) \quad (8)$$

Linked pairs of trapsets ($TS_1; TS_2$)

Two trapsets TS_1 and TS_2 are linked via the operator *next* (denoted ';') if and only if there exists a pair of edges in M which are labelled with traps of TS_1 and TS_2 respectively and which are connected through a location so that if any of traps in TS_1 is updated to $true$ on the k-th transition of model M execution trace σ then some trap of TS_2 is updated to $true$ in the $(k+1)$-th transition of that trace:

$$[\![TS_1; TS_2]\!] \text{ iff } \forall i \in [0, n_1], \exists j \in [0, n_2], \sigma, k : [|TS_1[i]|]_{\sigma^k} \Rightarrow [\![TS_2[j]]\!]_{\sigma^{k+1}}, \quad (9)$$

where $[\![TS]\!]_\sigma$ denotes the interpretation of the trapset TS on the trace σ and σ^l denotes the l-th suffix of the trace σ, i.e. the suffix which starts from l-th location of σ; n_1 and n_2 denote cardinalities of trapsets TS_1 and TS_2 respectively. Note that operator ';' enables expressing one of the "classical" structural coverage criteria *"selected transition pairs"*.

4.3.3 Interpretation of TDL^{TP} expressions

Quantifiers of trapsets

Given the definitions (6) - (9) of trapset operations we define the semantics of bounded universal quantifier A and bounded existential quantifier E of a trapset TS as follows:

$$[\![A(TS)]\!] \text{ iff } \forall i \in [0, n_i] : TS[i] \tag{10}$$

$$[\![E(TS)]\!] \text{ iff } \exists i \in [0, n_i] : TS[i], \tag{11}$$

where n denotes the cardinality of the trapset TS.

Note that quantification is defined on the trapsets only and not applicable on TDL^{TP} higher level operator expressions.

Logic connectives

Since recursive nesting of TDL^{TP} logic and temporal operators is allowed for better expressiveness we define the semantics of these higher level operators where the argument terms are not trapset formulas but derived from them using recursive nesting of logic and temporal operator symbols. Let SE, SE_1 and SE_2 denote such argument sub-formulas, then

$$[\![SE_1 \& SE_2]\!] \text{ iff } [\![SE_1]\!] \text{ and } [\![SE_2]\!] \tag{12}$$

$$[\![SE_1 \text{ or } SE_2]\!] \text{ iff } [\![SE_1]\!] \text{ or } [\![SE_2]\!] \tag{13}$$

$$SE_1 \Rightarrow SE_2 \equiv not(SE_1) \vee SE_2 \tag{14}$$

$$SE_1 \Leftrightarrow SE_2 \equiv (SE_1 \Rightarrow SE_2) \wedge (SE_2 \Rightarrow SE_1) \tag{15}$$

Temporal operators

- *'Leads to'* operator ($SE_1 \sim> SE_2$) in TDL^{TP} is inspired by Computation Tree Logic CTL *'always leads to'* operator, denoted by '$\varphi ==> \psi$' in Uppaal, which is equivalent to CTL formula $A\Box(\varphi \Rightarrow A\Diamond\psi)$. *Leads to* expresses that after reaching the state which satisfies φ in the computation all possible continuations of this computation reaches the state in which ψ is satisfied. For clarity we substitute the meta-symbols φ and ψ with non-terminals SE_1 and SE_2 of TDL^{TP}.

$$[\![SE_1 \sim> SE_2]\!] \text{ iff } \forall\sigma, \exists k, l, k \leq l : [\![SE_1]\!]_{\sigma^k} \Rightarrow [\![SE_2]\!]_{\sigma^l}, \tag{16}$$

  where σ^k denotes the k-th suffix of the trace σ, i.e. the suffix which starts from k-th location of σ, and $[\![SE]\!]_{\sigma^k}$ denotes the interpretation of TS on the k-th suffix of trace σ.

- *'Time bounded leads to'* means that TS_2 must occur after TS_1 and the time instance of TS_2 occurrence (measured relative to TS_1 occurrence time instance) satisfies constraint $\odot n$, where $\odot \in \{<, =, >, \leq, \geq\}$ and $n \in \mathbb{N}$:

$$[\![SE_1 \sim>_{[\odot n]} SE_2]\!] \text{ iff } \forall\sigma, \exists k, l, k \leq l : [\![SE_1]\!]_{\sigma^k} \Rightarrow [\![SE_2]\!]_{\sigma^l}, \tag{17}$$

- *'Conditional repetition'*. Let k enumerate the occurrences of $[\![SE]\!]$, then

$$[\![\#SE \odot n]\!] \text{ iff } \exists k : [\![SE]\!]^1 \sim> \cdots \sim> [\![SE]\!]^k \text{ and } k \odot n, \tag{18}$$

  where index variable k satisfies constraint $\odot n$, $\odot \in \{<, =, >, \leq, \geq\}$ and $n \in \mathbb{N}$.

The application of logic *not* to non-ground level TDL^{TP} terms has following interpretation:

$$\text{not } (A(TS)) \text{ iff } \exists i : [\![TS[i]]\!] = \text{ false} \tag{19}$$

$$\text{not } (E(TS)) \text{ iff } \forall i : [\![TS[i]]\!] = \text{ false} \tag{20}$$

$$\text{not } (SE_1 \wedge SE_2) \equiv \text{ not } (SE_1) \vee \text{ not } (SE_2) \tag{21}$$

$$\text{not } (SE_1 \vee SE_2) \equiv \text{ not } (SE_1) \wedge \text{ not } (SE_2) \tag{22}$$

$$\text{not } (SE_1 \Rightarrow SE_2) \equiv SE_1 \wedge \text{ not } (SE_2) \tag{23}$$

$$\text{not } (SE_1 \Leftrightarrow SE_2) \equiv \text{ not } (SE_1 \Rightarrow SE_2) \vee \text{ not } (SE_2 \Rightarrow SE_1) \tag{24}$$

$$\begin{aligned} &[\![\text{not}(SE_1 \sim> SE_2)]\!] \text{ iff } [\![\text{not}(SE_1)]\!] \text{ or } \forall k,l,k \leq l : \\ &\qquad\qquad [\![SE_1]\!]_{\sigma^k} \text{ and not } [\![SE_2]\!]_{\sigma^l} \end{aligned} \tag{25}$$

$$\begin{aligned} &\text{not } (SE_1 \sim>_{\odot n} SE_2) \equiv \text{ not}(SE_1 \sim> SE_2) \vee \forall \varphi : (SE_1 \sim>_{\varphi} SE_2) \\ &\qquad\qquad \Rightarrow (\varphi \Rightarrow) \text{ not}(\odot n)) \end{aligned} \tag{26}$$

$$\text{not } (\#TS \odot n) \equiv \forall \varphi : (\#TS\varphi) \Rightarrow (\varphi \Rightarrow \text{ not}(\odot n)) \tag{27}$$

where φ denotes the time bound constraint that yields the negation of constraint $\odot n$.

4.4 Mapping TDL^{TP} expressions to behavior recognizing automata

When mapping the TDL^{TP} formulae to test supervisor component automata we implement the mappings starting from ground level terms and move towards the root term by following the structure of the TDL^{TP} formula parse tree. The terminal nodes of any TDL^{TP} formula parse tree are trapset identifiers. The next above the terminal layer of the parse tree constitute the trapset operation symbols. The trapset operation symbols, in turn, are the arguments of logic and temporal operators. The ground level trapsets and the trapsets which are the results of trapset operations are mapped to the labelling of SUT model M^{SUT}. In the following the mappings are specified for TDL^{TP} trapset operations, logic operators and temporal operators in separate subsections.

4.4.1 Mapping TDL^{TP} trapset expressions to SUT model M^{SUT} labelling

When mapping the TDL^{TP} formulae to test supervisor component automata we implement the mappings starting from ground level terms and move towards the root term by following the TDL^{TP} formula parse tree structure.

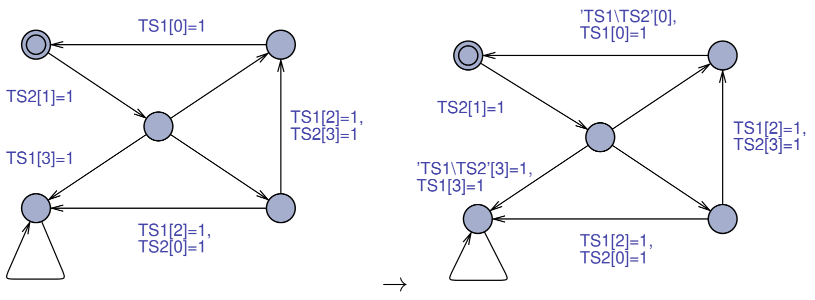


Figure 10 – Mapping TDL^{TP} expression TS_1/TS_2 to the SUT model labelling

<u>Mapping $M1$: Relative complement of trapsets (TS_1/TS_2)</u>

TS_1/TS_2 - mapping adds the expression TS_1/TS_2 traps only to these edges of M^{SUT}

which are labelled with traps of TS_1 and not with traps of TS_2. An example of such mapping is depicted in Figure 10.

Mapping $M2$: Absolute complement of a trapset ($!TS$)

The mapping of $!TS$ to SUT model labelling provides the labelling with $!TS$ traps all such edges of SUT model M^{SUT} which are not labelled with traps of TS. Example of this mapping is depicted in Figure 11.

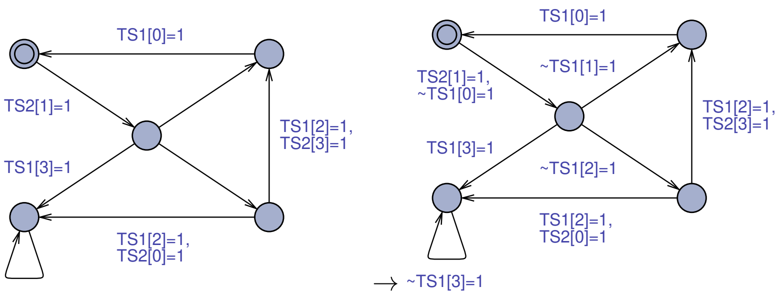


Figure 11 – Mapping TDL^{TP} expression $!TS$ to the SUT model labelling

Mapping $M3$: Linked pairs of trapsets ($TS_1;TS_2$)

The mapping of terms $TS_1;TS_2$ to labelling is implemented by the labelling algorithm Algorithm 3 ($L(TS_1;TS_2)$).

Algorithm 3 Labelling ($L(TS_1;TS_2)$)

```
1: for each $e', e'', i, j : pre(e'') = post(e') \wedge TS_1[i] \in L(e')$ do
2:     if $TS_2[j] \in L(e'')$ then
3:         $Asg(e') \leftarrow Asg(e'), flag(TS_1;TS_2) = true$
4:         $Asg(e'') \leftarrow Asg(e''), TS(TS_1;TS_2)[j] = (flag(TS_1;TS_2)?true : false)$
5:     end if
6:     $Asg(e'') \leftarrow Asg(e''), flag(TS_1;TS_2) = false$
7: end for
```

The example of Algorithm 3 application is demonstrated in Figure 12 Notice that the labelling concerns not only the edges that are labelled with traps of $TS1$ and $TS2$ but also those which depart from the same location as the edge with $TS2$ labelling. This is necessary for resetting the variable $flag$ which indicates the executing a trapset $TS1$ labelled edge in the previous step of the computation.

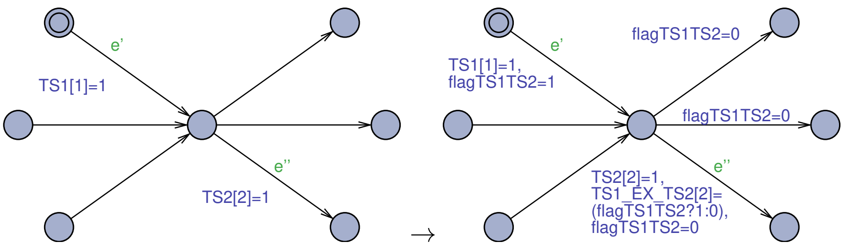


Figure 12 – Example of the application of Algorithm 3 ($L(TS_1;TS_2)$)

4.4.2 Mapping TDL^{TP} logic operators to recognizing automata

The indexing of trapset array elements, universal and existential quantifiers in Uppaal modelling language support direct mapping of trapset quantifiers to $forall$ and $exists$ expressions of Uppaal TA as shown in Figures 13 and 14.

Mapping M4: Universal quantification of the trapset

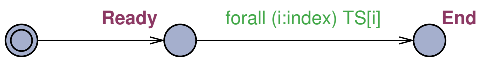


Figure 13 – An automaton that recognizes universally quantified trapset expressions

Mapping M5: Existential quantification of the trapset

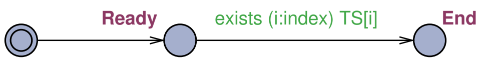


Figure 14 – The automaton that recognizes existentially quantified trapset expressions

Negation not

Since logic negation not can be pushed to ground level trapset terms by applying equivalences (19) - (27), the direct mappings of not formulas are not considered in this work.

Mapping M6: Conjunction of sub-formulas

The conjunction $SE1\&SE2$ is mapped to the automata fragment as shown in Figure 15. In the conjunction and disjunction automata depicted in the Figures 15 and 16 the guard conditions P and Q encode the argument terms $SE1$ and $SE2$ respectively. In conjunction automaton the End location is reachable from the initial location Idle if both P and Q are evaluate true in any order.

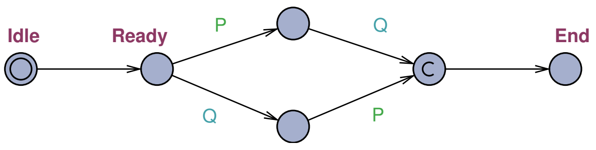


Figure 15 – The automaton that recognizes the conjunction of TDL^{TP} formulas P and Q

Mapping M7: Disjunction of sub-formulas

In the disjunction automaton the End location is reachable from the initial location $Idle$ if either P and Q are true.

The $implication$ of TDL^{TP} formulas can be defined using disjunction and negation as shown in formula (14) and their transformation to property automata are implemented through these mappings.

Similarly, the $equivalence$ of TDL^{TP} formulas can be expressed via conjunction and implication by using equivalence in formula (15).

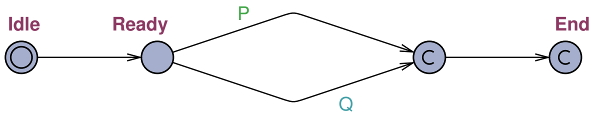


Figure 16 – The automaton that recognizes the disjunction of TDL^{TP} formulas P and Q

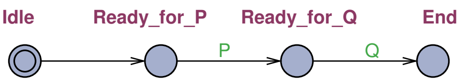


Figure 17 – 'Leads to' formula ($p \sim> q$) recognizing automaton

4.4.3 Mapping TDL^{TP} temporal operators to recognizing automata

Mapping M8: 'Leads to' ($p \sim> q$)

Mapping the 'leads to' operator to Uppaal TA produces the model fragment depicted in Figure 17.

Mapping M9: Timed leads to $p \sim>_{\odot con} q$

Mapping 'timed leads to' to a Uppaal TA fragment is depicted in Figure 18. It presumes an additional clock cl which is reset to 0 at the time instant when formula P become true. The condition '$cl <= d$' in Figure 18 a) sets the upper time bound d to the event when formula Q has to becomes true after P, i.e. after the clock cl reset. The mapping to property automaton depends on the time condition of *leads to*. So if the conditions is '$cl > d$' the mapping results in automaton shown in Figure 18 b).

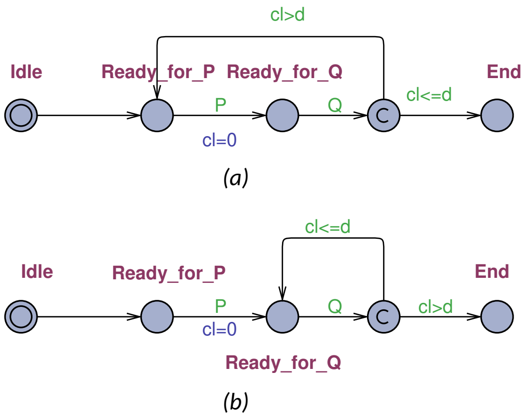


Figure 18 – 'Timed leads to' formula $P \sim>_{\odot d} Q$ recognizing automata a) with condition $cl \leq d$; b) with condition $cl > d$

Mapping M10: Conditional repetition $\#SE \odot n$:

The Uppaal TA fragment generated by the mapping of $\#SE \odot n$ includes a counter variable i to enumerate the events when the SE formula p becomes $true$. If the loop exit condition, e.g., '$i \geq n$', is satisfied then the transition to location End is fired without delay (the middle location is of type $committed$).

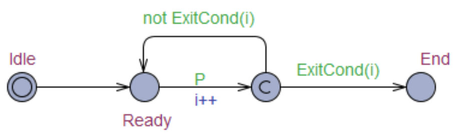


Figure 19 – Uppaal TA that implements conditional repetition

4.5 Reduction of the supervisor automata and the labelling of SUT

The TDL^{TP} expressions with many nested operators may become large and involve some overhead. Removal of this overhead in the formulas provides reduction in the state space needed for their model checking and improves their readability and comprehension.

The simplifications are formulated in terms of the parse tree of the TDL^{TP} formula and standard logic simplifications. Due to the nesting of operations in the TDL^{TP} formula the root operation can be any operator listed in the BNF grammar of TDL^{TP} but the terminals of the parse tree are always trapsets.

TDL^{TP} formulas consist of a static component (a trapset or a trapset expression) and optionally the logic and/or temporal component. The static component includes all sub-formulas of the parse tree branches from terminals to the lowest temporal expression, all sub-formulas above it are temporal and/or logic formulas (possibly mixed).

The trapset formulas are implemented by labelling operations such as *relative* and *absolute complement*. Only trapset formulas can be universally and existentially quantification. No nesting of quantifiers is allowed. Since the validity of root formula can be calculated only using the truth value of the highest trapset expression in the parse tree, all the trapsets being closer to the ground level trapset along the parse tree sub branche can be removed from the labelling of the SUT model. This reduction can be done after labelling the SUT model and applying all the trapset operations. An example of such reduction is demonstrated for relative complement operation $TS_1 \backslash TS_2$ in Figure 20.

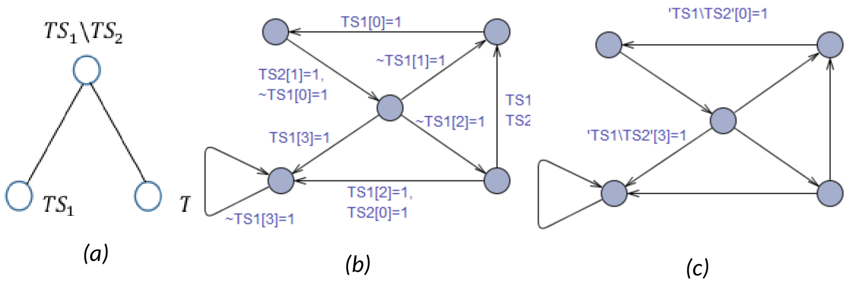


Figure 20 – Simplification of $TS_1 \backslash TS_2$ trapsets labelling: a) the parse tree of $TS_1 \backslash TS_2$; b) labelling of the SUT model with TS_1, TS_2 and $TS_1 \backslash TS_2$ c) reduced labelling of the SUT model M^{SUT}

Logic simplification follows after the trapset expression simplification is completed. Here standard logic simplifications are applicable:

$p \wedge p \equiv p$
$p \wedge$ not $p \equiv false$
$p \wedge false \equiv false$
$p \wedge true \equiv p$
$p \vee p \equiv p$
$p \vee$ not $p \equiv true$

$p \vee false \equiv p$
$p \vee true \equiv true$
We will introduce also a set of simplifications for TDL^{TP} temporal operations which follow from the semantics of operators and the properties of integer arithmetics:
$TS \equiv false$ if $TS = \emptyset$
$p \sim> false \equiv false$
$false \sim> p \equiv false$
$true \sim> p \equiv p$
$p \sim> true \equiv true$
$\#p = 1 \equiv p$
$\#p \odot n_1 \wedge \#p \odot n_2 \equiv \#p \odot max(n_1, n_2)$ if $\odot \in \{\geq, >\}$
$\#p \odot n_1 \vee \#p \odot n_2 \equiv \#p \odot min(n_1, n_2)$ if $\odot \in \{\geq, >, =\}$
$\#p \odot n_1 \wedge \#p \odot n_2 \equiv false$ if $\odot \in \{=\}$ and $n_1 \neq n_2$
$\#p \odot n_1 \sim> \#p \odot n_2 \equiv \#p \odot (n_1 + n_2)$ if $\odot \in \{\geq, >, =\}$
$\#p \odot n_1 \sim> \#p \odot n_2 \equiv \#p \odot min(n_1, n_2)$ if $\odot \in \{<\}$
$\#p \odot n_1 \wedge \#p \odot n_2 \equiv \#p \odot min(n_1, n_2)$ if $\odot \in \{<\}$
$\#p \odot n_1 \vee \#p \odot n_2 \equiv \#p \odot max(n_1, n_2)$ if $\odot \in \{<\}$
$p \sim>_{d_1} q \wedge p \sim>_{d_2} q \equiv p \sim>_{min(d_1,d_2)} q$ if $\odot \in \{\leq, <\}$
$p \sim>_{d_1} q \wedge p \sim>_{d_2} q \equiv p \sim>_{max(d_1,d_2)} q$ if $\odot \in \{>\}$

4.6 Composing the test supervisor model

The test supervisor model M^{SRV} is constructed as a parallel composition of single TDL^{TP} property recognizing automata each of which is produced by parsing the TDL^{TP} formula and mapping corresponding sub-formulae to the automaton template as defined in Section 4.5. To interrelate these sub-formula automata, two phases have to be completed:

1. Each trap labelled transition e of M^{SUT} (here we consider the traps which are left after labelling reduction as described in Subsection 4.5) has to be split in two edges e' and e'' connected via an auxiliary committed location l^c. The edge e' will inherit the labelling of e while e'' will be labelled with an auxiliary broadcast channel label that signals the trap update occurrence to the upper neighbor sub-formula automaton. We use the channel naming convention where a channel name has a prefix $ch\_$ followed by the trapset identifier, e.g. for an edge e labelled with the trap $TS[i]$, the broadcast channel label $ch\_TS!$ is added to the edge e'' (an example is shown in Figure 21 a)).

2. Each non-trapset formula automaton will be extended with a wrapping construct shown in Figure 21 b). The wrapper has one or two, depending if the sub-formula operation is unary or binary, channel labels to synchronize its state transition with those of its child expression(s). We call them downwards channels denoted by $Activate\_subOP1$, $Activate\_subOP2$ and used to activate the recognizing mode in the first and second sub-formula automata. Similarly, two broadcast channels are introduced to synchronize the state transition of sub-formula automata with their upper operation automaton. We call them upwards channels, denoted by $Activate\_OPi$ and $Done\_OPi$ in Figure 21 b). The root node is an exception because it has upwards channel only with the test $Stopwatch$ automaton (the $Stopwatch$ automaton will be explained in Section 4.7). If the sub-formulas of a given property automaton are mapped to trapset expressions then the back edge $Enn \rightarrow Idle$ to the initial state is labelled also with trapset reset function with TS being the argument trapset identifier. The TDL^{TP} operator automata extensions with wrapper constructs for

implementing their composition in test supervisor model M^{SVR} are shown in Figure 22.

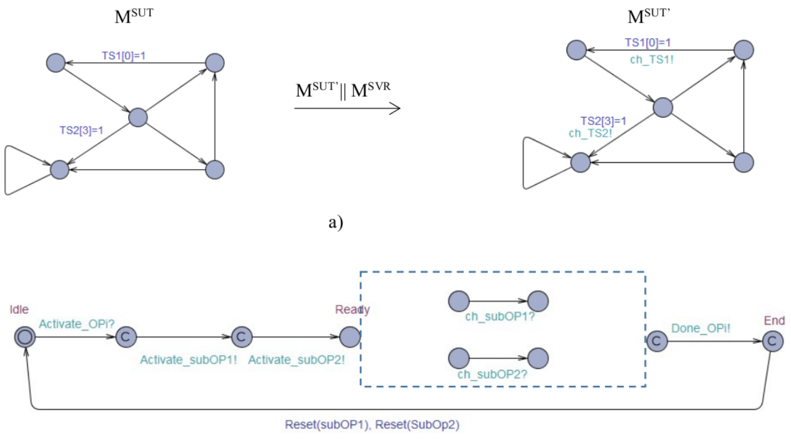


Figure 21 – a) Extending the trap labelled edges with synchronization conditions for composing the test supervisor; b) the wrapper pattern for composing operation recognizing automata

Note that the TDL^{TP} sub-formula meta-symbols P and Q in the original templates are replaced with channels which signal when the sub-formulas interpretation automata reach their local End locations.

4.7 Encoding the test verdict and test diagnostics in the tester model

The test verdict is yielded by the test $StopWatch$ automaton either when the automaton reaches its end state End within time bound TO. Otherwise, the timeout event $Swatch == TO$ triggers the transition to the terminal location $Failed$. Specifically, $Passed$ in the $StopWatch$ automaton is reached simultaneously with executing the root formula automaton transition to its End location. For example, in Figure 23, the automaton that implements root formula P, synchronizes its transition to the location End with $StopWatch$ transition to the location $Passed$ via upwards channel $Done\_P$. The construct is illustrated with StopWatch automaton depicted in Figure 23.

Another extension to the supervisor model is the capability of recording the test diagnostic information. For that each sub-formula of the test purpose specification formula φ^{TP} is indexed according to its position in the parse tree of φ^{TP}. A diagnostic array D of type Boolean and of the size equal to the number of sub-formulas in φ^{TP} is defined in the model. The initial valuation of D sets all its elements to $false$. Whenever a model fragment that corresponds to a sub-formula reaches its end state (that is sub-formula satisfaction state), the element in D that corresponds to that sub-formula is set to $true$. It means that if the test passes, the element of D that corresponds to the root expression is updated to $true$. Otherwise, in case the test fails, those elements of D remain $false$ which correspond to the sub-formula automata which conditions were not satisfied by the test model run. The updates $D[i] := true$ of array D elements, where i is the index of the sub-formula automaton M_i^{op}, are shown on the edges that enter their End locations. The expression automata M_i^{op} and their mapping to composition wrapping are shown in Figure 20.

The test model construction steps can be summarized now as follows:

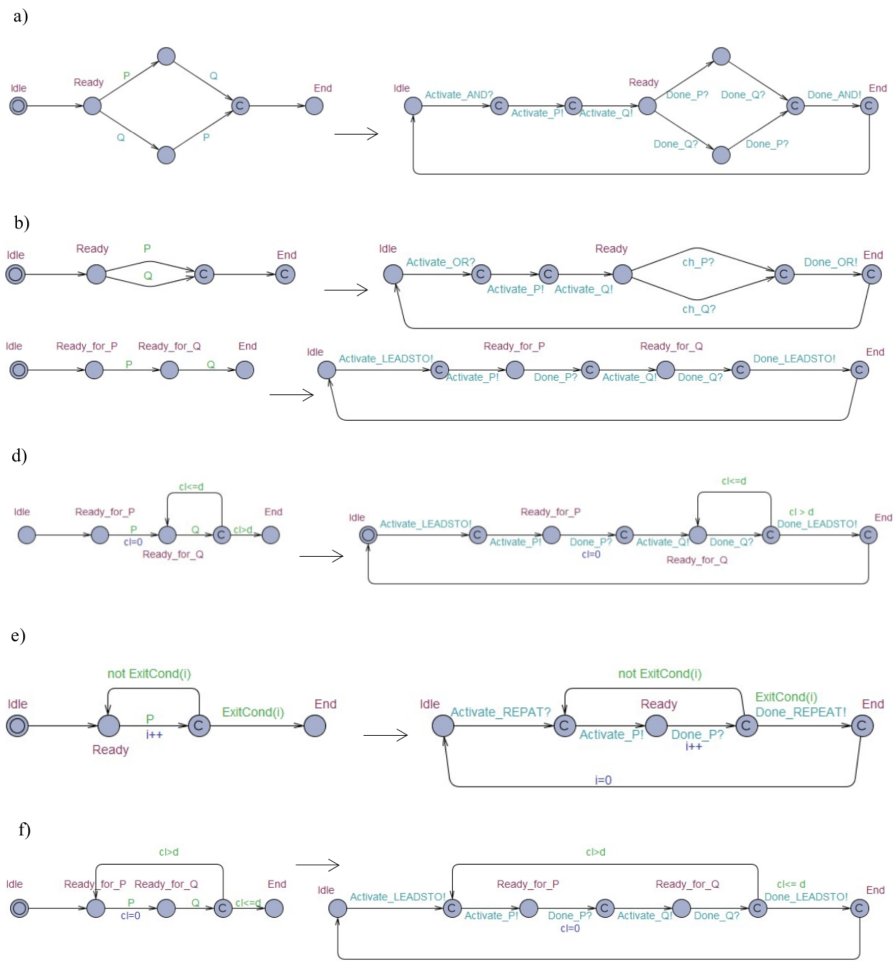


Figure 22 – Extending sub-formula automata templates with wrapping for test Supervisor composition a) And; b) Or; c) Leads to; d) Timed leads to with condition $cl \leq d$; e) Timed leads to with condition $cl > d$; f) Conditional repetition

1. The test purpose is specified as a TDL^{TP} expression φ^{TP} and simplified if possible;

2. Trapsets $TS_1, \ldots, TS_n$ are extracted from φ^{TP} and the ground level test coverage items are labelled with elements of non-intersecting trapsets;

3. The parse tree of the TDL^{TP} expression φ^{TP} is analysed and each of its sub-formula operator op_i is mapped using the mappings $M1$ to $M10$ to the automaton template M_i^{op} that corresponds to the sub-formula operation;

4. The labelling of M^{SUT} with traps is simplified by applying rules described in Section 4.6, and M^{SUT} linked with sub-formula automata M_i^{op} via wrapping construct that provides synchronization channels for signalling about reaching the state where sub-formula are satisfied;

5. Finally, the extension for diagnostics collection is added to automata M_i^{op} and the

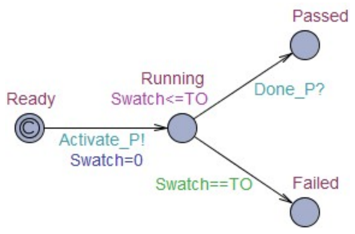


Figure 23 – Test Stopwatch automaton

root formula automaton is composed with $Stopwatch$ automaton M^{SW} which decides on the test pass or fail.

The total test model is synchronous parallel composition of component models $M^{SUT} \mid\mid M^{SW} \mid\mid M_i^{op}$.

4.8 Chapter summary

In this chapter the second main contribution of the thesis is presented. The test purpose specification language TDL^{TP} is introduced, its syntax and semantics are defined. Based on the formal definition of semantics a mapping of TDL^{TP} operators to Uppaal TA automata templates is defined. The mapping is used to automatize the construction of test models from the model of SUT and the test purpose specifications expressed in TDL^{TP}. To remove the possible overhead in TDL^{TP} expressions and in the test model we introduced a set of TDL^{TP} simplification rules to keep these specifications concise and readable, and also to reduce the size of the generated test model. Mapping of TDL^{TP} expressions to test automata is extended with diagnosis capability to identify the specification sub-formula which violation by SUT behavior could cause the test fail.

5 CASE STUDY

To demonstrate the practical usability of the test purpose specification language TDL^{TP} (Chapter 4) and provably correct test development method (Chapter 3) the TTU100 satellite program is chosen to be used as an example system. In this chapter, the general description of TTU100 satellite project is presented, its power management system as SUT is modelled for further test development. The test purposes are specified for three test cases which demonstrates the TDL^{TP} capability of expressing combinations of multiple coverage criteria in a single test case. From TDL^{TP} expressions the test models are constructed and the test sequences generated using Uppaal model checker. The chapter is concluding with comparison of the tests generated with the methods of this thesis and with those available using ordinary TCTL model checking.

5.1 System description

TTU100 satellite program is a university wide, interdisciplinary project that is carried out in association with partners from science- and commercial organizations. The main objective is to provide students of Tallinn University of Technology (TTU) with the experience of building a space system consisting of a 1U (10 cm x 10 cm x 10 cm) Nanosatellite and a ground station with mission planning and mission control software and later scientific experiments. The mission of the TTU100 Nanosatellite is to:

- perform remote sensing of Earth from Low Earth Orbit (LEO) in visible and infrared range of the electromagnetic spectrum. The satellite transmits remote sensing imaging data to ground stations on Earth which can be used for educational, scientific, space technology development and knowledge transfer purposes;
- test new space-to-Earth high data rate communication platform
- demonstrate and develop the technology for satellite attitude determination and control, on-board computer and smart power supply

The $TTU100$ space system consists of a *Space Segment* and a *Ground Segment*. The overall system achitecture is depicted in Figure 24.

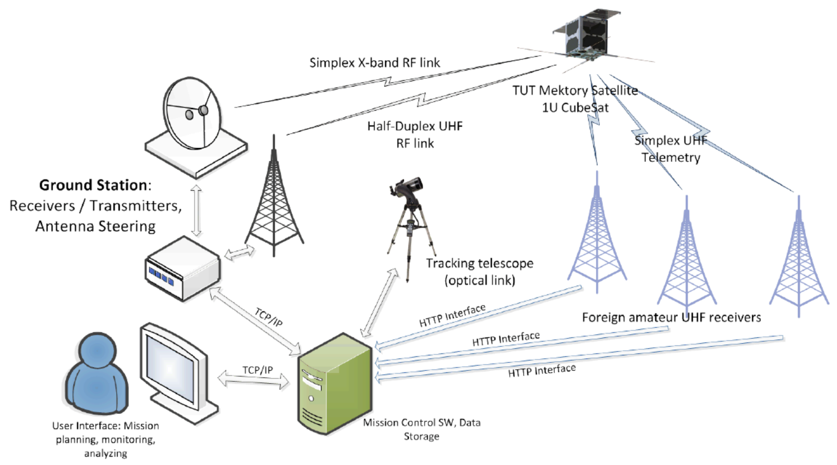

Simplex X-band RF link
TUT Mektory Satellite 1U CubeSat
Half-Duplex UHF RF link
Simplex UHF Telemetry
Ground Station: Receivers / Transmitters, Antenna Steering
Tracking telescope (optical link)
TCP/IP
HTTP Interface
HTTP Interface
HTTP Interface
Foreign amateur UHF receivers
TCP/IP
User Interface: Mission planning, monitoring, analyzing
Mission Control SW, Data Storage

Figure 24 – The space system architecture

5.1.1 Ground Segment

The *Ground Segment* is a system that is ment for communication with satellite and providing the means for storing and processing data aquired from the satellite. The Ground Segment consists of the

- Ground Station and
- Mission Control Software.

The role of the *Ground Station* is to forward telecommand frames from the mission control software to satellite as well as to receive the telemetry frames and high speed downlink frames from the satellite and forward these to the mission control software. In order to accomplish this task the ground station shall automatically track the satellite as it flies over the line of sight range of ground station. The ground station antenna alignment system shall maintain precise time of day in order to maintain precise satellite orbit propagation. The Ground Station is in a fixed location consisting of two separate communication systems:

- UHF telecommand and telemetry transceiver with steerable Yagi-Uda antennas;
- 10GHz band (ku-band) receiver with steerable parabolic dish antenna.

It also consists of receivers and transmiters for the radio frequency links, the antenna positioning and tracking system and the interface to mission control software.

The Mission Control Software provides user interface to carry out experiments on satellite and visualize the downloaded data. It enables building the experiment schedule and exchange data/files with satellite. The mission control software chops the data into individual packets to be sent to the satellite via ground station. The communication with the satellite may happen in burst mode where a number of packets is sent to satellite and then a number of packets is expected from the satellite. The mission control software also receives all frames from ground station, extracts the data from frames into respective data structures and provides visualization of selected data.

5.1.2 Space Segment

The Space Segment is a 1U size nanosatellite, according to CalPoly Cubesat Design Specification, on Earth's Sun Synchronous Orbit (650km altitude). The satellite hosts the following onboard functions and payload systems to carry out various experiments:

- smart electrical power supply (EPS) with solar energy harvesting and storage,
- satellite attitude determination and control system (ADCS),
- on-board computer (OBC) and (image) data processing,
- UHF band radio frequency communication system for remote control and status monitoring,
- Ku-band high speed radio frequency transmitter for image data downlink,
- camera and optics payload for image capture in RGB as well as in NIR bands.

For demonstration of our testing technique the smart Electrical Power Supply (EPS) subsystem is selected for SUT and its test purpose specifications are given in TDL^{TP}.

5.2 System under test modelling

The Electrical Power Supply subsystem (EPS) receives commands from other system components to change its operation mode and respond with its status information. In the high level model used for integration level testing we abstract from the concrete content of the commands and responses and give generic description of EPS interface behavior in response to the commands sent from its environment.

EPS is sampling its input periodically with period 20 time units. EPS wakeup time when detecting a new input command can vary within interval [15, 20] time units after previous sampling. After wakeup it is ready to receive incoming commands. Due to some internal maintenance procedure of EPS some of the commands when sent during the self-maintenance time can be ignored, they are not responded and need to be sent again. The command processing after its successful receiving takes at most 20 time units. Thereafter the validity of the command is checked with the help of CRC error-detecting code. If the error is detected the error data will be sent back to the EPS output port in $o\_response$ message. If the received command data are correct, the command is processed and its results returned in the output (response) $o\_message$. Since the EPS internal processing time is negligible compared to that of input sampling period and wakeup time, all the other locations except $start$ and $commandCreated$ are modelled as committed locations. The model M^{SUT} of the EPS is depicted in Figure 25.

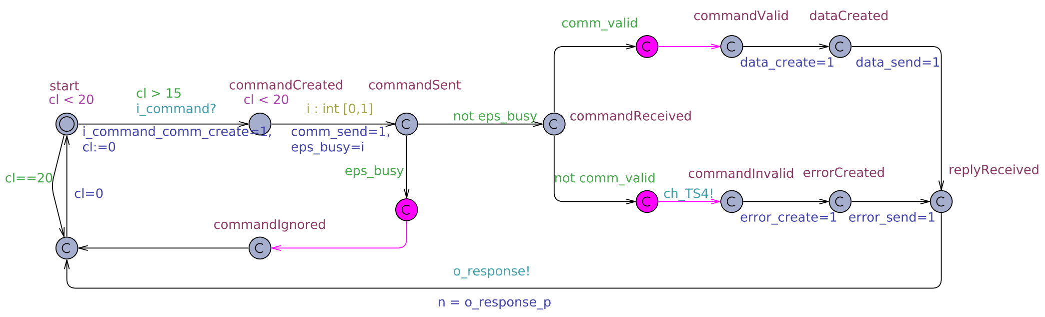


Figure 25 – The model M^{SUT} of the Electrical Power Supply subsystem

5.3 Test case 1

The goal of test case 1 is to show that after invalid command has been received the valid command can be received correctly and responded with acknowledgement.

5.3.1 Test purpose specification

We specify the test purpose in TDL^{TP} as formula

$$A(TS2;TS4) \sim> E(TS2;TS3), \tag{28}$$

which expresses that all transition pairs labelled with traps of TS_2 and TS_4 must lead in the trace to some pair of transitions labelled with traps of trapsets TS_2 and TS_3.

5.3.2 Labelling of M^{SUT}

The labelling of M^{SUT} starts from the ground level trapsets TS_2, TS_3 and TS_4 of the formula (28). These traps guide branching conditions to be satisfied in the scenario of Test Case 1. The labelling is shown in Figure 26.

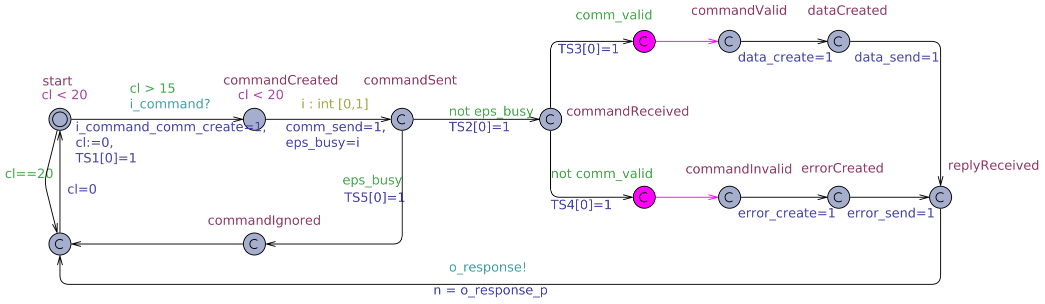


Figure 26 – Initial labelling for test case 1

Second level labelling results in applying trapset operation next “;” for pairs $TS_2;TS_3$ and $TS_2;TS_4$ which presumes introducing auxiliary variables $fl23$ and $fl24$ to identify occurrence of traps of TS_3 and TS_4 right after traps of TS_2. Since $TS_2;TS_3$ and $TS_2;TS_4$ are arguments of the upper “forall” and "exists" formula their occurrence should be signaled respectively to “forall”- and “exists” automata. For this purpose additional committed locations and edges (colored with magenta) with upwards channels ch_TS23 and ch_TS24 are introduced in Figure 27.

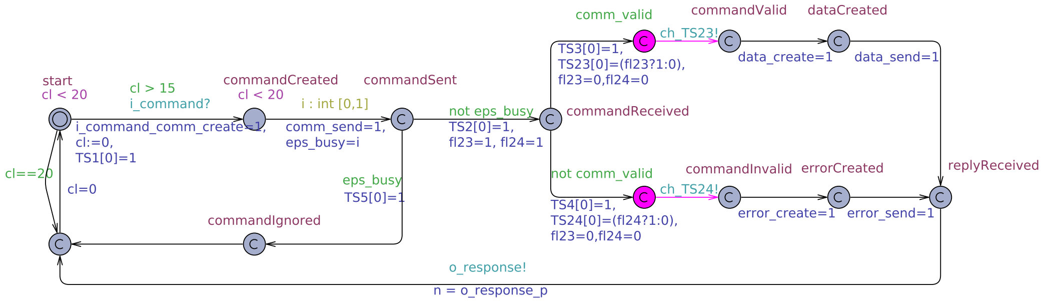


Figure 27 – Marking TS2;TS3 and TS2;TS4 trapsets for Test Case 1

5.3.3 Test model construction

When moving upwards in the parse tree of formula (28) the next operators that have $TS_2;TS_4$ and $TS_2;TS_3$ in arguments are forall $A(TS2;TS4)$ and exists $E(TS2;TS3)$ which automata are depicted in Figure 28a and 28b.

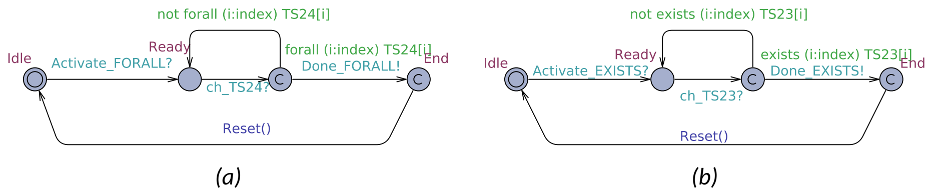


Figure 28 – a) automation that recognizes A(TS2;TS4); b) automation that recognizes E(TS2;TS3) respectively

The root operator in the formula (28) is ‘leads to’ the arguments of which are $A(TS2;TS4)$

and $E(TS2;TS3)$. The automaton that recognizes $A(TS2;TS4) \sim> E(TS2;TS3)$ is depicted in Figure 29.

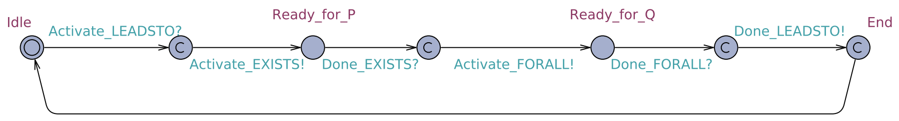


Figure 29 – Recognizing automaton of $A(TS2;TS4) \sim> E(TS2;TS3)$

The full test model for generating test sequences of test scenario $A(TS2;TS4) \sim> E(TS2;TS3)$ is shown in figure 30.

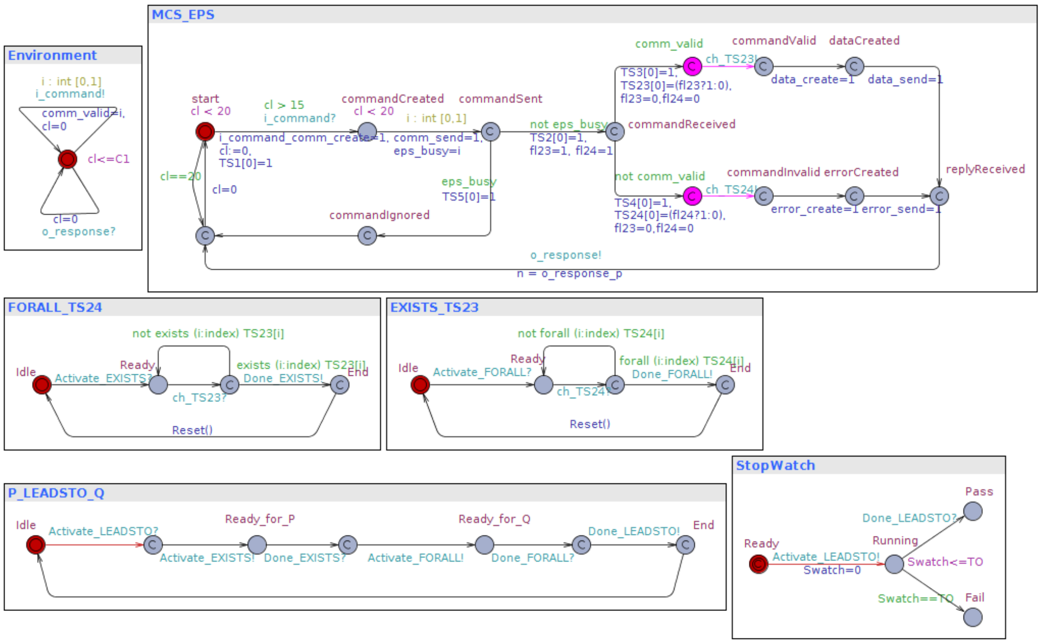


Figure 30 – Test model for implementing test scenario $A(TS2;TS4) \sim> E(TS2;TS3)$

5.3.4 Generating test sequences

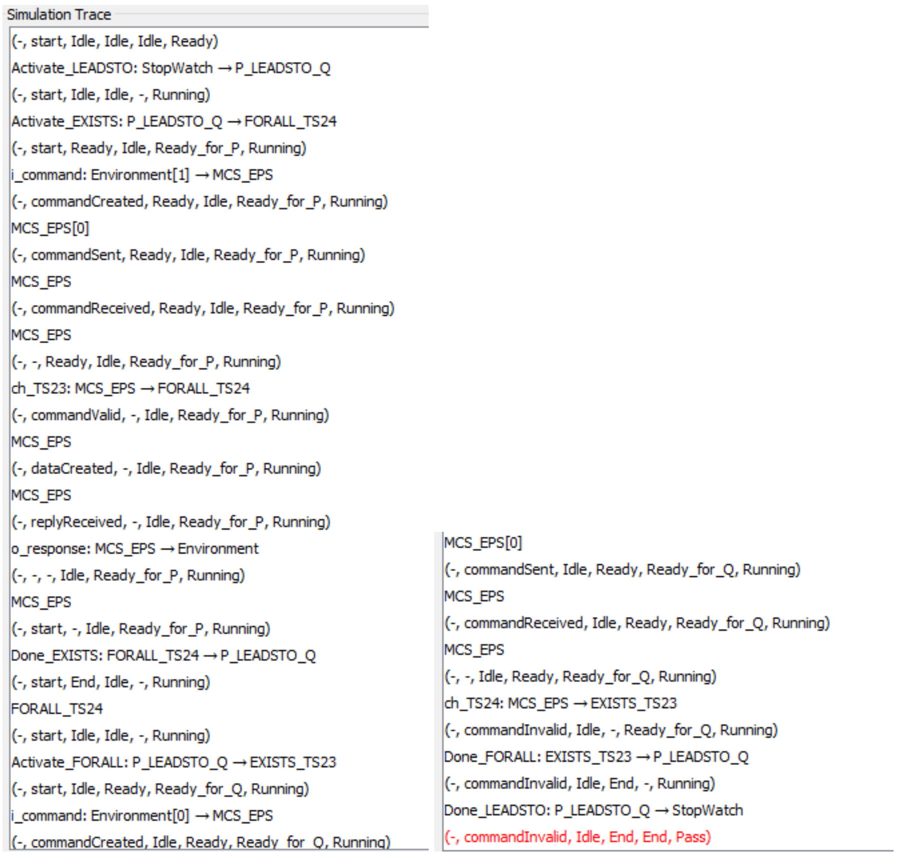
The test sequences of the SUT model M^{SUT} shown in Figure 25 and of the scenario $A(TS2;TS4) \sim> E(TS2;TS3)$ are generated by running the model checking query

$$E <> StopWatch.Pass$$

There are three options of selecting the trace for test – shortest, fastest, or some. The trace generated with model checking option $shortest$ is shown in the Figure 31.

5.4 Test case 2

In the Test Case 2 we exemplify how to specify the multiple repetition of earlier specified test scenarios which have slight modifications. Let the Scenario 1 has to be modified so that the quantification of $TS2;TS4$ and $TS2;TS3$ is switched. Instead of $A(TS2;TS4) \sim>$



```
Simulation Trace
(-, start, Idle, Idle, Idle, Ready)
Activate_LEADSTO: StopWatch → P_LEADSTO_Q
(-, start, Idle, Idle, -, Running)
Activate_EXISTS: P_LEADSTO_Q → FORALL_TS24
(-, start, Ready, Idle, Ready_for_P, Running)
i_command: Environment[1] → MCS_EPS
(-, commandCreated, Ready, Idle, Ready_for_P, Running)
MCS_EPS[0]
(-, commandSent, Ready, Idle, Ready_for_P, Running)
MCS_EPS
(-, commandReceived, Ready, Idle, Ready_for_P, Running)
MCS_EPS
(-, -, Ready, Idle, Ready_for_P, Running)
ch_TS23: MCS_EPS → FORALL_TS24
(-, commandValid, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, dataCreated, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, replyReceived, -, Idle, Ready_for_P, Running)
o_response: MCS_EPS → Environment
(-, -, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, start, -, Idle, Ready_for_P, Running)
Done_EXISTS: FORALL_TS24 → P_LEADSTO_Q
(-, start, End, Idle, -, Running)
FORALL_TS24
(-, start, Idle, Idle, -, Running)
Activate_FORALL: P_LEADSTO_Q → EXISTS_TS23
(-, start, Idle, Ready, Ready_for_Q, Running)
i_command: Environment[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready_for_Q, Running)
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready_for_Q, Running)
MCS_EPS
(-, commandReceived, Idle, Ready, Ready_for_Q, Running)
MCS_EPS
(-, -, Idle, Ready, Ready_for_Q, Running)
ch_TS24: MCS_EPS → EXISTS_TS23
(-, commandInvalid, Idle, -, Ready_for_Q, Running)
Done_FORALL: EXISTS_TS23 → P_LEADSTO_Q
(-, commandInvalid, Idle, End, -, Running)
Done_LEADSTO: P_LEADSTO_Q → StopWatch
(-, commandInvalid, Idle, End, End, Pass)
```

Figure 31 – Shortest trace that satisfies the test purpose of the Test Case 1

$E(TS2;TS3)$ the test purpose is to execute scenario with $E(TS2;TS4) \sim> A(TS2;TS3)$ and run it at least 2 times in Scenario 2. The test purpose specification in TDL^{TP} is expressed in formula (29).

$$\#(E(TS2;TS4) \sim> A(TS2;TS3)) >= 2 \tag{29}$$

5.4.1 Test model construction

The first steps of test model construction for Test Case 2 are similar to that of Test Case 1 and therefore we discard them. We consider only the introduction of a bounded repeat construction which runs the scenario $(E(TS2;TS4) \sim> A(TS2;TS3))$ two or more times. The channel $Activate\_LEADSTO$ starts the automata that interpret the expression $E(TS2;TS4) \sim> A(TS2;TS3)$. Local counter variable i is used to count the occurrences of this nested expression. Each time the expression is satisfied it is signaled to $repeat$ automaton via channel $Done\_LEADSTO$ (Figure 32).

The entire test model of Test Case 2 is depicted in Figure 33.

5.4.2 Generating test sequences

The test sequences of the SUT model M^{SUT} shown in Figure 25 and of the scenario

$$\#(E(TS2;TS4)A(TS2;TS3)) >= 2$$

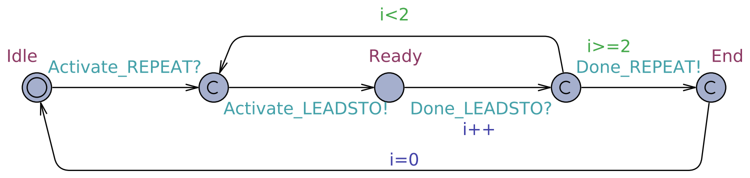


Figure 32 – The automaton of the root operator of TDLTP expression $\#(E(TS2;TS4) \sim> A(TS2;TS3)) >= 2$

are generated by running the model checking query

$$E <> StopWatch.Pass$$

similarly to Test Case 1. The trace generated with model checking option fastest is provided in the Figure 34.

5.5 Test case 3

The Test Case 3 is designed to show that after being non-responsive due to the internal maintenance procedures in EPS it is still capable of reacting to incoming commands regardless the commands are valid or with corrupted data. Both cases are allowed in the test run multiple times, all valid command receives have to be tried two times and some of corrupted receives three or more times. This is expressed in TDL^{TP} formula (30).

$$E(TS5) \sim> (\#(A(TS3)) == 2 \wedge \#(E(TS4)) >= 3) \tag{30}$$

5.5.1 Test model construction

The test purpose stated in (5.3) needs different labelling than Test Cases 1 and 2. Since the test purpose does not include 'next' trapset operations the labelling with traps of $TS3$, $TS4$ and $TS5$ suffices. The SUT model labelled with these traps and auxiliary edges with upwards signaling channels (coloured with magenta) is presented in Figure 35.

The repetitions of universal trapset $A(TS3)$ and existential trapset $E(TS4)$ are implemented with automata that recognize sub-formulas $\#(A(TS3)) == 2$ and $\#(E(TS4)) >= 3$ represented in the Figure 36 a) and Figure 36 b) respectively.

The recognizing automata of sub-formulas $E(TS5)$, $A(TS3)$ and $E(TS4)$ are represented in Figure 37.

The root operator is 'leads to' that is implemented by the automaton in Figure 38 and full test model of the Test Case 3 is depicted in Figure 39.

5.5.2 Generating test sequences

The model checking query for generating the test sequence of scenario

$$E(TS5) \sim> (\#(A(TS3)) == 2 \wedge \#(E(TS4)) >= 3)$$

is executed with option 'some' and the trace is exposed in Figure 40.

5.6 Chapter summary

Our experiments with three test cases on a SUT TTU100 Nanosatellite Electrical Power Supply subsystem show that the test purpose specification language TDL^{TP} developed in the thesis is applicable when complex test scenarios with multiple coverage criteria are of

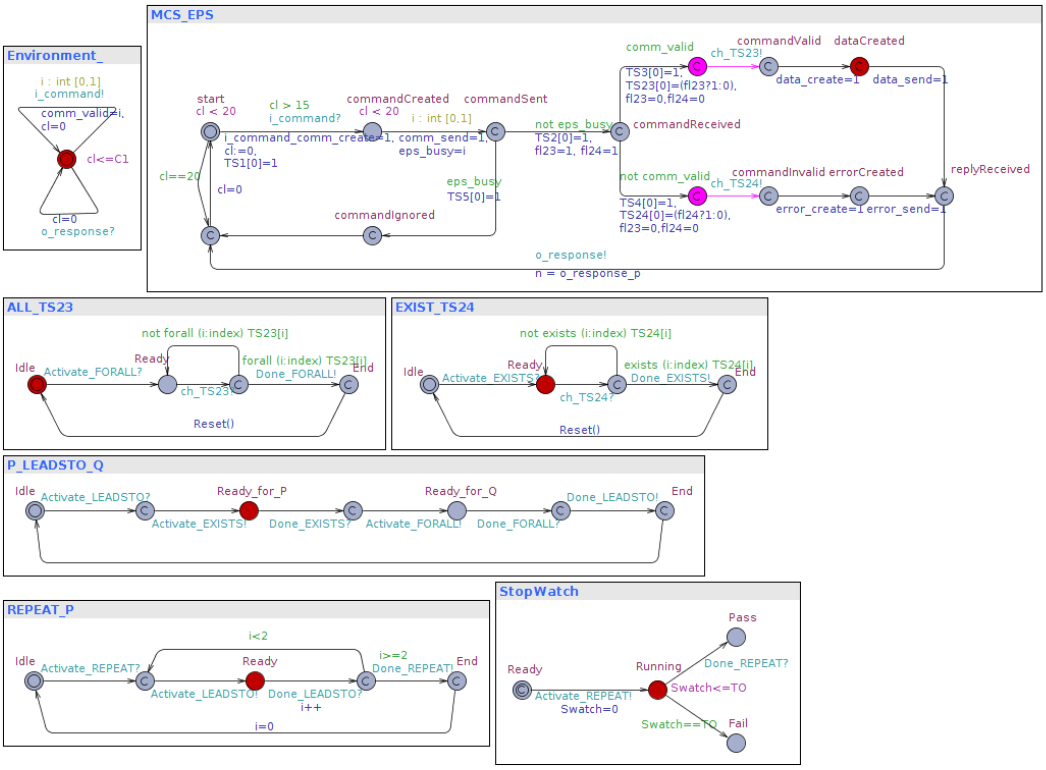
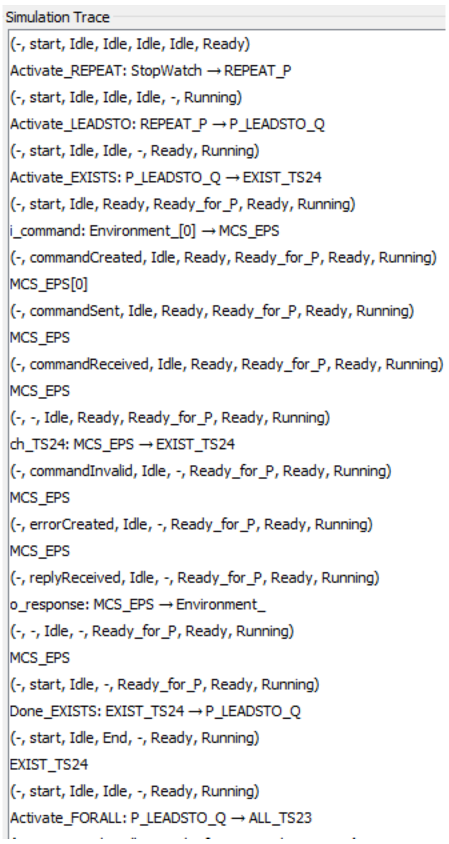
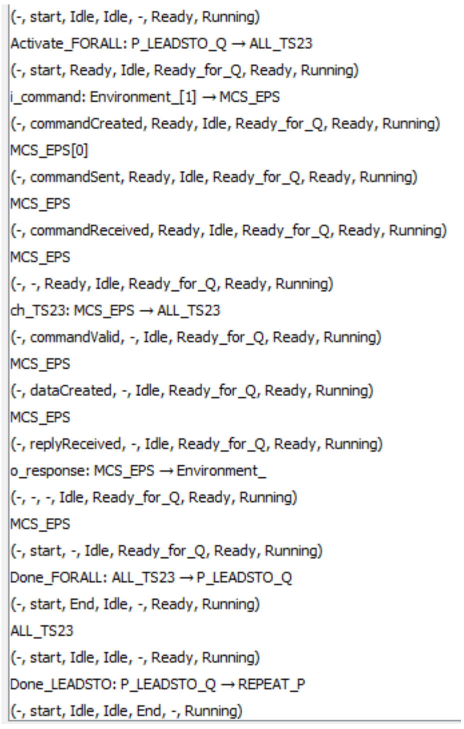


Figure 33 – Test model of the Test Case 2

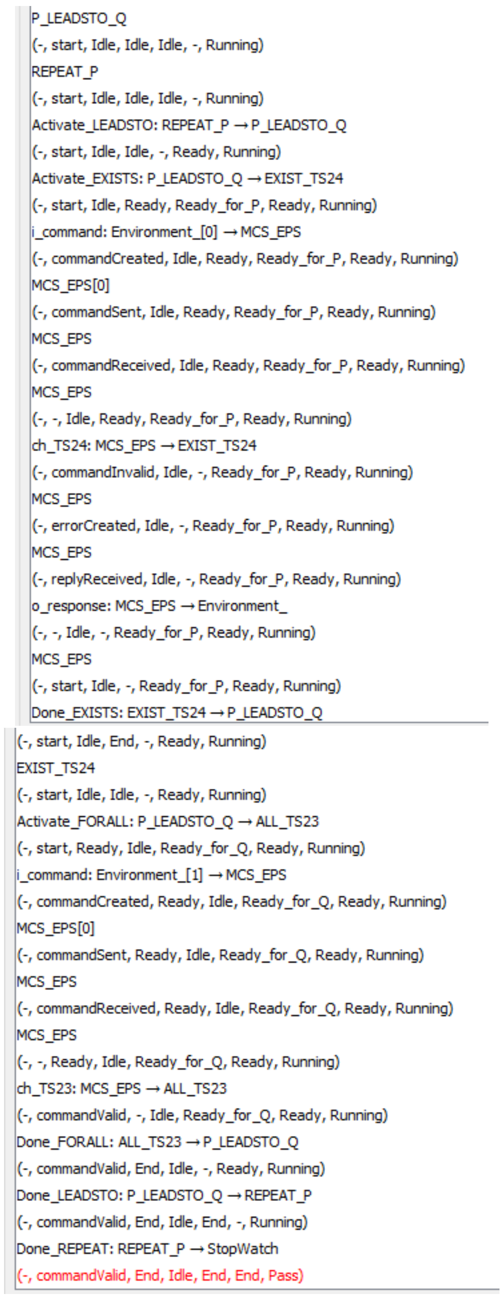
interest. TDL^{TP} featured specially with compactness and expressiveness of test purpose description. Three test cases were chosen to permute the nesting of TDL^{TP} operators to demonstrate that the formulas of TDL^{TP} have strictly more expressive that those applied in Uppal query language TCTL. Regardless the number and type of operators in the test purpose specification the Uppaal model checker generated the test traces in less than 0.2 second and using less than 8 MB residual and 29 MB virtual memory in peaks. The method of test model construction from TDL^{TP} expressions is designed with the goal to keep the interleaving of parallel sub-formula automata minimal. The performance characteristics with non-trivial test cases show that the method has a good prospective for scalability.



```
Simulation Trace
(-, start, Idle, Idle, Idle, Idle, Ready)
Activate_REPEAT: StopWatch → REPEAT_P
(-, start, Idle, Idle, Idle, -, Running)
Activate_LEADSTO: REPEAT_P → P_LEADSTO_Q
(-, start, Idle, Idle, -, Ready, Running)
Activate_EXISTS: P_LEADSTO_Q → EXIST_TS24
(-, start, Idle, Ready, Ready_for_P, Ready, Running)
i_command: Environment_[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready_for_P, Ready, Running)
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready_for_P, Ready, Running)
MCS_EPS
(-, commandReceived, Idle, Ready, Ready_for_P, Ready, Running)
MCS_EPS
(-, -, Idle, Ready, Ready_for_P, Ready, Running)
ch_TS24: MCS_EPS → EXIST_TS24
(-, commandInvalid, Idle, -, Ready_for_P, Ready, Running)
MCS_EPS
(-, errorCreated, Idle, -, Ready_for_P, Ready, Running)
MCS_EPS
(-, replyReceived, Idle, -, Ready_for_P, Ready, Running)
o_response: MCS_EPS → Environment_
(-, -, Idle, -, Ready_for_P, Ready, Running)
MCS_EPS
(-, start, Idle, -, Ready_for_P, Ready, Running)
Done_EXISTS: EXIST_TS24 → P_LEADSTO_Q
(-, start, Idle, End, -, Ready, Running)
EXIST_TS24
(-, start, Idle, Idle, -, Ready, Running)
Activate_FORALL: P_LEADSTO_Q → ALL_TS23
```



```
(-, start, Idle, Idle, -, Ready, Running)
Activate_FORALL: P_LEADSTO_Q → ALL_TS23
(-, start, Ready, Idle, Ready_for_Q, Ready, Running)
i_command: Environment_[1] → MCS_EPS
(-, commandCreated, Ready, Idle, Ready_for_Q, Ready, Running)
MCS_EPS[0]
(-, commandSent, Ready, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, commandReceived, Ready, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, -, Ready, Idle, Ready_for_Q, Ready, Running)
ch_TS23: MCS_EPS → ALL_TS23
(-, commandValid, -, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, dataCreated, -, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, replyReceived, -, Idle, Ready_for_Q, Ready, Running)
o_response: MCS_EPS → Environment_
(-, -, -, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, start, -, Idle, Ready_for_Q, Ready, Running)
Done_FORALL: ALL_TS23 → P_LEADSTO_Q
(-, start, End, Idle, -, Ready, Running)
ALL_TS23
(-, start, Idle, Idle, -, Ready, Running)
Done_LEADSTO: P_LEADSTO_Q → REPEAT_P
(-, start, Idle, Idle, End, -, Running)
```



```
P_LEADSTO_Q
(-, start, Idle, Idle, -, Running)
REPEAT_P
(-, start, Idle, Idle, -, Running)
Activate_LEADSTO: REPEAT_P → P_LEADSTO_Q
(-, start, Idle, -, Ready, Running)
Activate_EXISTS: P_LEADSTO_Q → EXIST_TS24
(-, start, Idle, Ready, Ready_for_P, Ready, Running)
i_command: Environment_[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready_for_P, Ready, Running)
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready_for_P, Ready, Running)
MCS_EPS
(-, commandReceived, Idle, Ready, Ready_for_P, Ready, Running)
MCS_EPS
(-, -, Idle, Ready, Ready_for_P, Ready, Running)
ch_TS24: MCS_EPS → EXIST_TS24
(-, commandInvalid, Idle, -, Ready_for_P, Ready, Running)
MCS_EPS
(-, errorCreated, Idle, -, Ready_for_P, Ready, Running)
MCS_EPS
(-, replyReceived, Idle, -, Ready_for_P, Ready, Running)
o_response: MCS_EPS → Environment_
(-, -, Idle, -, Ready_for_P, Ready, Running)
MCS_EPS
(-, start, Idle, -, Ready_for_P, Ready, Running)
Done_EXISTS: EXIST_TS24 → P_LEADSTO_Q
(-, start, Idle, End, -, Ready, Running)
EXIST_TS24
(-, start, Idle, Idle, -, Ready, Running)
Activate_FORALL: P_LEADSTO_Q → ALL_TS23
(-, start, Ready, Idle, Ready_for_Q, Ready, Running)
i_command: Environment_[1] → MCS_EPS
(-, commandCreated, Ready, Idle, Ready_for_Q, Ready, Running)
MCS_EPS[0]
(-, commandSent, Ready, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, commandReceived, Ready, Idle, Ready_for_Q, Ready, Running)
MCS_EPS
(-, -, Ready, Idle, Ready_for_Q, Ready, Running)
ch_TS23: MCS_EPS → ALL_TS23
(-, commandValid, -, Idle, Ready_for_Q, Ready, Running)
Done_FORALL: ALL_TS23 → P_LEADSTO_Q
(-, commandValid, End, Idle, -, Ready, Running)
Done_LEADSTO: P_LEADSTO_Q → REPEAT_P
(-, commandValid, End, Idle, End, -, Running)
Done_REPEAT: REPEAT_P → StopWatch
(-, commandValid, End, Idle, End, End, Pass)
```

Figure 34 – Fastest trace that satisfies the test purpose of the Test Case 2

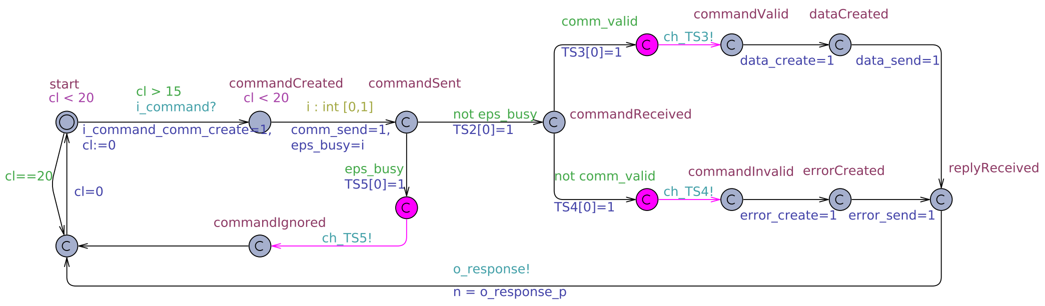
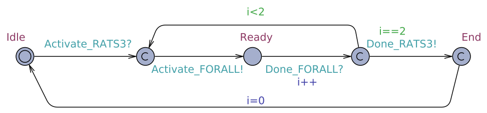
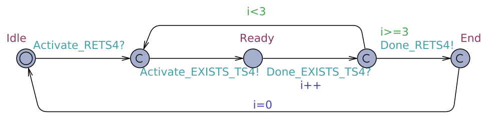


Figure 35 – Labelling of the SUT model M^{SUT} with ground level trapsets TS3, TS4 and TS5

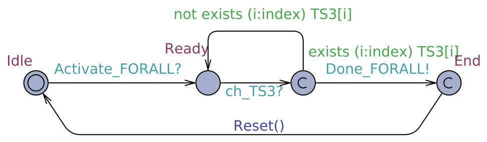


(a)

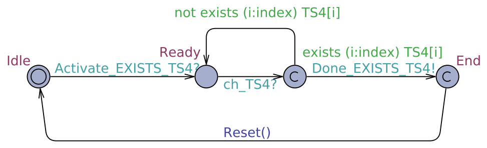


(b)

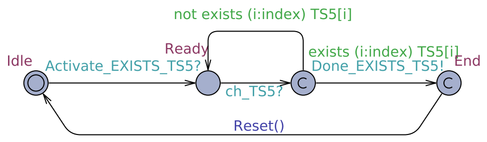
Figure 36 – Recognizing automata of sub-formulas a) $\#(A(TS3)) == 2$ b) $\#(E(TS4)) >= 3$



(a) $A(TS3)$



(b) $E(TS4)$



(c) $E(TS5)$

Figure 37 – Recognizing automata of sub-formulas a) $A(TS3)$, b) $E(TS4)$ and c) $E(TS5)$

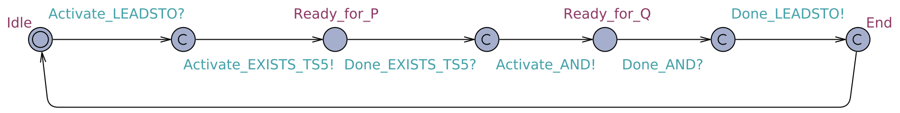


Figure 38 – Recognizing automaton of the scenario specification root formula $E(TS5) \sim> (\#(A(TS3)) == 2 \wedge \#(E(TS4)) >= 3)$

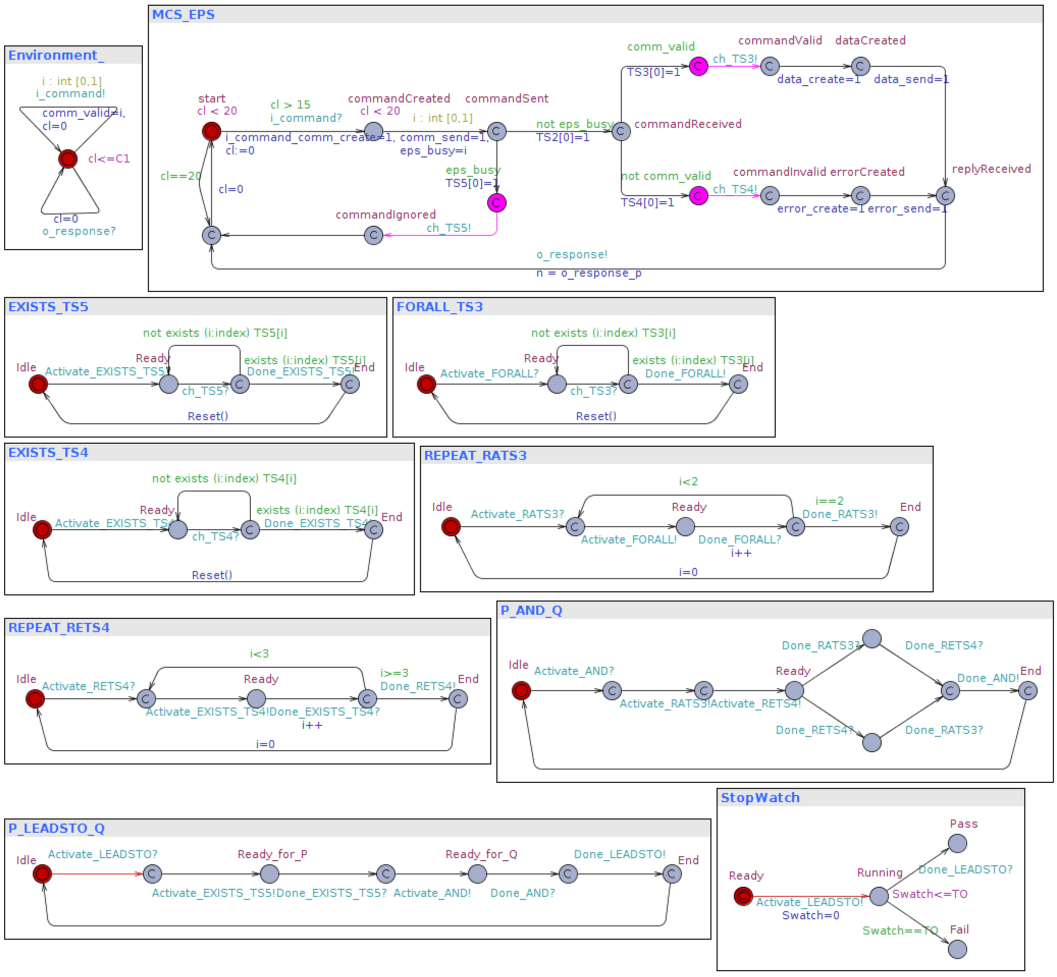
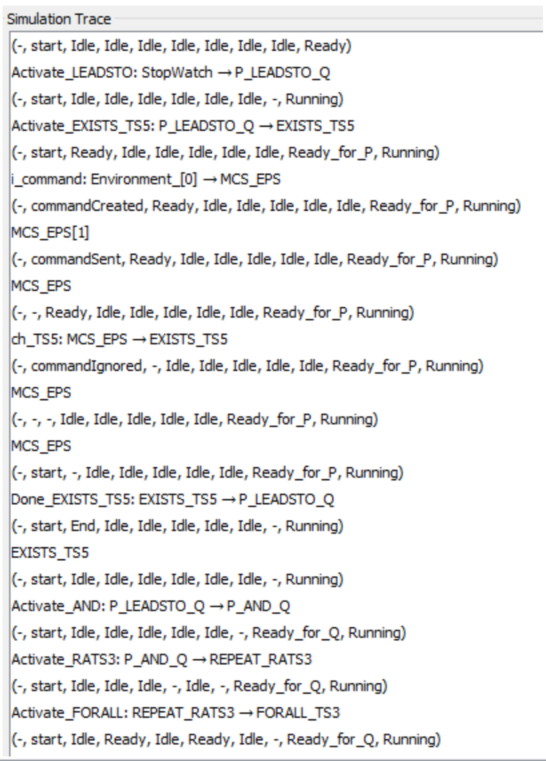
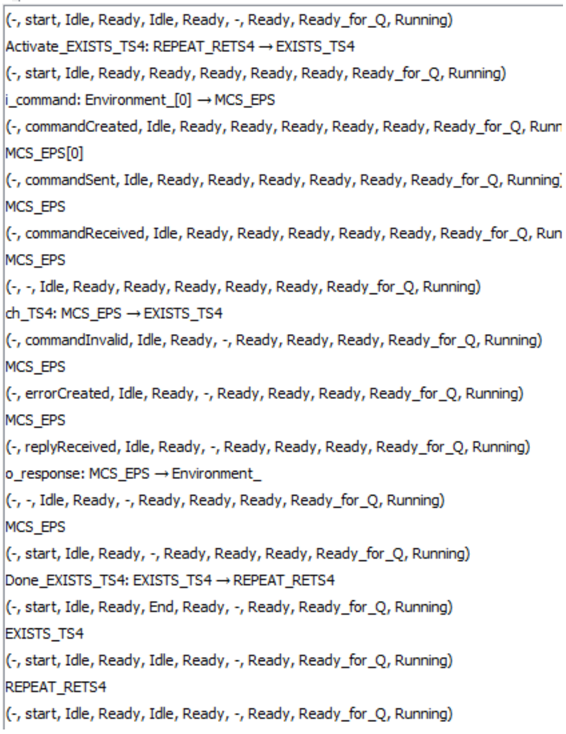


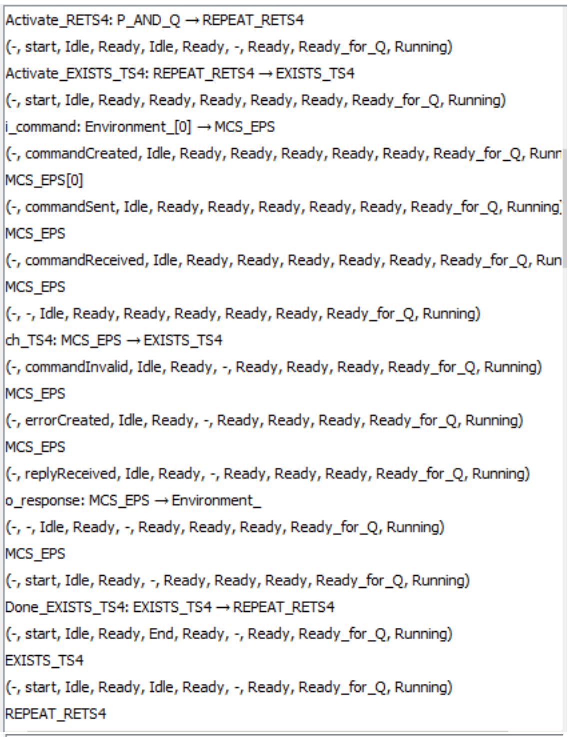
Figure 39 – Full test model for the Test Case 3 scenario $E(TS5) \sim> (\#(A(TS3)) == 2 \wedge \#(E(TS4)) >= 3)$



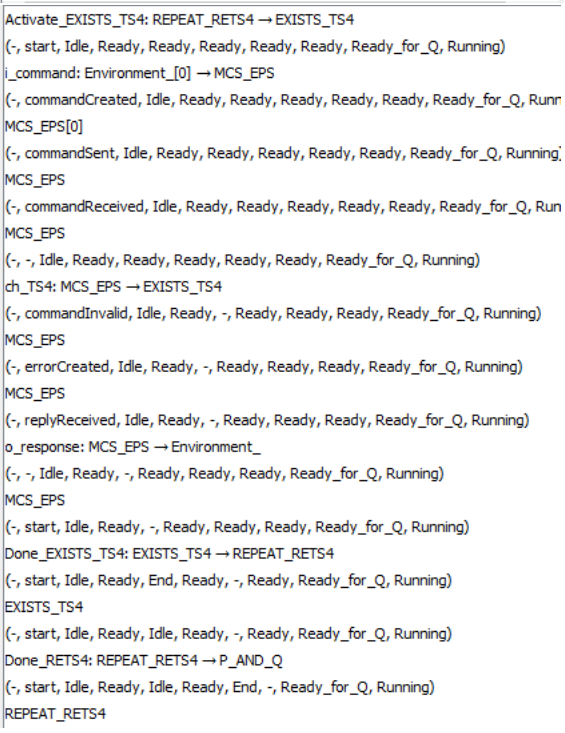
```
Simulation Trace
(-, start, Idle, Idle, Idle, Idle, Idle, Idle, Idle, Ready)
Activate_LEADSTO: StopWatch → P_LEADSTO_Q
(-, start, Idle, Idle, Idle, Idle, Idle, Idle, -, Running)
Activate_EXISTS_TS5: P_LEADSTO_Q → EXISTS_TS5
(-, start, Ready, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
i_command: Environment_[0] → MCS_EPS
(-, commandCreated, Ready, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
MCS_EPS[1]
(-, commandSent, Ready, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
MCS_EPS
(-, -, Ready, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
ch_TS5: MCS_EPS → EXISTS_TS5
(-, commandIgnored, -, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
MCS_EPS
(-, -, -, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
MCS_EPS
(-, start, -, Idle, Idle, Idle, Idle, Idle, Ready_for_P, Running)
Done_EXISTS_TS5: EXISTS_TS5 → P_LEADSTO_Q
(-, start, End, Idle, Idle, Idle, Idle, Idle, -, Running)
EXISTS_TS5
(-, start, Idle, Idle, Idle, Idle, Idle, Idle, -, Running)
Activate_AND: P_LEADSTO_Q → P_AND_Q
(-, start, Idle, Idle, Idle, Idle, Idle, -, Ready_for_Q, Running)
Activate_RATS3: P_AND_Q → REPEAT_RATS3
(-, start, Idle, Idle, Idle, -, Idle, -, Ready_for_Q, Running)
Activate_FORALL: REPEAT_RATS3 → FORALL_TS3
(-, start, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
```



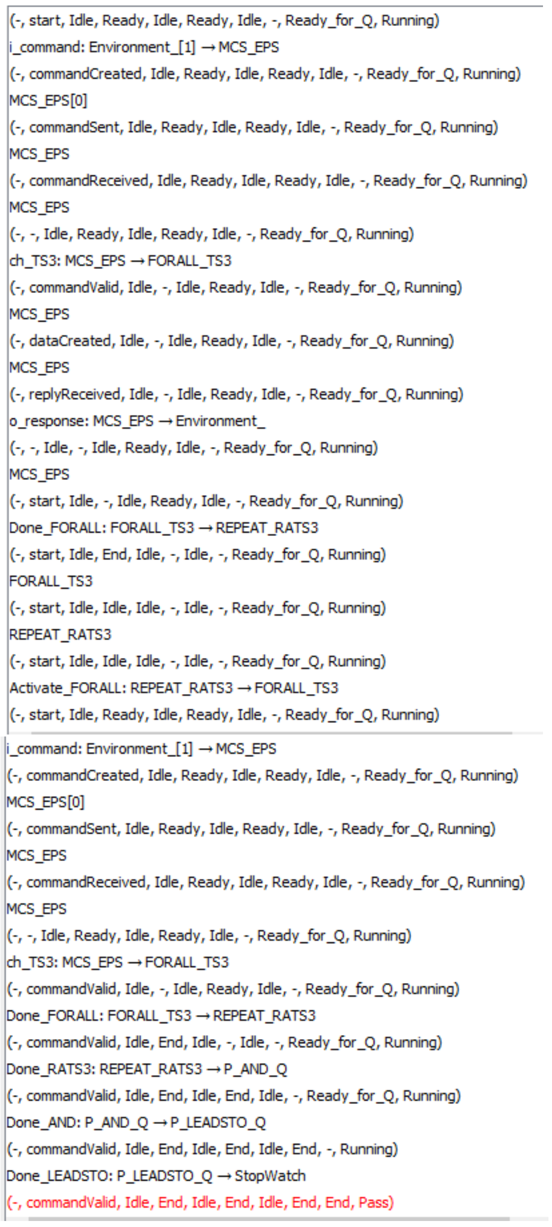
```
(-, start, Idle, Ready, Idle, Ready, -, Ready, Ready_for_Q, Running)
Activate_EXISTS_TS4: REPEAT_RETS4 → EXISTS_TS4
(-, start, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running)
i_command: Environment_[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Runr
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running
MCS_EPS
(-, commandReceived, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Run
MCS_EPS
(-, -, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running)
ch_TS4: MCS_EPS → EXISTS_TS4
(-, commandInvalid, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, errorCreated, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, replyReceived, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
o_response: MCS_EPS → Environment_
(-, -, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, start, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
Done_EXISTS_TS4: EXISTS_TS4 → REPEAT_RETS4
(-, start, Idle, Ready, End, Ready, -, Ready, Ready_for_Q, Running)
EXISTS_TS4
(-, start, Idle, Ready, Idle, Ready, -, Ready, Ready_for_Q, Running)
REPEAT_RETS4
(-, start, Idle, Ready, Idle, Ready, -, Ready, Ready_for_Q, Running)
```



```
Activate_RETS4: P_AND_Q → REPEAT_RETS4
(-, start, Idle, Ready, Idle, Ready, -, Ready, Ready_for_Q, Running)
Activate_EXISTS_TS4: REPEAT_RETS4 → EXISTS_TS4
(-, start, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running)
i_command: Environment_[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Runr
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running]
MCS_EPS
(-, commandReceived, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Run
MCS_EPS
(-, -, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running)
ch_TS4: MCS_EPS → EXISTS_TS4
(-, commandInvalid, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, errorCreated, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, replyReceived, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
o_response: MCS_EPS → Environment_
(-, -, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, start, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
Done_EXISTS_TS4: EXISTS_TS4 → REPEAT_RETS4
(-, start, Idle, Ready, End, Ready, -, Ready, Ready_for_Q, Running)
EXISTS_TS4
(-, start, Idle, Ready, Idle, Ready, -, Ready, Ready_for_Q, Running)
REPEAT_RETS4
```



```
Activate_EXISTS_TS4: REPEAT_RETS4 → EXISTS_TS4
(-, start, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running)
i_command: Environment_[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Runr
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running]
MCS_EPS
(-, commandReceived, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Run
MCS_EPS
(-, -, Idle, Ready, Ready, Ready, Ready, Ready, Ready_for_Q, Running)
ch_TS4: MCS_EPS → EXISTS_TS4
(-, commandInvalid, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, errorCreated, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, replyReceived, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
o_response: MCS_EPS → Environment_
(-, -, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
MCS_EPS
(-, start, Idle, Ready, -, Ready, Ready, Ready, Ready_for_Q, Running)
Done_EXISTS_TS4: EXISTS_TS4 → REPEAT_RETS4
(-, start, Idle, Ready, End, Ready, -, Ready, Ready_for_Q, Running)
EXISTS_TS4
(-, start, Idle, Ready, Idle, Ready, -, Ready, Ready_for_Q, Running)
Done_RETS4: REPEAT_RETS4 → P_AND_Q
(-, start, Idle, Ready, Idle, Ready, End, -, Ready_for_Q, Running)
REPEAT_RETS4
```



```
(-, start, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
i_command: Environment_[1] → MCS_EPS
(-, commandCreated, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS[0]
(-, commandSent, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, commandReceived, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, -, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
ch_TS3: MCS_EPS → FORALL_TS3
(-, commandValid, Idle, -, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, dataCreated, Idle, -, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, replyReceived, Idle, -, Idle, Ready, Idle, -, Ready_for_Q, Running)
o_response: MCS_EPS → Environment_
(-, -, Idle, -, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, start, Idle, -, Idle, Ready, Idle, -, Ready_for_Q, Running)
Done_FORALL: FORALL_TS3 → REPEAT_RATS3
(-, start, Idle, End, Idle, -, Idle, -, Ready_for_Q, Running)
FORALL_TS3
(-, start, Idle, Idle, Idle, -, Idle, -, Ready_for_Q, Running)
REPEAT_RATS3
(-, start, Idle, Idle, Idle, -, Idle, -, Ready_for_Q, Running)
Activate_FORALL: REPEAT_RATS3 → FORALL_TS3
(-, start, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
i_command: Environment_[1] → MCS_EPS
(-, commandCreated, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS[0]
(-, commandSent, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, commandReceived, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
MCS_EPS
(-, -, Idle, Ready, Idle, Ready, Idle, -, Ready_for_Q, Running)
ch_TS3: MCS_EPS → FORALL_TS3
(-, commandValid, Idle, -, Idle, Ready, Idle, -, Ready_for_Q, Running)
Done_FORALL: FORALL_TS3 → REPEAT_RATS3
(-, commandValid, Idle, End, Idle, -, Idle, -, Ready_for_Q, Running)
Done_RATS3: REPEAT_RATS3 → P_AND_Q
(-, commandValid, Idle, End, Idle, End, Idle, -, Ready_for_Q, Running)
Done_AND: P_AND_Q → P_LEADSTO_Q
(-, commandValid, Idle, End, Idle, End, Idle, End, -, Running)
Done_LEADSTO: P_LEADSTO_Q → StopWatch
(-, commandValid, Idle, End, Idle, End, Idle, End, End, Pass)
```

Figure 40 – Test trace of the Test Case3 generated with model checking option "some"

CONCLUSIONS

The study of related work on model-based testing and particularly in the domain of test purpose specification methods shows that low software quality is mainly due to the *problematical test coverage and incorrect requirements*. Approximately, a half of incorrect requirements are caused by incomplete specification and another half of cases by *unclear* and *ambiguous* requirements.

To address this problem, the research of the thesis is focused on model-based testing, specifically, on the test purpose specification and test generation technique to address the test coverage and faults back-traceability problems. The problem is addressed from three perspectives:

- The *application domain* that dictates the needs and constraints on the model-based testing method;
- The model-based *testing technology* to meet these needs;
- The *formal framework* used to automatize the test purpose specification and test generation procedures.

The thesis is oriented to applications that require extensive test effort, i.e. the systems that integrate typically many functions onto one while ensuring the safe segregation of functions with different criticality levels. Common features to be addressed when testing the use cases such as satellite mission software are: long communication delays, security vulnerabilities, functional interference between software components, non-determinism regarding events timing, varying control and data transmission capabilities. These features expose most clearly in *Integration* and *System* level testing where the functionality, timing, safety, security and other aspects of MCS are inspected in their most entangled form. The analysis of the related work show that current test purpose specification languages focus on certain groups of test coverage criteria and are supporting less their integration in multi-coverage criteria.

From this point of view, the thesis task was set to develop an *expressive test purpose specification language* and the method of *extracting complex test cases* from SUT models. The requirements to the specification language could be summarised as follows: it should express the coverage criteria important in the application domain, be correct, which means that they should not signal errors in correct implementations, should be *meaningful*, i.e. erroneous implementations should be detected with high probability. To address the problems of complexity and traceability in MBT the thesis extends the model-based conformance testing with a *scenario based test description language* TDL^{TP} and an automatic *test generation technique* that is based on this language.

Since the theory of timed automata and its extension Uppal timed automata theory satisfy the criteria of the modelling language for critical systems, the thesis relies on the underlying theory of Uppaal TA and related UPPAAL tool family. Although this tool family has means for model checking, controller synthesis, and test execution, its property specification language TCTL is used with substantial limited form, namely, the nesting of temporal operators is not supported in the tools. This makes proving more complex properties and generating tests from specifications that include several temporal constraint difficult. It can be done using extra property automata but it requires deep knowledge in Uppaal TA semantics and is error prone even for experts.

As an extension to TCTL based test purpose specification language, the thesis built an extra language layer (Test Purpose Definition Language - TDL^{TP}) for test scenario specification that is expressive, free from the limitations of 'flat' TCTL, is interpretable in Uppaal TA, and thus, suited for automatic test generation using Uppaal model checker.

The benefits of TDL^{TP} based test purpose specification and test generation can be summarized as follows:

- Due to its high expressive power the representation of test scenarios in TDL^{TP} is more compact compared to that of TCTL;
- Formal semantics of TDL^{TP} expressions enables
  - formal correctness verification of test models and test purpose specifications, incl. evaluation of their feasibility, time/space complexity (the statistics of model checking are exposed as part of model checking results);
  - automated generation of tests from verified models;
  - interpretation of different coverage criteria and back-tracing the root causes of found bugs.
- The TDL^{TP} expressions can be interrelated with other test coverage criteria and coverage metrics like structural coverage, function coverage, requirement coverage etc.

The main results and novelties of the thesis in the field of model based testing can be concluded as follows:

1. A highly expressive test purpose specification language TDL^{TP} for complex test scenario specifications that is needed for MBT in safety and time critical systems is defined. The test purpose specification language TDL^{TP} syntax and formal semantics are introduced. The mapping is used to automatize the construction of test models from the model of SUT and the test purpose expressions in TDL^{TP}.

2. The operational semantics of TDL^{TP} that is defined in terms of model transformation rules that map declarative TDL^{TP} expressions to executable test models represented as Uppaal timed automata is defined. To remove the possible overhead in TDL^{TP} expressions and in the test models we introduced a set of TDL^{TP} simplification rules to keep these specifications concise and readable, and also reduce the size of the generated test models.

3. A provably correct test development process description is introduced and correctness conditions specified to be verified when showing correctness of test development steps. This work has been motivated by the need to increase the trust on testing results and to avoid running infeasible tests or tests that could lead to incorrect conclusions. Secondly, to reduce the test development time and to detect the test development faults in earliest possible phases the proposed approach enables verifying each intermediate test development product whenever it is available not just waiting for the end of development process when full implementation is available. The verification conditions and technique provided are relatively independent from the specifics of development method. This makes the verification approach customizable to different development process models and modeling formalisms. Another advantage of the approach is that it does not focus on functional properties only. The correctness verification steps enable to prove also the correctness aspects of mixed critical SUT where timing and data constraints both are substantial.

4. The theoretical results of the thesis are validated on the TUT100 satellite software testing case study. Our experiments with three test cases on a SUT TUT100 Nanosatellite Electrical Power Supply subsystem show that the test purpose specification language TDL^{TP} is applicable when complex test scenarios with multiple coverage criteria are of interest. TDL^{TP} features especially with compactness and expressiveness of test purpose descriptions. Three test cases were chosen to permute the nesting of TDL^{TP} operators to demonstrate that the formulas of TDL^{TP} have strictly more expressive power that those applied in Uppal query language TCTL. Regardless the number and type of operators in the test purpose specification the Uppaal model checker generated the test traces without any difficulty. The method of test model construction from TDL^{TP} expressions keeps the interleaving of parallel sub-formula automata minimal.

Future work

As for the further extension of the research highlighted in the thesis we can outline at least three possible directions:

Integration of TDL^{TP} with reactive planning tester (RPT) generation method for online testing of non-deterministic systems means that the controllable (by test) part of the environment model of SUT should be substituted with one or many reactive planning testers which guarantee the efficient reachability of test coverage items - trapsets. For integration the control of RPT instances has to be added to the test automata elaborated in the thesis.

Design by contract is gaining popularity in provably correct development research community. In particular, the multi viewpoint contract interference issues are under study. Generating the tests directly from viewpoint contracts would reduce the effort if specifying test purposes simultaneously with design specification by system developers. Declarative TDL^{TP} provides possibilities for specifying behavioral properties on high level of abstraction by both parties so that the contracts for different design views can be specified and same TDL^{TP} expressions used as test purpose specifications.

Though current thesis address the problems of conformance testing the usage of TDL^{TP} can be studied also for mutation testing, especially, to express the mutations symbolically.

The scalability of the test purpose specification using TDL^{TP} and test generation has shown promising results when applying it on TUT100 integration testing. The scalability of the approach can be studied even on larger SUT models, e.g. for acceptance tests of the TUT100 satellite system after its full release is available.

List of Figures

(if needed)

References

[1] D1.6.1 - meta-models for platform-specific modelling, dreams consortium, 5/2016.

[2] Etsi es 202 553: Methods for testing and specification (mts). TPLan: A notation for expressing Test Purposes, v1.2.1. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, June 2009.

[3] ETSI ES 202 951 Methods for Testing and Specification (MTS); Model-Based Testing (MBT). Available: `http://www.etsi.org/`. Requirements for Modelling Notations (July 2011).

[4] ETSI Test Description Language (TDL). Available: `http://www.etsi.org/`.

[5] Eu speeds project. inria research report n. 8147 november 2012. [Online]. Available: `https://hal-univ-tlse2.archives-ouvertes.fr/hal-01178467/document`. (Last accessed: Jan. 20 2019).

[6] Eu speeds project. inria research report n. 8147 november 2012. [Online]. Available: `https://hal-univ-tlse2.archives-ouvertes.fr/hal-01178467/document`. (Last accessed: Jan. 20 2019).

[7] International telecommunication union (itu): Recommendation z.120. Available: `http://www.itu.int/rec/T-REC-Z.120-199804-I!AnnB/en`. Annex B: Formal Semantics of Message Sequence Chart (MSC), 04/98. Online: Z.120 Annex B (04/98), Standard document.

[8] International telecommunication union (itu): Recommendation z.120: Message sequence chart (msc), 02/11. online: Z.120 (02/11), standard document. Available: `http://www.itu.int/rec/T-REC-Z.120-201102-I/en`.

[9] Iso: Road vehicles - open test sequence exchange format - part 3: Standard extensions and requirements. International ISO multipart standard No. 13209–3 (2012).

[10] Iso/iec: Information technology - open systems interconnection - conformance testing methodology and framework - part 1: General concepts. International ISO/IEC multipart standard No. 9646–1 (1994–1998).

[11] A. Anier and J. Vain. Model based continual planning and control for assistive robots. *HealthInf 2012. Vilamoura, Portugal*, 2012.

[12] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[13] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[14] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the 3rd Workshop on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007 International Symposium on Software Testing and Analysis, London, United Kingdom, July 9-12*, pages 95–104. ACM, 2007.

[15] E. Brinksma. A formal approach to conformance testing. *Proceedings of International Workshop on Protocol Test Systems*, pages 311–325, 1989.

[16] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen. Uppaal SMC tutorial. *STTT*, 17(4):397–415, 2015.

[17] A. David, K. G. Larsen, S. Li, and B. Nielsen. A game-theoretic approach to real-time system testing. In D. Sciuto, editor, *Design, Automation and Test in Europe, DATE 2008, Munich, Germany, March 10-14, 2008*, pages 486–491. ACM, 2008.

[18] A. David, K. G. Larsen, M. Mikucionis, O. Nguena-Timo, and A. Rollet. Remote testing of timed specifications. In H. Yenigün, C. Yilmaz, and A. Ulrich, editors, *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*, volume 8254 of *Lecture Notes in Computer Science*, pages 65–81. Springer, 2013.

[19] J. P. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ROS packages: A mobile robot case study. In *2015 European Conference on Mobile Robots, ECMR 2015, Lincoln, United Kingdom, September 2-4, 2015*, pages 1–7. IEEE, 2015.

[20] M. P. et al. Evolving the etsi test description language. in: Grabowski j., herbold s. (eds) system analysis and modeling. technology-specific aspects of models. sam 2016. lecture notes in computer science, vol 9959. springer.

[21] J. Grossmann and W. Müller. A formal behavioral semantics for testml. *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2006*, pages 441–448, 2006.

[22] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, Y. Fusero, and M. Schieber. A meta-model for tests of avionics embedded systems. In S. Hammoudi, L. F. Pires, J. Filipe, and R. C. das Neves, editors, *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*, pages 5–13. SciTePress, 2013.

[23] E. Halling, J. Vain, A. Boyarchuk, and O. Illiashenko. Test scenario specification language for model-based testing. *International Journal of Computing*, 2019.

[24] G. Hamon, L. M. de Moura, and J. M. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, pages 261–270. IEEE Computer Society, 2004.

[25] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[26] B. Hunt, T. Abolfotouh, and J. Carpenter. Software test costs and return on investment (roi) issues. 2014.

[27] M. Kääramees. *A symbolic Approach to Model-Based Online Testing*. PhD thesis, PhD Thesis, Tallinn University of Technology, Tallinn, 2012.

[28] E. A. Lee. Cyber-physical systems-are computing foundations adequate, in position paper for nsf workshop on cyber-physical systems: Re-search motivation, techniques and roadmap, vol. 2, 2006.

[29] J. Musa. Software reliability engineering. *AuthorHouse, 2nd ed.*, 2004.

[30] A. C. D. Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. in proceedings of the 1st acm international workshop on empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd ieee/acm international conference on automated software engineering (ase) 2007 (weaseltech '07). acm, new york, ny, usa, 31-36.

[31] OMG. Object management group. ccdl whitepaper. Technical report, Razorcat Technical Report, 2014.

[32] T. Pajunen, T. Takala, and M. Katara. Model-based testing with a general purpose keyword-driven test automation framework. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 242–251. IEEE Computer Society, 2011.

[33] F. Siavashi, J. Iqbal, D. Truscan, and J. Vain. Testing web services with model-based mutation. *Communications in Computer and Information Science*, 743:45–67, 2017.

[34] P. Skokovic. Requirements-based testing process in practice. IJIEM, Vol.1 No 4, 2010, pp. 155-161.

[35] J. Tretmans. Model based testing. Available: `http://slideplayer.com/slide/5082174/`.

[36] J. Tretmans. A formal approach to conformance testing. In O. Rafiq, editor, *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993*, volume C-19 of *IFIP Transactions*, pages 257–276. North-Holland, 1993.

[37] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[38] W. Tsai, A. Saimi, L. Yu, and R. A. Paul. Scenario-based object-oriented testing framework. In *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*, page 410. IEEE Computer Society, 2003.

[39] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012.

[40] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012.

[41] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test., Verif. Reliab.*, 22(5):297–312, 2012.

[42] J. Vain, A. Anier, and E. Halling. Provably correct test development for timed systems. In H. Haav, A. Kalja, and T. Robal, editors, *Databases and Information Systems VIII - Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014, 8-11 June 2014, Tallinn, Estonia*, volume 270 of *Frontiers in Artificial Intelligence and Applications*, pages 289–302. IOS Press, 2014.

[43] 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, 8-11 June 2014, Tallinn, Estonia*, pages 289–302, 2014.

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

[45] J. Vain and E. Halling. Constraint-based testing scenario description language. *Proceedings of 13th Biennial Baltic Electronics Conference, BEC 2012*, IEEE:89–92, 2012.

[46] J. Vain, E. Halling, G. Kanter, A. Anier, and D. Pal. Model-based testing of real-time distributed systems. In G. Arnicans, V. Arnicane, J. Borzovs, and L. Niedrite, editors, *Databases and Information Systems - 12th International Baltic Conference, DB&IS 2016, Riga, Latvia, July 4-6, 2016, Proceedings*, volume 615 of *Communications in Computer and Information Science*, pages 272–286. Springer, 2016.

[47] J. Vain, M. Kääramees, and M. Markvardt. Online testing of nondeterministic systems with the reactive planning tester. In *Dependability and Computer Engineering: Concepts for Software-Intensive Systems. IGI Global*, pages 113–150, 2012.

[48] J. Vain, K. Raiend, A. Kull, and J. P. Ernits. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 363–372. ACM, 2007.

[49] G. H. Walton and J. H. Poore. Generating transition probabilities to support model-based software testing. *Softw., Pract. Exper.*, 30(10):1095–1106, 2000.

[50] J. Wittevrongel and F. Maurer. SCENTOR: scenario-based testing of e-business applications. In *10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2001), 20-22 June 2001, Cambridge, MA, USA*, pages 41–48. IEEE Computer Society, 2001.

Acknowledgements

First of all, my sincere gratitude goes to my supervisor Professor Jüri Vain for his support and guidance. Without him this thesis would never have been completed.

A special word of thanks goes also to my colleagues at the Tallinn University of Technology for their encouragement and mental support.

This work was also partially supported by:

- Nations Support Program for the ICT in Higher Education "Tiger University",
- the Compentence Centre programme of Enterprise Estonia and the European Union trough the European Regional Development Fund.

European Union
European Regional
Development Fund

Investing
in your future

Abstract
Scenario Oriented Model-Based Testing

Mission Critical Systems (MCS) are systems whose failure might cause catastrophic consequences, such as someone dying, damage to property, severe financial losses, damage to national security and others. A well-designed MCS, even in case of failures, if properly predicted, timely detected and recovered, should be able to operate under severe exploitation conditions without catastrophic consequences.

Detection of software bugs, especially those deeply nested in software loops which manifest sporadically as wrong timing in complex time critical systems, is a real challenge for current MCS software engineering methods. The methods of risk mitigation, in particular the provably correct software synthesis, formal verification as well as model-based testing, are powerful but time and computationally expensive which limits their wider application in practice.

According to Inria research report n° 8147, in *automotive* and *medical* domain the *system integration level test* and *verification* cause project delays respectively in 63% and in 66,7% of cases. It gives indication that the *software integration level test* and *verification* are the *main bottlenecks* where new verification and test development methods and their tooling are of key importance. Model-based testing is considered to be one of the promising validation method to address the complex verification use cases.

In this thesis, the research is focused on model-based testing, specifically, on the test purpose specification and test generation techniques to address the test coverage and faults back-traceability problems. The scope of thesis is defined from three interrelated perspectives:

- The *application domain* that dictates needs and constraints on the testing approach;
- The *testing technology* applied to meet these needs;
- The *formal framework* used to automatize the test purpose specification and generation procedures.

The MCSs that require extensive test effort are typically applications that integrate many functions onto one while ensuring the safe segregation of functions with different criticality levels. These systems are called also mixed-criticality systems. Examples of such systems are surgical robots and assisting robots used in medical treatment procedures as well as spacecrafts having long term autonomous missions.

Common features to be addressed when testing the mixed critical MCSs such as satellites and distributed robotic systems are: significantly longer communication delays compared to that of local computations, security vulnerabilities, functional interference between software components, non-determinism regarding events' timing, varying control and data transmission capabilities, etc.

The thesis scope is model-based conformance testing of MCS. The aim is to develop an *expressive test purpose specification language* and the *method of extracting complex test cases* from SUT models. The derived tests should satisfy the multiple coverage criteria specified in the test purpose, be provably correct, which means that they should not signal errors in correct implementations, should be *meaningful*, i.e. erroneous implementations should be detected and traced back to the requirements. To address the problems of complexity and traceability in MBT the thesis extends the model-based conformance testing approach with a *scenario based test description language* and a related to that *automatic test generation technique*.

MBT relies on formal models. The models are built from the requirements or design specifications in order to describe the expected behaviour of SUT in interaction with its environment. The model should be precise, unambiguous, and presented in the way that is relevant for correctness verification and test generation. The thesis relies on the underlying theory of Uppaal Timed Automata (Uppaal TA) and related UPPAAL tool family (www.uppaal.org) that supports modelling, validation and verification of real-time systems.

As a result of related work analysis it is concluded that there is a need for more expressive test purpose description language (it is called in thesis TDL^{TP}) than currently available, to allow compact, multi-criterial test coverage specifications, and which is formally verifiable and efficiently implementable for test generation.

The technical tasks of the thesis to be solved for these goals are the following:

- Defining the syntax and formal semantics of test purpose specification language TDL^{TP};
- Defining the interpretation of TDL^{TP} in terms of language recognizing Uppaal TA;
- Designing and implementing the interpreter of TDL^{TP} and based on that a symbolic test generator;
- Integrating the TDL^{TP} usage into provably correct testing workflow.
- Demonstrating the feasibility of TDL^{TP} usage on a practical non-trivial test purpose specification and test generation case study.

The main contribution of the thesis achieved in the field of model based testing can be summarized as follows:

1. The thesis defines a highly expressive test purpose specification language TDL^{TP} for specifying complex test scenarios that are needed for MBT of complex MCSs.
2. The syntax and semantics of TDL^{TP} operators are defined and the transformation rules that map declarative TDL^{TP} expressions to executable test models represented as Uppaal TA.
3. A provably correct test development process model is introduced and proof obligations specified to be verified when showing correctness of test development steps.
4. The validation of thesis theoretical results has been done on the TUT100 satellite software case study.

As for the further extension of the research highlighted in the thesis following possible directions are outlined:

Integration of TDL^{TP} with reactive planning tester synthesis method for online testing of non-deterministic MCS. The reactive planning tester synthesis method has been suggested by the Autonomy and Software Technology group at *NASA's Deep Space One* programme but its public version is limited with FSM models only. The use of TDL^{TP} would enable to extend it to more expressive Uppaal TA models.

Second direction could be related to generating tests from multi-viewpoint software contracts to reduce the effort of specifying test purposes simultaneously with design specifications by system developers. Declarative TDL^{TP} could provide the possibilities for specifying behavioral properties on high level of abstraction by both parties so that the contracts for different design views can be specified and same TDL^{TP} expressions used as test purpose specifications.

Though the current thesis address the problems of conformance testing the usage of TDL^{TP} can be studied also for mutation testing, especially, to express the mutations symbolically.

The scalability of the test purpose specification using TDL^{TP} and test generation has shown promising results when applying it on TUT100 satellite software integration testing. The scalability of the approach can be studied even on larger SUT models, e.g. for acceptance tests of the TUT100 satellite system after its full release is available.

Kokkuvõte
Stsenaariumjuhitud mudelipõhine testimine

Missioonikriitilised süsteemid (MKS) on süsteemid, mille tõrked ja vead võivad põhjustada katastroofilisi tagajärgi nagu näiteks seada ohtu inimelusid, tekitada varalist ja rahalist kahju, ohustada rahvuslikku julgeolekut jms. Õigesti projekteeritud MKS suudab töötada isegi karmides ekspluatatsioonitingimustes ilma fataalsete tõrgeteta eeldusel, et nende tõrgetega on projekteerimisel arvestatud, tõrked on õigeaegselt avastatud ja kompenseeritud.

Tarkvaravigade avastamine eriti juhul, kui need vead asuvad sügavates programmitsüklites ning avalduvad ajakriitilistes süsteemides juhuslikel hetkedel sündmuste vale ajastusena, on kaasagsetele tarkvara arendusmeetoditele endiselt tõsiseks probleemiks. Tarkvara vigadega seotud riskide vähendamise meetodid nagu tõestatavalt korrektse tarkvara süntees, formaalne verifitseerimine ja mudeli-põhine testimine on küll võimsad, kuid nii aja- kui arvutusressursi mahukad, mis omakorda piirab nende laiemat kasutust tarkvaratehnika igapäeva praktikas.

Inria uuringuaruande nr. 8147 kohaselt põhjustab plaanivälist hilistumist projektides 63% juhul autotööstuses ja 66,7% juhtudest meditsiinitehnikas süsteemide integratsiooni testimine ja verifitseerimine. See fakt näitab, et tarkvara integratsiooni testimine ja verifitseerimine on kogu arendusprotsessi kitsaskohtadeks ja nende automatiseerimine omab arenduse efektiivsemaks muutmisel võtmerolli. Mudeli-põhine testimine ja selle automatiseerimine on üks enam lubavaid valideerimistehnikaid keerukate verifitseerimisülesannete lahendamisel.

Käesoleva väitekirja uurimisobjektiks on mudeli-põhine testimine, täpsemalt testi eesmärgi spetsifitseerimine ja testide genereerimine saavutamaks paremat testikatvust ja vigade põhjuste diagnoositavust. Väitekirja skoop on määratletud kolmest aspektist lähtuvalt:

- Testimistehnoloogia rakendusvaldkond, mis määrab nõuded ja kitsendused testimistehnikale;
- Testimistehnika ise mis peab vastama eeltoodu nõuetele ja kitsendustele;
- Formaalne aparatuur, mis on rakendatav testi eesmärkide spetsifitseerimise ja testi genereerimise automatiseerimisel.

MKS-d, mis nõuavad mahukat testimist, on tüüpiliselt rakendused, mis integreerivad erinevaid funktsionaalsusi, mille puhul on nõutav nende koostöö ohutus erinevatel kriitilisusastmetel. Niisugusi süsteeme nimetatakse põimkriitilisteks (mixed-critical). Niisuguste süsteemide näiteks on kirurgias ja meditsiinilistes raviprotseduurides kasutatavad robotid, samuti pikaaegsetel autonoomsetel missioonidel kasutatavad kosmoseaparaadid. Niisuguste intensiivset testimist vajavate süsteemide ühisteks tunnusteks on: kommunikatsiooniga seotud hilistumised on oluliselt suuremad kui arvutusprotsessidega seotud hilistumised, turvalisusega seotud probleemid, komponentide funktsionaalsuse ristmõjutused, sündmuste ajastuse mitte-determinism, juhtimis- ja andmeedastusfunktsioonide varieeruv jõudlus jne.

Käesolev väitekiri keskendub MKS-de mudeli-põhisele konformsustestimisele. Väitekirja eesmärgiks on luua suure väljendusvõimsusega testieesmärkide spetsifitseerimise keel ja tehnika testitava süsteemi mudelist ning testieesmärgi kirjeldusest testide genereerimiseks. Loodav keel ja genereeritavad testid peavad võimaldama väljendada ja kontrollida erinevaid testikatte kriteeriume, olema tõestatavalt korrektsed st signaliseerima vigadest

ainult tõeliste vigade korral, võimaldama tuvastada vigade põhjusi ja lokaliseerida disaini nõuded, mida avastatud vead puudutavad.

Mudeli-põhine testimine põhineb formaalsete mudelite olemasolul. Mudelid konstrueeritakse süsteemi nõuete spetsifikatsioonist ja need kirjeldavad interaktsioone süsteemi ja tema keskkonna vahel. Mudelite puhul eeldatakse, et need on ühemõtteliselt mõistetavad ja kasutatavad nii nõuete korrektsuse verifitseermiseks kui testide genereerimiseks. Väitekirjas on valitud selleks Uppaali Ajaga Automaatide (Uppaal TA) teooria ja sellega seotud tööriistade kogu (www.uppaal, org), mis toetab reaalajasüsteemide modelleerimist, valideerimist ja verifitseerimist.

Töökäigus läbiviidud kirjanduse analüüs näitab, et teadaolevad testieemärgi spetsifitseerimise keeled ei ole piisava väljendusvõimsusega või kui on, siis nad ei toeta testide automaatset genereerimist. Alternatiivina väitekirjas loodava testieesmärgi kirjeldamise keele TDLTP puhul on seatud eesmärgiks esitada korraga mitut testikattekriteeriumit, tagada verifitseeritavus ja efektiivne rakendatavus testide genereerimisel.

Keele loomisega seotud konkreetsemad tehnilised ülesanded on seatud järgmiselt:

- Defineerida testieesmärgi kirjeldamise keele TDL^{TP} süntaks ja formaalne semantika;
- Defineerida TDL^{TP} termide interpretatsioon termide poolt kirjeldatud käitumiste tuvastamiseks, tuvastusreeglid esitatakse Uppaali ajaga automaatide kujul;
- Projekteerida ja realiseerida programmiliselt TDL^{TP} avaldiste interpretaator ja testimudeli generaator;
- Integreerida TDL^{TP} tõetatavalt korrektse testimise töövoogu;
- Demonstreerida TDL^{TP} otstarbekus ja teostatavus praktilisel mitte-triviaalsel testide spetsifitseerimise ja genereerimise näitel.

Lähtuvalt püstitatud eesmärkidest on väitekirjas loodud uudne tehnika mudelipõhise testimise automatiseerimiseks, mis sisaldab järgmisi tulemusi:

1. On loodud suure väljendusvõimsusega testieesmärkide spetsifitseerimise keel TDL^{TP}, mis võimaldab kompaktselt kirjeldada keerulisi testi stsenaariume ja ühendada stsenaariumides erinevaid testi kattekriteeriume, mis on vajalikud missioonikriitiliste süsteemide testimisel;
2. Defineeriti keele TDLTP operaatorite süntaks ja semantika ning teisendusreeglid TDL^{TP} deklaratiivsete avaldiste teisendamiseks täidetavaks testimudeliks, mis esitatakse Uppaal TA kujul;
3. Defineeritakse tõestatavalt korrektse testi arendusprotsessi mudel ja sellega seotud verifitseerimistingimused;
4. Väitekirja teoreetilised tulemused on valideeritud rakendusnäitel, milleks on TUT100 sateliidi tarkvara testimine.

Väitekirja tulemuste edasiarendamise võimalustena saab välja tuua järgmist:

Keele TDL^{TP} integreerimine reaktiivse planeeriva testri sünteesimeetodiga, mis on loodud mittedeterministlike kriitiliste süsteemide online testimiseks. Reaktiivse planeeriva kontrolleri idee pakuti välja esmakordselt *NASA Deep Space One* programmi raames Autonomy and Software Technology töörühma poolt, kui algne meetod kasutas ainult lõplike automaatide formaalset mudelit. TDL^{TP} sidumine meetodiga võimaldab sünteesimeetodit kasutusele võtta ka suurema väljendusvõimsusega Uppaal TA mudelite puhul.

Teine võimalik suund oleks siduda TDL^{TP} mitme-aspektiliste lepingute teooriaga selleks, et kasutades ühte ja sama spetsifitseerimiskeelt nii disaini, kui testi spetsifitseerimisel. TDL^{TP} väljendusvõimsus, deklaratiivsus ja kõrge abstraktsiooni tase annavad võimaluse genereerida teste otse lepingute spetsifikatsioonidest.

Kolmas suund oleks TDL^{TP} kasutamine lisaks konformsustestimisele ka mutatsioonitestimisel. See annab võimaluse spetsifitseerida mutatsioone mitte ainult testitava süsteemi mudeli terminites vaid ka abstraktsemalt testi eesmärgi spetsifikatsiooni enda mutatsioonidena.

Neljas praktilisem suund oleks keele TDL^{TP} testimudeli genereerimise tehnika skaleeruvuse täpsem uurimine ja mõõdistamine. Katsed TUT100 satelliidi tarkvara alamsüsteemide testimisel olid lubavad. Parema ettekujutuse skaleeruvusest annab satelliidi süsteemi terviktestimine, milleks avaneb võimalus satelliidi lõpliku valmimise järel.

Appendix 1

TestCase 1 test sequence

```
State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Idle EXISTS_TS23.Idle P_LEADSTO_Q.Idle
↪  StopWatch.Ready )
cl=0 Environment.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  StopWatch.Ready->StopWatch.Running { 1, Activate_LEADSTO!, Swatch := 0 }
  P_LEADSTO_Q.Idle->P_LEADSTO_Q._id31 { 1, Activate_LEADSTO?, 1 }

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Idle EXISTS_TS23.Idle P_LEADSTO_Q._id31
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  P_LEADSTO_Q._id31->P_LEADSTO_Q.Ready_for_P { 1, Activate_EXISTS!, 1 }
  FORALL_TS24.Idle->FORALL_TS24.Ready { 1, Activate_EXISTS?, 1 }

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Ready EXISTS_TS23.Idle P_LEADSTO_Q.Ready_for_P
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Delay: 15.5

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Ready EXISTS_TS23.Idle P_LEADSTO_Q.Ready_for_P
↪  StopWatch.Running )
cl=15.5 Environment.cl=15.5 StopWatch.Swatch=15.5 i_command_comm_create=0 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0

Transitions:
  Environment._id13->Environment._id13 { 1, i_command!, comm_valid := 1, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
  ↪  0, TS1[0] := 1 }

State:
( Environment._id13 MCS_EPS.commandCreated FORALL_TS24.Ready EXISTS_TS23.Idle
↪  P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment._id13 MCS_EPS.commandSent FORALL_TS24.Ready EXISTS_TS23.Idle
↪  P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0
```

```
Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1, f123 := 1, f124 :=
   ↪  1 }

State:
( Environment._id13 MCS_EPS.commandReceived FORALL_TS24.Ready EXISTS_TS23.Idle
↪  P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=1
↪  MCS_EPS.f124=1

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id1 { comm_valid, tau, TS3[0] := 1, TS23[0] := f123 ? 1 : 0,
   ↪  f123 := 0, f124 := 0 }

State:
( Environment._id13 MCS_EPS._id1 FORALL_TS24.Ready EXISTS_TS23.Idle P_LEADSTO_Q.Ready_for_P
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  MCS_EPS._id1->MCS_EPS.commandValid { 1, ch_TS23!, 1 }
  FORALL_TS24.Ready->FORALL_TS24._id15 { 1, ch_TS23?, 1 }

State:
( Environment._id13 MCS_EPS.commandValid FORALL_TS24._id15 EXISTS_TS23.Idle
↪  P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  MCS_EPS.commandValid->MCS_EPS.dataCreated { 1, tau, data_create := 1 }

State:
( Environment._id13 MCS_EPS.dataCreated FORALL_TS24._id15 EXISTS_TS23.Idle
↪  P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  MCS_EPS.dataCreated->MCS_EPS.replyReceived { 1, tau, data_send := 1 }

State:
( Environment._id13 MCS_EPS.replyReceived FORALL_TS24._id15 EXISTS_TS23.Idle
↪  P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id2 { 1, o_response!, n := o_response_p }
  Environment._id13->Environment._id13 { 1, o_response?, cl := 0 }

State:
( Environment._id13 MCS_EPS._id2 FORALL_TS24._id15 EXISTS_TS23.Idle P_LEADSTO_Q.Ready_for_P
↪  StopWatch.Running )
```

```
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  MCS_EPS._id2->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment._id13 MCS_EPS.start FORALL_TS24._id15 EXISTS_TS23.Idle P_LEADSTO_Q.Ready_for_P
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  FORALL_TS24._id15->FORALL_TS24.End { exists (i:(const (label index:(range (int) "0" "M -
   ↪  1")))) TS23[i], Done_EXISTS!, 1 }
  P_LEADSTO_Q.Ready_for_P->P_LEADSTO_Q._id29 { 1, Done_EXISTS?, 1 }

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.End EXISTS_TS23.Idle P_LEADSTO_Q._id29
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  FORALL_TS24.End->FORALL_TS24.Idle { 1, tau, Reset() }

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Idle EXISTS_TS23.Idle P_LEADSTO_Q._id29
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Transitions:
  P_LEADSTO_Q._id29->P_LEADSTO_Q.Ready_for_Q { 1, Activate_FORALL!, 1 }
  EXISTS_TS23.Idle->EXISTS_TS23.Ready { 1, Activate_FORALL?, 1 }

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Idle EXISTS_TS23.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=15.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0

Delay: 15.5

State:
( Environment._id13 MCS_EPS.start FORALL_TS24.Idle EXISTS_TS23.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=15.5 Environment.cl=15.5 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0

Transitions:
  Environment._id13->Environment._id13 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
   ↪  0, TS1[0] := 1 }

State:
```

```
( Environment._id13 MCS_EPS.commandCreated FORALL_TS24.Idle EXISTS_TS23.Ready
↪  P_LEADSTO_Q.Ready_for_Q StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment._id13 MCS_EPS.commandSent FORALL_TS24.Idle EXISTS_TS23.Ready
↪  P_LEADSTO_Q.Ready_for_Q StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1, fl23 := 1, fl24 :=
   ↪  1 }

State:
( Environment._id13 MCS_EPS.commandReceived FORALL_TS24.Idle EXISTS_TS23.Ready
↪  P_LEADSTO_Q.Ready_for_Q StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=1
↪  MCS_EPS.fl24=1

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id0 { !comm_valid, tau, TS4[0] := 1, TS24[0] := fl24 ? 1 :
   ↪  0, fl23 := 0, fl24 := 0 }

State:
( Environment._id13 MCS_EPS._id0 FORALL_TS24.Idle EXISTS_TS23.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=1 TS5[0]=0 TS23[0]=1 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0

Transitions:
  MCS_EPS._id0->MCS_EPS.commandInvalid { 1, ch_TS24!, 1 }
  EXISTS_TS23.Ready->EXISTS_TS23._id18 { 1, ch_TS24?, 1 }

State:
( Environment._id13 MCS_EPS.commandInvalid FORALL_TS24.Idle EXISTS_TS23._id18
↪  P_LEADSTO_Q.Ready_for_Q StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=1 TS5[0]=0 TS23[0]=1 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0

Transitions:
  EXISTS_TS23._id18->EXISTS_TS23.End { forall (i:(const (label index:(range (int) "0" "M -
   ↪  1")))) TS24[i], Done_FORALL!, 1 }
  P_LEADSTO_Q.Ready_for_Q->P_LEADSTO_Q._id27 { 1, Done_FORALL?, 1 }

State:
( Environment._id13 MCS_EPS.commandInvalid FORALL_TS24.Idle EXISTS_TS23.End P_LEADSTO_Q._id27
↪  StopWatch.Running )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=1 TS5[0]=0 TS23[0]=1 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0

Transitions:
  P_LEADSTO_Q._id27->P_LEADSTO_Q.End { 1, Done_LEADSTO!, 1 }
  StopWatch.Running->StopWatch.Pass { 1, Done_LEADSTO?, 1 }

State:
( Environment._id13 MCS_EPS.commandInvalid FORALL_TS24.Idle EXISTS_TS23.End P_LEADSTO_Q.End
↪  StopWatch.Pass )
cl=0 Environment.cl=0 StopWatch.Swatch=31 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=1 TS5[0]=0 TS23[0]=1 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=1 MCS_EPS.error_send=0 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0
```

Appendix 2

TestCase 2 test sequence

```
State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q.Idle REPEAT_P.Idle
↪  StopWatch.Ready )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  StopWatch.Ready->StopWatch.Running { 1, Activate_REPEAT!, Swatch := 0 }
  REPEAT_P.Idle->REPEAT_P._id34 { 1, Activate_REPEAT?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q.Idle REPEAT_P._id34
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  REPEAT_P._id34->REPEAT_P.Ready { 1, Activate_LEADSTO!, 1 }
  P_LEADSTO_Q.Idle->P_LEADSTO_Q._id31 { 1, Activate_LEADSTO?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q._id31
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  P_LEADSTO_Q._id31->P_LEADSTO_Q.Ready_for_P { 1, Activate_EXISTS!, 1 }
  EXIST_TS24.Idle->EXIST_TS24.Ready { 1, Activate_EXISTS?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0
↪  MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=0

Delay: 15.125

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=15.125 Environment_.cl=15.125 StopWatch.Swatch=15.125 i_command_comm_create=0 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  Environment_._id13->Environment_._id13 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
  ↪  0, TS1[0] := 1 }

State:
( Environment_._id13 MCS_EPS.commandCreated ALL_TS23.Idle EXIST_TS24.Ready
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0
```

```
Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id13 MCS_EPS.commandSent ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1, fl23 := 1, fl24 :=
   ↪  1 }

State:
( Environment_._id13 MCS_EPS.commandReceived ALL_TS23.Idle EXIST_TS24.Ready
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=1 MCS_EPS.fl24=1 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id0 { !comm_valid, tau, TS4[0] := 1, TS24[0] := fl24 ? 1 :
   ↪  0, fl23 := 0, fl24 := 0 }

State:
( Environment_._id13 MCS_EPS._id0 ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS._id0->MCS_EPS.commandInvalid { 1, ch_TS24!, 1 }
  EXIST_TS24.Ready->EXIST_TS24._id15 { 1, ch_TS24?, 1 }

State:
( Environment_._id13 MCS_EPS.commandInvalid ALL_TS23.Idle EXIST_TS24._id15
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandInvalid->MCS_EPS.errorCreated { 1, tau, error_create := 1 }

State:
( Environment_._id13 MCS_EPS.errorCreated ALL_TS23.Idle EXIST_TS24._id15 P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.errorCreated->MCS_EPS.replyReceived { 1, tau, error_send := 1 }

State:
( Environment_._id13 MCS_EPS.replyReceived ALL_TS23.Idle EXIST_TS24._id15
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id2 { 1, o_response!, n := o_response_p }
  Environment_._id13->Environment_._id13 { 1, o_response?, cl := 0 }

State:
( Environment_._id13 MCS_EPS._id2 ALL_TS23.Idle EXIST_TS24._id15 P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS._id2->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24._id15 P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  EXIST_TS24._id15->EXIST_TS24.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS24[i], Done_EXISTS!, 1 }
  P_LEADSTO_Q.Ready_for_P->P_LEADSTO_Q._id29 { 1, Done_EXISTS?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.End P_LEADSTO_Q._id29 REPEAT_P.Ready
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  EXIST_TS24.End->EXIST_TS24.Idle { 1, tau, Reset() }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q._id29
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  P_LEADSTO_Q._id29->P_LEADSTO_Q.Ready_for_Q { 1, Activate_FORALL!, 1 }
  ALL_TS23.Idle->ALL_TS23.Ready { 1, Activate_FORALL?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Delay: 15.125

State:
```

```
( Environment_._id13 MCS_EPS.start ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=15.125 Environment_.cl=15.125 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  Environment_._id13->Environment_._id13 { 1, i_command!, comm_valid := 1, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
   ↪  0, TS1[0] := 1 }

State:
( Environment_._id13 MCS_EPS.commandCreated ALL_TS23.Ready EXIST_TS24.Idle
↪  P_LEADSTO_Q.Ready_for_Q REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id13 MCS_EPS.commandSent ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1, fl23 := 1, fl24 :=
   ↪  1 }

State:
( Environment_._id13 MCS_EPS.commandReceived ALL_TS23.Ready EXIST_TS24.Idle
↪  P_LEADSTO_Q.Ready_for_Q REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=1 MCS_EPS.fl24=1 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id1 { comm_valid, tau, TS3[0] := 1, TS23[0] := fl23 ? 1 : 0,
   ↪  fl23 := 0, fl24 := 0 }

State:
( Environment_._id13 MCS_EPS._id1 ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS._id1->MCS_EPS.commandValid { 1, ch_TS23!, 1 }
  ALL_TS23.Ready->ALL_TS23._id18 { 1, ch_TS23?, 1 }

State:
( Environment_._id13 MCS_EPS.commandValid ALL_TS23._id18 EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.commandValid->MCS_EPS.dataCreated { 1, tau, data_create := 1 }

State:
( Environment_._id13 MCS_EPS.dataCreated ALL_TS23._id18 EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.dataCreated->MCS_EPS.replyReceived { 1, tau, data_send := 1 }

State:
( Environment_._id13 MCS_EPS.replyReceived ALL_TS23._id18 EXIST_TS24.Idle
↪  P_LEADSTO_Q.Ready_for_Q REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id2 { 1, o_response!, n := o_response_p }
  Environment_._id13->Environment_._id13 { 1, o_response?, cl := 0 }

State:
( Environment_._id13 MCS_EPS._id2 ALL_TS23._id18 EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  MCS_EPS._id2->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23._id18 EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  ALL_TS23._id18->ALL_TS23.End { forall (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS23[i], Done_FORALL!, 1 }
  P_LEADSTO_Q.Ready_for_Q->P_LEADSTO_Q._id27 { 1, Done_FORALL?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.End EXIST_TS24.Idle P_LEADSTO_Q._id27 REPEAT_P.Ready
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0

Transitions:
  ALL_TS23.End->ALL_TS23.Idle { 1, tau, Reset() }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q._id27
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_P.i=0
```

```
Transitions:
  P_LEADSTO_Q._id27->P_LEADSTO_Q.End { 1, Done_LEADSTO!, 1 }
  REPEAT_P.Ready->REPEAT_P._id36 { 1, Done_LEADSTO?, i++ }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q.End REPEAT_P._id36
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  P_LEADSTO_Q.End->P_LEADSTO_Q.Idle { 1, tau, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q.Idle REPEAT_P._id36
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  REPEAT_P._id36->REPEAT_P._id34 { i < 2, tau, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q.Idle REPEAT_P._id34
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  REPEAT_P._id34->REPEAT_P.Ready { 1, Activate_LEADSTO!, 1 }
  P_LEADSTO_Q.Idle->P_LEADSTO_Q._id31 { 1, Activate_LEADSTO?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q._id31
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  P_LEADSTO_Q._id31->P_LEADSTO_Q.Ready_for_P { 1, Activate_EXISTS!, 1 }
  EXIST_TS24.Idle->EXIST_TS24.Ready { 1, Activate_EXISTS?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1

Delay: 15.25

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=15.25 Environment_.cl=15.25 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1
```

```
Transitions:
  Environment_._id13->Environment_._id13 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
  ↪  0, TS1[0] := 1 }

State:
( Environment_._id13 MCS_EPS.commandCreated ALL_TS23.Idle EXIST_TS24.Ready
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id13 MCS_EPS.commandSent ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1, fl23 := 1, fl24 :=
  ↪  1 }

State:
( Environment_._id13 MCS_EPS.commandReceived ALL_TS23.Idle EXIST_TS24.Ready
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=1
↪  MCS_EPS.fl24=1 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id0 { !comm_valid, tau, TS4[0] := 1, TS24[0] := fl24 ? 1 :
  ↪  0, fl23 := 0, fl24 := 0 }

State:
( Environment_._id13 MCS_EPS._id0 ALL_TS23.Idle EXIST_TS24.Ready P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS._id0->MCS_EPS.commandInvalid { 1, ch_TS24!, 1 }
  EXIST_TS24.Ready->EXIST_TS24._id15 { 1, ch_TS24?, 1 }

State:
( Environment_._id13 MCS_EPS.commandInvalid ALL_TS23.Idle EXIST_TS24._id15
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandInvalid->MCS_EPS.errorCreated { 1, tau, error_create := 1 }

State:
( Environment_._id13 MCS_EPS.errorCreated ALL_TS23.Idle EXIST_TS24._id15 P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.errorCreated->MCS_EPS.replyReceived { 1, tau, error_send := 1 }

State:
( Environment_._id13 MCS_EPS.replyReceived ALL_TS23.Idle EXIST_TS24._id15
↪  P_LEADSTO_Q.Ready_for_P REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id2 { 1, o_response!, n := o_response_p }
  Environment_._id13->Environment_._id13 { 1, o_response?, cl := 0 }

State:
( Environment_._id13 MCS_EPS._id2 ALL_TS23.Idle EXIST_TS24._id15 P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  MCS_EPS._id2->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24._id15 P_LEADSTO_Q.Ready_for_P
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  EXIST_TS24._id15->EXIST_TS24.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS24[i], Done_EXISTS!, 1 }
  P_LEADSTO_Q.Ready_for_P->P_LEADSTO_Q._id29 { 1, Done_EXISTS?, 1 }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.End P_LEADSTO_Q._id29 REPEAT_P.Ready
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=1 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  EXIST_TS24.End->EXIST_TS24.Idle { 1, tau, Reset() }

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Idle EXIST_TS24.Idle P_LEADSTO_Q._id29
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  P_LEADSTO_Q._id29->P_LEADSTO_Q.Ready_for_Q { 1, Activate_FORALL!, 1 }
  ALL_TS23.Idle->ALL_TS23.Ready { 1, Activate_FORALL?, 1 }

State:
```

```
( Environment_._id13 MCS_EPS.start ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Delay: 15.5

State:
( Environment_._id13 MCS_EPS.start ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=15.5 Environment_.cl=15.5 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=1 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  Environment_._id13->Environment_._id13 { 1, i_command!, comm_valid := 1, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
  ↪  0, TS1[0] := 1 }

State:
( Environment_._id13 MCS_EPS.commandCreated ALL_TS23.Ready EXIST_TS24.Idle
↪  P_LEADSTO_Q.Ready_for_Q REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id13 MCS_EPS.commandSent ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1, fl23 := 1, fl24 :=
  ↪  1 }

State:
( Environment_._id13 MCS_EPS.commandReceived ALL_TS23.Ready EXIST_TS24.Idle
↪  P_LEADSTO_Q.Ready_for_Q REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=1
↪  MCS_EPS.fl24=1 REPEAT_P.i=1

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id1 { comm_valid, tau, TS3[0] := 1, TS23[0] := fl23 ? 1 : 0,
  ↪  fl23 := 0, fl24 := 0 }

State:
( Environment_._id13 MCS_EPS._id1 ALL_TS23.Ready EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.fl23=0
↪  MCS_EPS.fl24=0 REPEAT_P.i=1

Transitions:
  MCS_EPS._id1->MCS_EPS.commandValid { 1, ch_TS23!, 1 }
  ALL_TS23.Ready->ALL_TS23._id18 { 1, ch_TS23?, 1 }
```

```
State:
( Environment_._id13 MCS_EPS.commandValid ALL_TS23._id18 EXIST_TS24.Idle P_LEADSTO_Q.Ready_for_Q
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  ALL_TS23._id18->ALL_TS23.End { forall (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS23[i], Done_FORALL!, 1 }
  P_LEADSTO_Q.Ready_for_Q->P_LEADSTO_Q._id27 { 1, Done_FORALL?, 1 }

State:
( Environment_._id13 MCS_EPS.commandValid ALL_TS23.End EXIST_TS24.Idle P_LEADSTO_Q._id27
↪  REPEAT_P.Ready StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=1

Transitions:
  P_LEADSTO_Q._id27->P_LEADSTO_Q.End { 1, Done_LEADSTO!, 1 }
  REPEAT_P.Ready->REPEAT_P._id36 { 1, Done_LEADSTO?, i++ }

State:
( Environment_._id13 MCS_EPS.commandValid ALL_TS23.End EXIST_TS24.Idle P_LEADSTO_Q.End
↪  REPEAT_P._id36 StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=2

Transitions:
  REPEAT_P._id36->REPEAT_P.End { i >= 2, Done_REPEAT!, 1 }
  StopWatch.Running->StopWatch.Pass { 1, Done_REPEAT?, 1 }

State:
( Environment_._id13 MCS_EPS.commandValid ALL_TS23.End EXIST_TS24.Idle P_LEADSTO_Q.End
↪  REPEAT_P.End StopWatch.Pass )
cl=0 Environment_.cl=0 StopWatch.Swatch=61 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=1
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=1 TS24[0]=0 comm_valid=1 MCS_EPS.comm_create=0
↪  MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=1
↪  MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1 MCS_EPS.f123=0
↪  MCS_EPS.f124=0 REPEAT_P.i=2
```

Appendix 3

TestCase 3 test sequence

```
State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Idle StopWatch.Ready )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  StopWatch.Ready->StopWatch.Running { 1, Activate_LEADSTO!, Swatch := 0 }
  P_LEADSTO_Q.Idle->P_LEADSTO_Q._id36 { 1, Activate_LEADSTO?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q._id36 StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  P_LEADSTO_Q._id36->P_LEADSTO_Q.Ready_for_P { 1, Activate_EXISTS_TS5!, 1 }
  EXISTS_TS5.Idle->EXISTS_TS5.Ready { 1, Activate_EXISTS_TS5?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Ready FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=0 i_command_comm_create=0 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Delay: 15.015625

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Ready FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=15.015625 Environment_.cl=15.015625 StopWatch.Swatch=15.015625 i_command_comm_create=0
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=0
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0

Transitions:
  Environment_._id14->Environment_._id14 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
   ↪  0 }

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Ready FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=15.015625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=0 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Delay: 0.015625

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Ready FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0.015625 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=0
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0
```

```
Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 1 }

State:
( Environment_._id14 MCS_EPS.commandSent EXISTS_TS5.Ready FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0.015625 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS._id0 { eps_busy, tau, TS5[0] := 1 }

State:
( Environment_._id14 MCS_EPS._id0 EXISTS_TS5.Ready FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0.015625 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=1 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id0->MCS_EPS.commandIgnored { 1, ch_TS5!, 1 }
  EXISTS_TS5.Ready->EXISTS_TS5._id19 { 1, ch_TS5?, 1 }

State:
( Environment_._id14 MCS_EPS.commandIgnored EXISTS_TS5._id19 FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0.015625 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=1 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandIgnored->MCS_EPS._id3 { 1, tau, 1 }

State:
( Environment_._id14 MCS_EPS._id3 EXISTS_TS5._id19 FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0.015625 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=1 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id3->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5._id19 FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q.Ready_for_P StopWatch.Running )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=1 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  EXISTS_TS5._id19->EXISTS_TS5.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS5[i], Done_EXISTS_TS5!, 1 }
  P_LEADSTO_Q.Ready_for_P->P_LEADSTO_Q._id34 { 1, Done_EXISTS_TS5?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.End FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q._id34 StopWatch.Running )
```

```
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=1 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  EXISTS_TS5.End->EXISTS_TS5.Idle { 1, tau, Reset() }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q.Idle P_LEADSTO_Q._id34 StopWatch.Running )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  P_LEADSTO_Q._id34->P_LEADSTO_Q.Ready_for_Q { 1, Activate_AND!, 1 }
  P_AND_Q.Idle->P_AND_Q._id54 { 1, Activate_AND?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3.Idle REPEAT_RETS4.Idle P_AND_Q._id54 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  P_AND_Q._id54->P_AND_Q._id48 { 1, Activate_RATS3!, 1 }
  REPEAT_RATS3.Idle->REPEAT_RATS3._id39 { 1, Activate_RATS3?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3._id39 REPEAT_RETS4.Idle P_AND_Q._id48 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  REPEAT_RATS3._id39->REPEAT_RATS3.Ready { 1, Activate_FORALL!, 1 }
  FORALL_TS3.Idle->FORALL_TS3.Ready { 1, Activate_FORALL?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id48 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  P_AND_Q._id48->P_AND_Q.Ready { 1, Activate_RETS4!, 1 }
  REPEAT_RETS4.Idle->REPEAT_RETS4._id44 { 1, Activate_RETS4?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id44 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0
```

```
Transitions:
  REPEAT_RETS4._id44->REPEAT_RETS4.Ready { 1, Activate_EXISTS_TS4!, 1 }
  EXISTS_TS4.Idle->EXISTS_TS4.Ready { 1, Activate_EXISTS_TS4?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0.015625 StopWatch.Swatch=15.03125 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Delay: 15.03125

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=15.03125 Environment_.cl=15.046875 StopWatch.Swatch=30.0625 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1 MCS_EPS.data_create=0 MCS_EPS.error_create=0
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=0 MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=0

Transitions:
  Environment_._id14->Environment_._id14 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
   ↪  0 }

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=1
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id14 MCS_EPS.commandSent EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=0 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1 }

State:
( Environment_._id14 MCS_EPS.commandReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.f123=0 MCS_EPS.f124=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id1 { !comm_valid, tau, TS4[0] := 1 }

State:
```

```
( Environment_._id14 MCS_EPS._id1 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id1->MCS_EPS.commandInvalid { 1, ch_TS4!, 1 }
  EXISTS_TS4.Ready->EXISTS_TS4._id16 { 1, ch_TS4?, 1 }

State:
( Environment_._id14 MCS_EPS.commandInvalid EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=0 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandInvalid->MCS_EPS.errorCreated { 1, tau, error_create := 1 }

State:
( Environment_._id14 MCS_EPS.errorCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=0
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.errorCreated->MCS_EPS.replyReceived { 1, tau, error_send := 1 }

State:
( Environment_._id14 MCS_EPS.replyReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id3 { 1, o_response!, n := o_response_p }
  Environment_._id14->Environment_._id14 { 1, o_response?, cl := 0 }

State:
( Environment_._id14 MCS_EPS._id3 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id3->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  EXISTS_TS4._id16->EXISTS_TS4.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
  ↪  TS4[i], Done_EXISTS_TS4!, 1 }
  REPEAT_RETS4.Ready->REPEAT_RETS4._id46 { 1, Done_EXISTS_TS4?, i++ }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.End
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id46 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  EXISTS_TS4.End->EXISTS_TS4.Idle { 1, tau, Reset() }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id46 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  REPEAT_RETS4._id46->REPEAT_RETS4._id44 { i < 3, tau, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id44 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  REPEAT_RETS4._id44->REPEAT_RETS4.Ready { 1, Activate_EXISTS_TS4!, 1 }
  EXISTS_TS4.Idle->EXISTS_TS4.Ready { 1, Activate_EXISTS_TS4?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=30.0625 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Delay: 15.0625

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=15.0625 Environment_.cl=15.0625 StopWatch.Swatch=45.125 i_command_comm_create=1
↪  o_response_p=0 n=0 TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0
↪  comm_valid=0 ch_TS24=0 ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1
↪  MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0 MCS_EPS.data_create=0 MCS_EPS.error_create=1
↪  MCS_EPS.data_send=0 MCS_EPS.error_send=1 MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0
↪  REPEAT_RETS4.i=1
```

```
Transitions:
  Environment_._id14->Environment_._id14 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
  ↪  0 }

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id14 MCS_EPS.commandSent EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1 }

State:
( Environment_._id14 MCS_EPS.commandReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id1 { !comm_valid, tau, TS4[0] := 1 }

State:
( Environment_._id14 MCS_EPS._id1 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS._id1->MCS_EPS.commandInvalid { 1, ch_TS4!, 1 }
  EXISTS_TS4.Ready->EXISTS_TS4._id16 { 1, ch_TS4?, 1 }

State:
( Environment_._id14 MCS_EPS.commandInvalid EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS.commandInvalid->MCS_EPS.errorCreated { 1, tau, error_create := 1 }

State:
```

```
( Environment_._id14 MCS_EPS.errorCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS.errorCreated->MCS_EPS.replyReceived { 1, tau, error_send := 1 }

State:
( Environment_._id14 MCS_EPS.replyReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id3 { 1, o_response!, n := o_response_p }
  Environment_._id14->Environment_._id14 { 1, o_response?, cl := 0 }

State:
( Environment_._id14 MCS_EPS._id3 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  MCS_EPS._id3->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=1

Transitions:
  EXISTS_TS4._id16->EXISTS_TS4.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS4[i], Done_EXISTS_TS4!, 1 }
  REPEAT_RETS4.Ready->REPEAT_RETS4._id46 { 1, Done_EXISTS_TS4?, i++ }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.End
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id46 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  EXISTS_TS4.End->EXISTS_TS4.Idle { 1, tau, Reset() }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id46 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  REPEAT_RETS4._id46->REPEAT_RETS4._id44 { i < 3, tau, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id44 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  REPEAT_RETS4._id44->REPEAT_RETS4.Ready { 1, Activate_EXISTS_TS4!, 1 }
  EXISTS_TS4.Idle->EXISTS_TS4.Ready { 1, Activate_EXISTS_TS4?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=45.125 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Delay: 15.125

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=15.125 Environment_.cl=15.125 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0
↪  n=0 TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  Environment_._id14->Environment_._id14 { 1, i_command!, comm_valid := 0, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
  ↪  0 }

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id14 MCS_EPS.commandSent EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1 }

State:
( Environment_._id14 MCS_EPS.commandReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id1 { !comm_valid, tau, TS4[0] := 1 }

State:
( Environment_._id14 MCS_EPS._id1 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Ready
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS._id1->MCS_EPS.commandInvalid { 1, ch_TS4!, 1 }
  EXISTS_TS4.Ready->EXISTS_TS4._id16 { 1, ch_TS4?, 1 }

State:
( Environment_._id14 MCS_EPS.commandInvalid EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS.commandInvalid->MCS_EPS.errorCreated { 1, tau, error_create := 1 }

State:
( Environment_._id14 MCS_EPS.errorCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS.errorCreated->MCS_EPS.replyReceived { 1, tau, error_send := 1 }

State:
( Environment_._id14 MCS_EPS.replyReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id3 { 1, o_response!, n := o_response_p }
  Environment_._id14->Environment_._id14 { 1, o_response?, cl := 0 }

State:
( Environment_._id14 MCS_EPS._id3 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  MCS_EPS._id3->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4._id16
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Ready P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=2

Transitions:
  EXISTS_TS4._id16->EXISTS_TS4.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS4[i], Done_EXISTS_TS4!, 1 }
  REPEAT_RETS4.Ready->REPEAT_RETS4._id46 { 1, Done_EXISTS_TS4?, i++ }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.End
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id46 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=1 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=3

Transitions:
  EXISTS_TS4.End->EXISTS_TS4.Idle { 1, tau, Reset() }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4._id46 P_AND_Q.Ready P_LEADSTO_Q.Ready_for_Q
↪  StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=3

Transitions:
  REPEAT_RETS4._id46->REPEAT_RETS4.End { i >= 3, Done_RETS4!, 1 }
  P_AND_Q.Ready->P_AND_Q._id50 { 1, Done_RETS4?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.End P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=3

Transitions:
  REPEAT_RETS4.End->REPEAT_RETS4.Idle { 1, tau, i := 0 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=60.25 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0
```

```
Delay: 15.25

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=15.25 Environment_.cl=15.25 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=0 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  Environment_._id14->Environment_._id14 { 1, i_command!, comm_valid := 1, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
   ↪  0 }

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id14 MCS_EPS.commandSent EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1 }

State:
( Environment_._id14 MCS_EPS.commandReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id2 { comm_valid, tau, TS3[0] := 1 }

State:
( Environment_._id14 MCS_EPS._id2 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id2->MCS_EPS.commandValid { 1, ch_TS3!, 1 }
  FORALL_TS3.Ready->FORALL_TS3._id23 { 1, ch_TS3?, 1 }

State:
( Environment_._id14 MCS_EPS.commandValid EXISTS_TS5.Idle FORALL_TS3._id23 EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=0 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandValid->MCS_EPS.dataCreated { 1, tau, data_create := 1 }

State:
( Environment_._id14 MCS_EPS.dataCreated EXISTS_TS5.Idle FORALL_TS3._id23 EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=0 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.dataCreated->MCS_EPS.replyReceived { 1, tau, data_send := 1 }

State:
( Environment_._id14 MCS_EPS.replyReceived EXISTS_TS5.Idle FORALL_TS3._id23 EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.replyReceived->MCS_EPS._id3 { 1, o_response!, n := o_response_p }
  Environment_._id14->Environment_._id14 { 1, o_response?, cl := 0 }

State:
( Environment_._id14 MCS_EPS._id3 EXISTS_TS5.Idle FORALL_TS3._id23 EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id3->MCS_EPS.start { 1, tau, cl := 0 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3._id23 EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=0 REPEAT_RETS4.i=0

Transitions:
  FORALL_TS3._id23->FORALL_TS3.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS3[i], Done_FORALL!, 1 }
  REPEAT_RATS3.Ready->REPEAT_RATS3._id41 { 1, Done_FORALL?, i++ }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.End EXISTS_TS4.Idle
↪  REPEAT_RATS3._id41 REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0
```

```
Transitions:
  FORALL_TS3.End->FORALL_TS3.Idle { 1, tau, Reset() }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3._id41 REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  REPEAT_RATS3._id41->REPEAT_RATS3._id39 { i < 2, tau, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Idle EXISTS_TS4.Idle
↪  REPEAT_RATS3._id39 REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  REPEAT_RATS3._id39->REPEAT_RATS3.Ready { 1, Activate_FORALL!, 1 }
  FORALL_TS3.Idle->FORALL_TS3.Ready { 1, Activate_FORALL?, 1 }

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=75.5 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Delay: 15.5

State:
( Environment_._id14 MCS_EPS.start EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=15.5 Environment_.cl=15.5 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0
↪  TS1[0]=0 TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0
↪  ch_TS23=0 MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  Environment_._id14->Environment_._id14 { 1, i_command!, comm_valid := 1, cl := 0 }
  MCS_EPS.start->MCS_EPS.commandCreated { cl > 15, i_command?, i_command_comm_create := 1, cl :=
   ↪  0 }

State:
( Environment_._id14 MCS_EPS.commandCreated EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandCreated->MCS_EPS.commandSent { 1, tau, comm_send := 1, eps_busy := 0 }

State:
( Environment_._id14 MCS_EPS.commandSent EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
```

```
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandSent->MCS_EPS.commandReceived { !eps_busy, tau, TS2[0] := 1 }

State:
( Environment_._id14 MCS_EPS.commandReceived EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=0 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS.commandReceived->MCS_EPS._id2 { comm_valid, tau, TS3[0] := 1 }

State:
( Environment_._id14 MCS_EPS._id2 EXISTS_TS5.Idle FORALL_TS3.Ready EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  MCS_EPS._id2->MCS_EPS.commandValid { 1, ch_TS3!, 1 }
  FORALL_TS3.Ready->FORALL_TS3._id23 { 1, ch_TS3?, 1 }

State:
( Environment_._id14 MCS_EPS.commandValid EXISTS_TS5.Idle FORALL_TS3._id23 EXISTS_TS4.Idle
↪  REPEAT_RATS3.Ready REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=1 REPEAT_RETS4.i=0

Transitions:
  FORALL_TS3._id23->FORALL_TS3.End { exists (i:(const (label index:(range (int) "0" "M - 1"))))
   ↪  TS3[i], Done_FORALL!, 1 }
  REPEAT_RATS3.Ready->REPEAT_RATS3._id41 { 1, Done_FORALL?, i++ }

State:
( Environment_._id14 MCS_EPS.commandValid EXISTS_TS5.Idle FORALL_TS3.End EXISTS_TS4.Idle
↪  REPEAT_RATS3._id41 REPEAT_RETS4.Idle P_AND_Q._id50 P_LEADSTO_Q.Ready_for_Q StopWatch.Running
↪  )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=2 REPEAT_RETS4.i=0

Transitions:
  REPEAT_RATS3._id41->REPEAT_RATS3.End { i == 2, Done_RATS3!, 1 }
  P_AND_Q._id50->P_AND_Q._id51 { 1, Done_RATS3?, 1 }

State:
( Environment_._id14 MCS_EPS.commandValid EXISTS_TS5.Idle FORALL_TS3.End EXISTS_TS4.Idle
↪  REPEAT_RATS3.End REPEAT_RETS4.Idle P_AND_Q._id51 P_LEADSTO_Q.Ready_for_Q StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=2 REPEAT_RETS4.i=0
```

```
Transitions:
  P_AND_Q._id51->P_AND_Q.End { 1, Done_AND!, 1 }
  P_LEADSTO_Q.Ready_for_Q->P_LEADSTO_Q._id32 { 1, Done_AND?, 1 }

State:
( Environment_._id14 MCS_EPS.commandValid EXISTS_TS5.Idle FORALL_TS3.End EXISTS_TS4.Idle
↪  REPEAT_RATS3.End REPEAT_RETS4.Idle P_AND_Q.End P_LEADSTO_Q._id32 StopWatch.Running )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=2 REPEAT_RETS4.i=0

Transitions:
  P_LEADSTO_Q._id32->P_LEADSTO_Q.End { 1, Done_LEADSTO!, 1 }
  StopWatch.Running->StopWatch.Pass { 1, Done_LEADSTO?, 1 }

State:
( Environment_._id14 MCS_EPS.commandValid EXISTS_TS5.Idle FORALL_TS3.End EXISTS_TS4.Idle
↪  REPEAT_RATS3.End REPEAT_RETS4.Idle P_AND_Q.End P_LEADSTO_Q.End StopWatch.Pass )
cl=0 Environment_.cl=0 StopWatch.Swatch=91 i_command_comm_create=1 o_response_p=0 n=0 TS1[0]=0
↪  TS2[0]=1 TS3[0]=1 TS4[0]=0 TS5[0]=0 TS23[0]=0 TS24[0]=0 comm_valid=1 ch_TS24=0 ch_TS23=0
↪  MCS_EPS.comm_create=0 MCS_EPS.comm_send=1 MCS_EPS.comm_rec=0 MCS_EPS.eps_busy=0
↪  MCS_EPS.data_create=1 MCS_EPS.error_create=1 MCS_EPS.data_send=1 MCS_EPS.error_send=1
↪  MCS_EPS.fl23=0 MCS_EPS.fl24=0 REPEAT_RATS3.i=2 REPEAT_RETS4.i=0
```

Appendix 4

Publication I

E. Halling, J. Vain, A. Boyarchuk, and O. Illiashenko. Test scenario specification language for model-based testing. *International Journal of Computing*, 2019

computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

TEST SCENARIO SPECIFICATION LANGUAGE FOR MODEL-BASED TESTING

Evelin Halling [1], Jüri Vain [1], Artem Boyarchuk [2], Oleg Illiashenko [2]

[1] Akadeemia tee 15A, Tallinn, , Estonia; evelin.halling@taltech.ee, http://www.taltech.ee
[2] Postal address, e-mail, Web address (URL)

Paper history:
Received …
Received in revised form …
Accepted …
Available online …

Keywords:
Model-based testing;
Test scenario description language;
Timed automata;
Verification by model checking;
Conformance testing.

Abstract: The paper defines a high-level test scenario specification language TDL^{TP} for specifying complex test scenarios that are needed for model-based testing of mission critical systems. The syntax and semantics of TDL^{TP} operators are defined and the transformation rules that map declarative TDL^{TP} expressions to executable test models represented as Uppaal Timed Automata are specified. . The scalability of the test purpose specification and test generation method has been demonstrated when applying it on TUT100 satellite software integration testing.

1. INTRODUCTION

In model-based testing (MBT), the requirements model of System Under Test (SUT) describes the expected correct behavior of the system under possible inputs from its environment. The model, represented in a suitable machine interpretable formalism, can be used to automatically generate the test cases either offline or online, and be used as the oracle that checks if the SUT behavior conforms to this model. Offline test generation means that tests are generated before test execution and executed when needed. In online test generation the model is executed in lock step with the SUT. The test model communicates with SUT via controllable inputs and observable outputs of the SUT.

Test description in MBT typically relies on two formal representations, SUT modelling language and the test purpose specification language.

The requirements to the test purpose specification languages for MBT can be summarized as following:

1. intuitive and user-friendly specification process;
2. expressiveness to capture the features and behaviours under test in a compact and unambiguous form;
3. formal semantics to make the test purpose specifications verifiable and pertinent for automated test generation;
4. decidability to make the test generation from test purpose specification algorithmically feasible.

The first two criteria have been capitalized in earlier attempts of designing test purpose specification languages. Check Case Definition Language (CCDL) [1] provides a high-level approach for requirements-based black-box system level testing. Test simulations and expected results specified in human readable form in CCDL can be compiled into executable test scripts. However, due to the lack of standardization, high-level test in CCDL are heavily tool-dependent and can be used only in tool specific testing processes.

High-level keyword-based test languages, such as the Robot Framework [2], have also been integrated with MBT [3]. In domains such as avionics [4] and automotive industry the efforts have been made to address the standardization of testing methods and languages, e.g. creating a meta-model for testing avionics systems [4], and the Automotive TestML [5]. Similarly, the Open Test Sequence Exchange Format (OTX) [6] standardized by ISO provides tool-independent XML-based data exchange format [7] for description and documentation of executable test sequences. These efforts have focused primarily on enabling the exchange of test specifications between involved stakeholders and tools. Due to their domain and purpose specialization the applicability of these languages in other domains is limited.

The Message Sequence Chart (MSC) [8] standardized by International Telecommunication Union was one of the first scenario specification languages though it was not only focusing on testing. The semantics of MSC is specified in [9]. Some of the features of MSC are adopted in UML, e.g. in Sequence Diagrams. Still, loose semantics limits its use as a consistent test description language [10].

Precise UML [11] introduces a subset of UML and OCL for MBT. The attempt to unify the semantics of different diagrams was motivated by the need for behavioral specifications of SUT which are well suited for generating test cases out of SUT models.

Concrete test scripting language, such as TTCN-3, regardless their strict semantics are not well suited for high-level description of test scenarios. They rather follow the style of syntax typical to imperative programming languages [12].

Thus, most of the test purpose specification languages referred above suffer from some of the disadvantages, either they have imprecise or informal semantics, lack of standardization, lack of comprehensive tool support, or poor interoperability with other development and testing tools.

European Telecommunications Standards Institute (ETSI) intended to address these shortcomings and developed a new specification language standard by introducing Test Purpose Language (TPLan) that supports the high-level expression of test purposes in prose [13]. Though TPLan provides notation for the standardized specification of test purposes, it leaves a gap between the declarative test purpose and its imperative implementation in test. Without formal semantics the development of test descriptions by means of different notations and dialects led to overhead and inconsistencies that need to be checked and fixed manually. As a consequence, ETSI started a new initiative by developing the Test Description Language TDL [12]. It is intended to bridge the gap between declarative test purposes and imperative test cases by offering a standardized approach for the specification of test descriptions. The main benefits of ETSI TDL outlined in [12] are higher quality tests through better design, easier layout to review by non-testing experts, better and faster test development, and seamless integration of methodology and tools.

The development of ETSI TDL was driven by industry where it is used primarily, but not exclusively, for functional testing. To enable the application of TDL in UML based working environments, a UML Profile for TDL (UP4TDL) [10] was developed. Domain-specific concepts are represented in UP4TDL by means of stereotypes.

Though TDL features one of the most advanced test purpose description language it has room for improvements. In the first place, automatic mapping of ETSI TDL to TTCN-3 is not fully implemented yet. The mapping is needed for generating executable tests from TDL descriptions and re-using the existing TTCN-3 tools and frameworks for test execution.

Second limitation of TDL is restricted timing semantics. The Time package in TDL contains concepts for the specification of time operations, time constraints, and timers. Since time in TDL is global and progresses monotonically in discrete quantities there is no way of expressing synchronization conditions between local time events of parallel processes and detecting possible Zeno computations that can be analyzed in continuous time models. Similarly time-divergency and timelock-freedom cannot be analyzed.

One step further towards automatic test generation was timed games based synthesis of test strategies introduced in [14] and implemented in the Uppaal Tiga tool. Timed Computation Tree Logic (TCTL) used for specifying test purpose in this approach has high expressive power and formal semantics relevant for expressing quantitative time properties combined with CTL operators such as *‘always’*, *‘inevitable’*, *‘potentially always’*, ‘*possible*’, and *‘leads-to’* [15].

Due to the complexity of model checking, the TCTL syntax in Uppaal tool is limited with un-nested operators making the TCTL expressions flat with respect to the temporal operators. On the other hand, to specify the properties of timed reachability the flat TCTL expressions are not sufficient for specifying complex properties and so called auxiliary property recognizing automata, e.g. ‘stopwatch’ automata are needed. Modifying the test model structure by adding property automata is not trivial for non-experts and may be an error prone process leading to the unintended changes of semantics of tests.

The aim of this work is to build an extra language layer (Test Scenario Definition Language - TDL^{TP}) for test scenario specification that is expressive, free from the limitations of ‘flat’ TCTL, interpretable in Uppaal TA, and suited for test generation.

In our approach, Uppaal Timed Automata (TA) [16] serve as a SUT specification language. Uppaal TA have been chosen because they are designed to express the timed behavior of state transition systems and there exists a mature set of tools that supports model construction, verification and online model-based testing [17].

For the test purpose specification to be concise and still expressive its specification language must be more abstract than SUT modeling language and not necessarily self-contained in the sense that its expressions are interpreted in the context of SUT model only. It means that the terms of test purpose specification refer to the SUT model structural elements of interest, they are called test coverage items (TCIs). The test purpose specification language TDL^{TP} proposed in our approach allows expressing multiple coverage criteria in terms of TCIs, including test scenario constraints such as *iteration*, *next*, *leads to*, and structural coverage criteria such as *selected states*, *selected transitions*, *transition pairs*, and timing constraints, e.g. *time bounded leads to*.

Generating the test model based on the SUT model and TDL^{TP} coverage expression includes two phases. In the first phase, the TCIs have to be labelled in the SUT model with Boolean variables called *traps*. The traps are needed to make TCIs referable in the TDL^{TP} expressions. In case of non-deterministic SUT model the coverage of those elementary TCIs is ensured by reactive planning tester (RPT) automata, one automaton for each conjunctive set of TCIs (see [19] for further details of RPT generation). In the second phase of generation, a test supervisor model M^{SVR} is constructed from the TDL^{TP} expression to trigger the RPT automata according to the test scenario so that the temporal and logical coverage constraints stated in TDL^{TP} specification would be satisfied. Since non-deterministic SUT models based tests are partially controllable only pseudo optimal traces can be generated by this method. Alternatively, in case of deterministic SUT models, the RPT automata generation phase can be discarded since Uppaal model checker generates optimal witness traces from the parallel composition of SUT and tester models. The rest of this paper is organized as follows. In Section 2 Uppaal Timed Automata formalism is introduced, Sections 3 and 4 define the TDL^{TP} language syntax and semantics respectively, Section 5 defines the map from TDL^{TP} to Uppaal TA that controls if the test scenario execution satisfies its declarative expression. In Section 6 the reduction rules of TDL^{TP} expressions are presented. Section 7 describes how the whole test model is composed by introducing test supervisor automaton. Section 8 explains how the test verdict and test diagnosis capability are encoded in the tester model, and finally the conclusions are drawn.

2. UPPAAL TIMED AUTOMATA

Uppaal Timed Automata [16] (TA) used for modelling SUT is defined as a closed network of extended timed automata that are called *processes*. The processes are gathered into a single system by parallel composition known from the process algebra CCS. An example of a system comprising two automata is given in Fig. 1.

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by double circle inside the location.

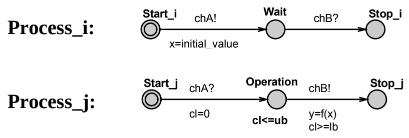


Figure 1 - A sample model: synchronous composition of two Uppaal automata Process_i and Process_j

Synchronous communication between the processes is by hand-shake synchronization links that are called *channels*. A channel relates a pair of edges labeled with symbols for input actions denoted by e.g. chA? and chB? in Fig. 1, and output actions denoted by chA! and chB!, where chA and chB are the names of the channels.

In Fig. 1, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronized transitions. Process_i, initially at location Start_i, initiates the call by executing the send action chA! that is synchronized with the receive action chA? in Process_j. The location Operation denotes the situation where Process_j computes the value to output variable y. Once done, the control is returned to Process_i by the action chB!.

The duration of executing the result is specified by the interval [*lb*, *ub*] where the upper bound *ub* is given by the *invariant* cl<=ub of location Operation, and the lower bound *lb* by the guard condition cl>=lb of the transition Operation ⟶ Stop_j. The *assignment* cl=0 on the transition Start_j ⟶ Operation ensures that the clock cl is reset when the control reaches the location Operation. The global variables x and y model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function f(x) defined in the declarations block.

While the synchronous communication between processes is modeled using channels, asynchronous communication between processes is modeled using global variables accessible to all processes.

Formally, the Uppaal TA are defined as follows:

Let $\sum$ denote a finite alphabet of actions *a, b, …* and *C* a finite set of real-valued variables *p, q, r*, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $p \in C$, $\sim \in \{\leq, \geq, =, <, >\}$ and $n \in N^{+}$. We use *G(C)* to denote the set of clock guards. A timed automaton *A* is a tuple $\langle N, l_0, E, I\rangle$ where *N* is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \sum \times 2^{C} \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \longrightarrow G(C)$

assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n$, $n \in N^{+}$.

Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to clock conditions, the propositions on integer variables *k* are of the form $k \sim n$ for $n \in N$, and $\sim \in \{\leq, \geq, =, <, >\}$. For the formal definition of Uppaal TA semantics we refer the reader to [18] and [16].

3. TDL^{TP} SYNTAX

The ground terms in TDL^{TP} are sets (denoted *TS*) of assignments to auxiliary variables called *trap variables* or simply *traps* added to the SUT model for test purpose specification. A trap is updated by Boolean variable assignment that labels a TCI. In case of Uppaal TA, the TCIs are edges of the SUT model M^{SUT}. The value of all traps is initially set to *false*. When the edge of M^{SUT} labelled with a trap is visited during test execution the trap update function is executed and the trap value is set to *true*. We say that a trap *tr* is *elementary trap* if its update function is unconditional, i.e. of shape *tr* := *true*.

Generally we assume that the trap names are unique, trap update functions are non-recursive and their arguments have definite values whenever the edge labelled with that trap is executed. The trap *tr* update condition, if *conditional trap*, is a Boolean expression (we call it also update constraint) the arguments of which range over the sets of variables and constants of M^{SUT} and over the auxiliary constants and variables occurring in the test purpose specification in TDL^{TP}, e.g. references to other traps, event counters and the time bounds of model clocks.

Although we deal with finite sets of traps and their value domains the quantifiers are introduced in TDL^{TP} for notational convenience. To refer to the situations where many traps have to be true or false at once, we group these traps to sets called *trapsets* denoted by *TS* and prefix them with trapset quantifiers - A for universal and E for existential quantification. *A*(*TS*) means that all traps and *E*(*TS*) means that at least one trap of the set *TS* has to be true. To represent a trapset in Uppaal TA syntax we encode them as one-dimensional trap arrays and refer to individual traps in the array by array index value, e.g. *i*-th trap in *TS* is referred to as *TS*[*i*].

In the following we give the syntax of TDL^{TP} expressions in BNF:

⟨*Expression*⟩ ::=
  '(' ⟨*Expression*⟩ ')'
  | *'A'* ⟨*TrapsetExpression*⟩
  | *'E'* ⟨*TrapsetExpression*⟩
  | ⟨*UnaryOp*⟩ ⟨*Expression*⟩
  | ⟨*Expression*⟩ ⟨*BinaryOp*⟩ ⟨*Expression*⟩
  | ⟨*Expression*⟩ ~> ⟨*Expression*⟩
  | ⟨*Expression*⟩ ~>'['⟨*RelOp*⟩⟨*NUM*⟩']' ⟨*Expression*⟩
  | *'#'* ⟨*Expression*⟩ ⟨*RelOp*⟩ ⟨*NUM*⟩

⟨*TrapsetExpression*⟩ ::=
  '(' ⟨ *TrapsetExpression*⟩')'
  | *'!'* ⟨*ID*⟩
  | ⟨*ID*⟩ *'\'* ⟨*ID*⟩
  | ⟨*ID*⟩ *';'* ⟨*ID*⟩

⟨*UnaryOp*⟩	::=	*'not'*
⟨*BinaryOp*⟩	::=	*'&'* | *'or'* | *'=>'* | *'<=>'*
⟨*RelOp*⟩	::=	*'<'* | *'='* | *'>'* | *'<='* | *'>='*
⟨*ID*⟩	::=	(*'TR'*) ⟨*NUM*⟩
⟨*NUM*⟩	::=	('0'..'9')+

4. TDL^{TP} SEMANTICS

To define the semantics of *TDL^{TP}* we assume there are given:

- an Uppaal TA model *M*;
- *Trapset TS* which is possibly a union of member trapsets $TS = \bigcup_{i=1,m} TS_i$, where the cardinality of each TS_i is n_i;
- $L\colon TS \longrightarrow E(M)$, the labelling function that maps the traps in *TS* to edges in *E*(*M*), where *E*(*M*) denotes the set of edges of the model *M*. We assume the *uniqueness* of the labeling within a trapset, i.e. there is at most one edge labelled with a trap from the given trapset but an edge can be labelled with many traps if each of them is from different trapset.

4.1 ATOMIC LABELLING FUNCTION

Atomic labelling function is non-surjective and injective-only mapping between *TS* and $E(M)$, i.e. each element of *TS* is mapped to a unique edge in $E(M)$:

$$\begin{gathered} \mathrm{L{:}\ TS \longrightarrow E(M),\ s.t.\ \forall e \in E(M){:}} \\ \mathrm{TS_k[i] \in L(e) \wedge TS_l[j] \in L(e) \Rightarrow k \neq l} \end{gathered} \quad (1)$$

4.2 DERIVED LABELLING OPERATIONS (TRAPSET OPERATIONS)

The formulas with a trapset operation symbol and trapset(s) identifiers being its argument(s) are called TDL^{TP} trapset formulas.

Relative complement of trapsets $(\boldsymbol{TS_1 \backslash TS_2})$**.** Only those edges labelled with traps of TS_1 and not with traps of TS_2 are in the relative complement trapset $TS_1 \backslash TS_2$:

$$\begin{gathered} [\![TS_1 \backslash TS_2]\!]\ iff \\ \forall i \in [0, n_1], j \in [0, n_2], \exists e \in E(M){:} \end{gathered} \quad (2)$$

$$TS_1[i] \in L(e) \wedge TS_2[j] \notin L(e)$$

Absolute complement of a trapset **(** $!TS$ **).** All edges that are not labelled with traps of TS are in the absolute complement trapset $!TS$:

$$[\![!TS]\!] \; iff \; \forall i \in [0,n], \exists e \in E(M): TS[i] \notin L(e) \quad (3)$$

Linked pairs of trapsets **(** $TS_1;\ TS_2$ **).** Two trapsets TS_1 and TS_2 are linked via operator *next* (denoted ';') if and only if there exists a pair of edges in M which are labelled with traps of TS_1 and TS_2 respectively and which are connected through a location so that if any of traps in TS_1 is updated to true on the k-th transition of model M execution trace σ then some trap of TS_2 is updated to true in the (k+1)-th transition of that trace:

$$[\![TS_1;\ TS_2\ ]\!] \quad iff \quad \forall i \in [0,n_1], \exists j \in [0,n_2], \sigma, k: \; [\![TS_1[i]]\!]_{\sigma^k} \Rightarrow [\![TS_2[j]]\!]_{\sigma^{k+1}} \quad (4)$$

where $[\![TS]\!]_\sigma$ denotes the interpretation of the trapset TS on the trace σ and σ^l denotes the *l*-th suffix of the trace σ, i.e. the suffix which starts from *l*-th location of σ; n_1 and n_2 denote cardinalities of trapsets TS_1 and TS_2 respectively. Note that operator ';' enables expressing one of the "classical" structural coverage criteria 'selected transition pairs'.

4.3 INTERPRETATION OF TDL EXPRESSIONS

Quantifiers of trapsets. Given the definitions 1 - 4 of trapset operations we define the semantics of bounded universal quantifier *A* and bounded existential quantifier *E* of a trapset *TS* as follows:

$$[\![A\ (TS)]\!] \; iff \; \forall i \in [0,n]: TS[i], \quad (5)$$
$$[\![E\ (TS)]\!] \; iff \; \exists i \in [0,n]: TS[i], \quad (6)$$

where *n* denotes the cardinality of the trapset *TS*.

Note that quantification is defined on the trapsets only and not on higher level operators.

Logic connectives. Since recursive nesting of TDL^{TP} logic and temporal operators is allowed for better expressiveness we define the semantics of these higher level operators where the argument terms are not trapset formulas but derived from them using recursive nesting of logic and temporal operator symbols. Let SE, SE_1 and SE_2denote such argument sub-formulas, then

$$[\![SE_1\ \&\ SE_2\ ]\!] \quad iff \quad [\![SE_1]\!]\ and\ [\![SE_2]\!] \quad (7)$$
$$[\![SE_1\ or\ SE_2\ ]\!] \quad iff \quad [\![SE_1]\!]\ or\ [\![SE_2]\!] \quad (8)$$

$$SE_1 => SE_2 \equiv not(SE_1) \vee SE_2 \quad (9)$$

$$SE_1 <=> SE_2 \equiv (SE_1 \Rightarrow SE_2) \wedge (SE_2 \Rightarrow SE_1) \quad (10)$$

Temporal operators

'Leads to' operator $'SE_1 \leadsto SE_2'$ in TDL^{TP} is inspired by Computation Tree Logic CTL 'always leads to' operator, denoted by $'\varphi --> \psi'$ in Uppaal, which is equivalent to CTL formula $A\square(\varphi \Rightarrow A\Diamond\psi)$. Leads to expresses that after reaching the state which satisfies φ in the computation all possible continuations of this computation reach the state in which ψ is satisfied. For clarity we substitute the meta-symbols φ and ψ with non-terminals $SE_1$and SE_2 of TDL^{TP}.

$$[\![SE_1 \sim>\ SE_2\ ]\!] \;\; iff \;\; \forall\sigma, \exists k, l, k \le l: [\![SE_1]\!]_{\sigma^k} \Rightarrow [\![SE_2]\!]_{\sigma^l} \quad (11)$$

where σ^k denotes the *k*-th suffix of the trace σ, i.e. the suffix which starts from *k*-th location of σ, and $[\![SE]\!]_{\sigma^k}$ denotes the interpretation of *TS* on the *k*-th suffix of trace σ.

'Time bounded leads to' means that TS_2 must occur after TS_1 and the time instance of TS_2 occurrence (measured relative to TS_1 occurrence satisfies constraint $\circledast\ n$, where $\circledast \in \{<, =, >, \le, \ge\}$ and $n \in \mathbf{N}$:

$$[\![SE_1 \sim>_{[\circledast n]}\ SE_2]\!] \quad iff \quad \forall\sigma, \exists k, l, k \le l: [\![SE_1]\!]_{\sigma^k} \Rightarrow [\![SE_2]\!]_{\sigma^l} \quad (12)$$

'Conditional repetition'. Let k enumerate the occurrences of $[\![SE]\!]$, then

$$[\![\#SE \circledast n\ ]\!] \;\; iff \leadsto \cdots \leadsto \; [\![SE]\!]^k \; and \; k \circledast n. \quad (13)$$

where index variable k satisfies constraint $\circledast\ n$, $\circledast \in \{<, =, >, \le, \ge\}$ and $n \in \mathbf{N}$.

The application of logic *not* to non-ground level TDL^{TP} terms has following interpretation:

$$not(A\ (TS))\ iff \quad \exists i: [\![TS[i]]\!] = false \quad (14)$$
$$not(E\ (TS))\ iff \quad \forall i: [\![TS[i]]\!] = false \quad (15)$$
$$not\ (SE_1 \wedge SE_2) \equiv not\ (SE_1)\ \vee\ not\ (SE_2) \quad (16)$$
$$not\ (SE_1 \vee SE_2) \equiv not\ (SE_1)\ \wedge\ not\ (SE_2) \quad (17)$$
$$not\ (SE_1 \Rightarrow SE_2) \equiv SE_1\ \wedge\ not\ (SE_2) \quad (18)$$
$$not\ (SE_1 \Leftrightarrow SE_2 \equiv not\ (SE_1 \Rightarrow SE_2) \vee not\ (SE_2 \Rightarrow SE_1) \quad (19)$$
$$[\![not(SE_1 \leadsto SE_2)]\!]\ iff \quad (20)$$

$$[\![not\ (SE_1)]\!]\ or\ \forall k,l,k \le l: [\![SE_1]\!]_{\sigma^k}\ and\ not[\![SE_2]\!]_{\sigma^l}$$

$$not\ (SE_1 \leadsto_{\circledast n}\ SE_2) \equiv not(SE_1 \leadsto SE_2) \vee \forall\phi: (SE_1 \leadsto_{\phi}\ SE_2) \Rightarrow (\phi \Rightarrow not(\circledast\, n)), \tag{21}$$

$$not\ (\#TS\ \circledast\, n) \equiv \forall\phi: (\#TS\ \phi) \Rightarrow (\phi \Rightarrow not(\circledast\, n)) \tag{22}$$

where ϕ denotes the time bound constraint that yields the negation of constraint $\circledast\, n$.

5. MAPPING *TDL^{TP}* EXPRESSIONS TO BEHAVIOR RECOGNIZING AUTOMATA

When mapping the TDL^{TP} formulae to test supervisor component automata we implement the mappings starting from ground level terms and move towards the root term by following the structure of the TDL^{TP} formula parse tree. The terminal nodes of any TDL^{TP} formula parse tree are trapset identifiers. The next above the terminal layer of the parse tree constitute the trapset operation symbols. The trapset operation symbols, in turn, are the arguments of logic and temporal operators. The ground level trapsets and the trapsets which are the results of trapset operations are mapped to the labelling of SUT model M^{SUT}. In the following the mappings are specified for TDL^{TP} trapset operations, logic operators and temporal operators in separate subsections.

5.1 MAPPING *TDL^{TP}* TRAPSET EXPRESSIONS TO SUT MODEL *M^{SUT}* LABELLING

Mapping *M*1: Relative complement of trapsets $TS_1 \backslash TS_2$: The $TS_1 \backslash TS_2$ – mapping adds the traps of the trapset $TS_1 \backslash TS_2$ only to these edges of M^{SUT} which are labelled with traps of TS_1 and not with traps of TS_2. An example of such mapping is depicted in Fig. 2.

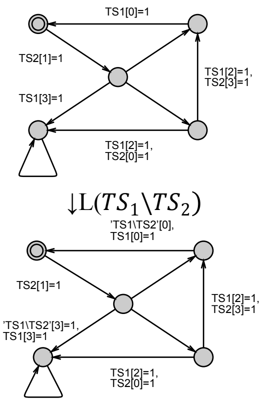


Figure 2 - Mapping TDL^{TP} expression $TS_1 \backslash TS_2$ to the SUT model labelling

Mapping *M*2: Absolute complement of a trapset !*TS*: The mapping of !*TS* to SUT model labelling provides the labelling with !*TS* traps all such edges of SUT model M^{SUT} which are not labelled with traps of *TS*. Example of this mapping is depicted in Fig. 3.

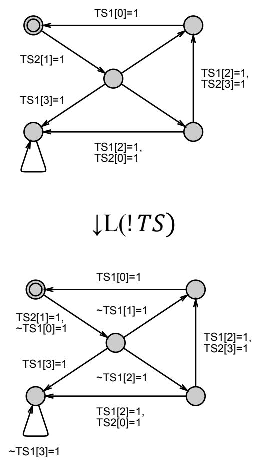


Figure 3 - Mapping TDL^{TP} expression !TS to the SUT model labelling

Mapping *M*3: Linked pairs of trapsets $TS_1; TS_2$:

The mapping of terms $TS_1; TS_2$ to labelling is implemented by the labelling algorithm *Algorithm* 1 ($L(TS_1; TS_2)$)

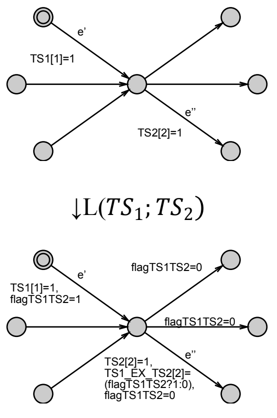


Figure 4 - Example of the application of ALGORITHM 1 ($L(TS_1; TS_2)$)

The example of Algorithm 1 application is demonstrated in Fig. 4. Notice that the labelling concerns not only the edges that are labelled with traps of *TS1* and *TS2* but also those which depart from the same location as the edge with *TS2* labelling. This is necessary for resetting the variable *flag* which indicates the executing a trapset *TS1* labelled edge in the previous step of the computation.

```
$\boldsymbol{forall}$
  $e', e'', i, j: pre(e'') = post(e') \wedge TS_1[i] \in L(e')$
  $\boldsymbol{if}\ TS_2[j] \in L(e'')$
    $\boldsymbol{then}$
      $Asg(e') \leftarrow Asg(e'), flag(TS_1; TS_2)$
                $= true,$
      $Asg(e'') \leftarrow Asg(e''), TS(TS1; TS2)[j]$
      $= (flag(TS_1; TS_2)?\, true: false),$
  $\boldsymbol{fi}$
  $Asg(e'') \leftarrow Asg(e''), flag(TS_1; TS_2) = false$
$\boldsymbol{end\ forall}$
```

5.2 MAPPING TDL^{TP} LOGIC OPERATORS TO RECOGNIZING AUTOMATA

The indexing of trapset array elements, universal and existential quantifiers in Uppaal modelling language support direct mapping of trapset quantifiers to *forall* and *exists* expressions of Uppaal TA as shown in Fig. 5 and 6.

Mapping M4: Universal quantifier of the trapset

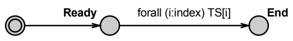


Figure 5 - An automaton that recognizes universally quantified trapset expressions

Mapping M5: Existential quantifier of the trapset

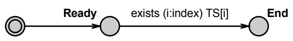


Figure 6 - The automaton that recognizes existentially quantified trapset expressions

Negation not

Since logic negation *not* can be pushed to ground level trapset terms by applying equivalences (14 – 22) and the direct mappings of *not* formulas are not considered in this work.

Mapping M6: Conjunction of sub-formulas

The conjunction *SE*1 & *SE*2 is mapped to the automata fragment as shown in Fig. 7. In the conjunction and disjunction automata depicted in the Fig. 7 and 8 the guard conditions *P* and *Q* encode the argument terms *SE*1 and *SE*2 respectively. In conjunction automaton the *End* location is reachable from the initial location *Idle* if both *P* and *Q* evaluate to true in any order.

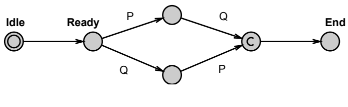


Figure 7 - The automaton that recognizes the conjunction of TDL^{TP} formulas P and Q

Mapping M7: Disjunction of sub-formulas

In the disjunction automaton the *End* location is reachable from the initial location *Idle* if either *P* and *Q* are true.

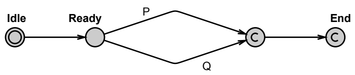


Figure 8 - Automaton that recognizes the disjunction of TDL^{TP} formulas P and Q

The *implication* of *TDL^{TP}* formulas can be defined using disjunction and negation as shown in formula (9) and their transformation to property automata are implemented through these mappings.

Similarly, the *equivalence* of TDL^{TP} formulas can be expressed via conjunction and implication by using equivalence in formula (10).

5.3 MAPPING TDL^{TP} TEMPORAL OPERATORS TO RECOGNIZING AUTOMATA

Mapping M8: 'Leads to' $p \rightsquigarrow q$

Mapping the *leads to* operator to Uppaal TA produces the model fragment depicted in Fig. 9.

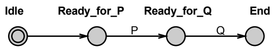


Figure 9 - 'Leads to' formula $p \rightsquigarrow q$ recognizing automaton

Mapping M9: Timed leads to $p \rightsquigarrow_{\circledast con} q$

Mapping *'timed leads to'* to a Uppaal TA fragment is depicted in Fig. 10. It presumes an additional clock cl which is reset to 0 at the time instant when formula *P* become true. The condition 'cl<=d' in Fig. 10 a) sets the upper time bound d to the event when formula *Q* has to becomes true after *P*, i.e. after the clock cl reset.

a)

b)

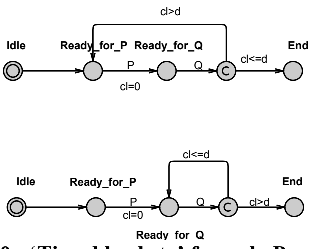


Figure 10 - 'Timed leads to' formula P $\rightsquigarrow_{\circledast d}$ Q recognizing automata a) with condition cl≤d; b) with condition cl>d

The mapping to property automaton depends on the time condition of *leads to*. For instance if the conditions is 'cl>d' the mapping results in automaton shown in Fig. 10 b).

Mapping M10: Conditional repetition $\#SE \circledast n$:

The Uppaal TA fragment generated by the mapping of $\#SE \circledast n$ (Fig. 11) includes a counter variable i to enumerate the events when the *SE* formula *P* becomes *true*. If the loop exit condition, e.g., ‘i >=n’, is satisfied then the transition to location *End* is fired without delay (the middle location is of type *committed*).

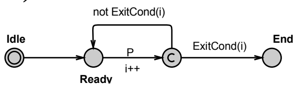


Figure 11 - Uppaal TA that implements conditional repetition

6. REDUCTION OF THE SUPERVISOR AUTOMATA AND THE LABELLING OF SUT

The TDL^{TP} expressions with many nested operators may become large and involve some overhead. Removal of this overhead in the formulas provides reduction in the state space needed for their model checking and improves the readability and comprehension of this formula.

The simplifications are formulated in terms of the parse tree of the TDL^{TP} formula and standard logic simplifications. Due to the nesting of operations in the TDL^{TP} formula the root operation can be any operator listed in the BNF grammar of TDL^{TP} but the terminals of the parse tree are always trapsets.

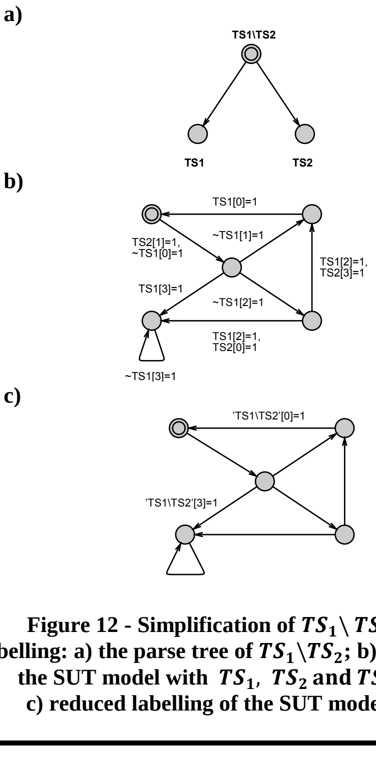


Figure 12 - Simplification of $TS_1 \setminus TS_2$ trapsets labelling: a) the parse tree of $TS_1 \setminus TS_2$; b) labelling of the SUT model with TS_1, TS_2 and $TS_1 \setminus TS_2$ c) reduced labelling of the SUT model M^{SUT}

TDL^{TP} formulas consist of a static component (a trapset or a trapset expression) and optionally the logic and/or temporal component. The static component includes all sub-formulas of the parse tree branches from terminals to the lowest temporal expression, all sub-formulas above it are temporal and/or logic formulas (possibly mixed).

The trapset formulas are implemented by labelling operations such as *relative* and *absolute complement*. Only trapset formulas can be universally and existentially quantification. No nesting of quantifiers is allowed. Since the validity of root formula can be calculated only using the truth value of the highest trapset expression in the parse tree, all the trapsets being closer to the ground level trapset along the parse tree sub branch can be removed from the labelling of the SUT model. This reduction can be done after labelling the SUT model and applying all the trapset operations. An example of such reduction is demonstrated for relative complement operation $TS_1 \setminus TS_2$ in Fig. 12.

Logic simplification follows after the trapset expression simplification is completed. Here standard logic simplifications are applicable:

$$\begin{aligned}
&p \wedge p \equiv p\\
&p \wedge not\ p \equiv false,\\
&p \wedge false \equiv false,\\
&p \wedge true \equiv p,\\
&p \vee p \equiv p, \qquad (13)\\
&p \vee not\ p \equiv true,\\
&p \vee false \equiv p,\\
&p \vee true \equiv true,
\end{aligned}$$

We will introduce also a set of simplifications for TDL^{TP} temporal operators which follow from their semantics and the properties of integer arithmetic:

$$\begin{aligned}
&TS \equiv false\ if\ TS = \emptyset\\
&p \rightsquigarrow false \equiv false\\
&false \rightsquigarrow p \equiv false\\
&true \rightsquigarrow p \equiv p\\
&p \rightsquigarrow true \equiv true\\
&\#p = 1 \equiv p\\
&\#p \circledast n_1 \wedge \#p \circledast n_2\\
&\qquad \equiv \#p \circledast max(n_1, n_2)\ if\\
&\qquad \circledast \in \{\geq, >\} \qquad (14)\\
&\#p \circledast n_1 \vee \#p \circledast n_2\\
&\qquad \equiv \#p \circledast min(n_1, n_2)\ if\ \circledast\\
&\qquad \in \{\geq, >, =\}\\
&\#p \circledast n_1 \wedge \#p \circledast n_2 \equiv false\ if\ \circledast\\
&\qquad \in \{=\}\ and\ n_1 \neq n_2\\
&\#p \circledast n_1 \rightsquigarrow \#p \circledast n_2\\
&\qquad \equiv \#p \circledast (n_1 + n_2) \quad if\ \circledast\\
&\qquad \in \{\geq, >, =\}
\end{aligned}$$

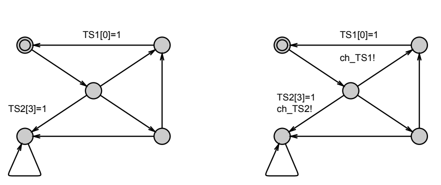
$$\#p \circledast n_1 \rightsquigarrow \#p \circledast n_2 \equiv \#p \circledast min(n_1, n_2) \ if \ \circledast \in \{<\}$$
$$\#p \circledast n_1 \wedge \#p \circledast n_2 \equiv \#p \circledast min(n_1, n_2) \ if \ \circledast \in \{<\}$$
$$\#p \circledast n_1 \vee \#p \circledast n_2 \equiv \#p \circledast max(n_1, n_2) \ if \ \circledast \in \{<\}$$
$$p \rightsquigarrow_{d_1} q \wedge p \rightsquigarrow_{d_2} q \equiv p \rightsquigarrow_{\min(d_1,d_2)} q \quad if \ \circledast \in \{\leq, <\}$$
$$p \rightsquigarrow_{d_1} q \wedge p \rightsquigarrow_{d_2} q \equiv p \rightsquigarrow_{\max(d_1,d_2)} q \quad if \ \circledast \in \{>\}$$

7. COMPOSING THE TEST SUPERVISOR MODEL

The test supervisor model M^{SVR} is constructed as a parallel composition of single TDL^{TP} property recognizing automata each of which is produced by parsing the TDL^{TP} formula and mapping corresponding sub-formulae to the automaton template as defined in Section 5. To interrelate these sub-formula automata, two phases have to be completed:

1) Each trap labelled transition *e* of M^{SUT} (here we consider the traps which are left after labels reduction as described in Section 6) has to be split in two edges *e*' and *e*" connected via an auxiliary committed location l^c. The edge *e*' will inherit the labelling of *e* while *e*" will be labelled with an auxiliary broadcast channel that signals the trap update occurrence to the upper neighbor sub-formula automaton. We use the channel naming convention where a channel name has a prefix *ch_* followed by the trapset identifier, e.g. for an edge *e* labelled with the trap *TS*[*i*], the broadcast channel label *ch_TS*! is added to the edge *e*" (an example is shown in Fig. 13 a)).
2) Each non-trapset formula automaton will be extended with a wrapping construct shown in Fig. 13 b). The wrapper has one or two, channel labels, depending if the sub-formula operation is unary or binary, to synchronize its state transition with those of its child expression(s). We call them downwards channels denoted by *Activate_subOP1*, *Activate_subOP2* and used to activate the recognizing mode in the first and second sub-formula automata. Similarly, two broadcast channels are introduced to synchronize the state transition of sub-formula automata with their upper operation automaton. We call them upwards channels, denoted by *Activate_OPi* and *Done OPi* in Fig. 13 b). The root node is an exception because it has upwards channel only with the test *Stopwatch* automaton (the *Stopwatch* automaton will be explained in Section 8). If the sub-formulas of given property automaton are mapped to trapset expressions then the back edge *End*→*Idle* to the initial state is labelled also with trapset reset function with *TS* being the argument trapset identifier. The TDL^{TP} operator automata extensions with wrapper constructs for implementing their composition in test supervisor model M^{SVR} are shown in Fig. 14.

a)



b)

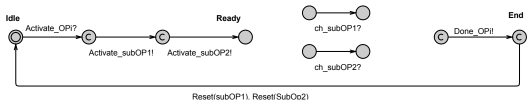
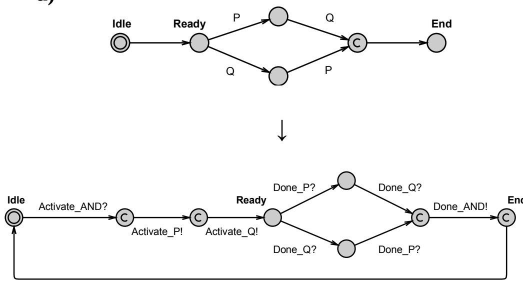
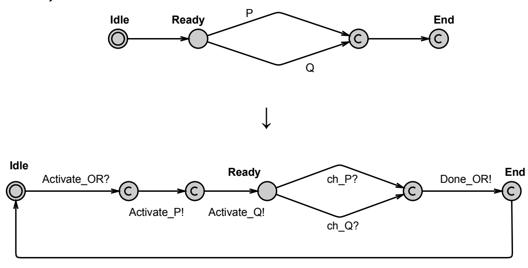


Figure 13 - a) Extending the trap labelled edges with synchronization conditions for composing the test supervisor; b) the wrapper pattern for composing operation recognizing automata

a)



b)



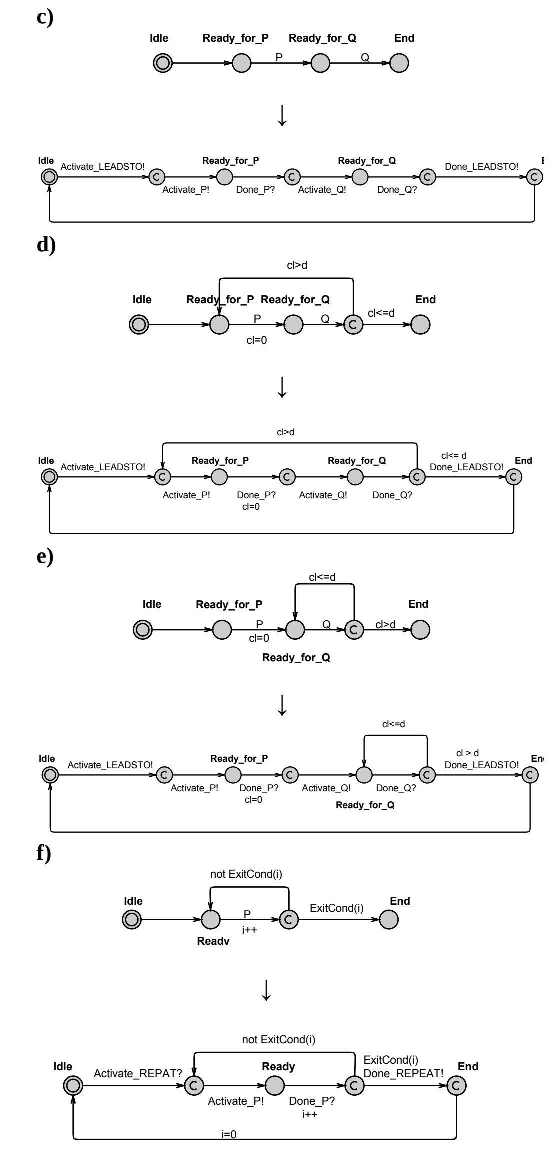


Figure 14 - Extending sub-formula automata templates with wrapping for test Supervisor composition a) And; b) Or; c) Leads to; d) Timed leads to with condition cl≤d; e) Timed leads to with condition cl>d; f) Conditional repetition

Note that the TDL^{TP} sub-formula meta symbols *P* and *Q* in the original templates are replaced with channels which signal when the sub-formulas interpretation automata reach their local *End* locations.

8. ENCODING THE TEST VERDICT AND TEST DIAGNOSTICS IN THE TESTER MODEL

The test verdict is yielded by the test *StopWatch* automaton either when the automaton reaches its end state *End* within time bound *TO*. Otherwise, the *timeout* event *Swatch==TO* triggers the transition to the terminal location *Failed*. Specifically, *Passed* in the *StopWatch* automaton is reached simultaneously with executing the test purpose formula φ^{TP} automaton transition to its *End* location. For example, in Fig. 15, the automaton that implements root formula *P*, synchronizes its transition to the location *End* with *StopWatch* transition to the location *Passed* via upwards channel *Done_P*.

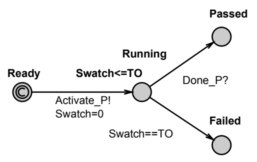


Figure 15 - Test Stopwatch automaton.

Another extension to *the supervisor model* is the capability of recording the test diagnostic information. For that each sub-formula of the test purpose specification formula φ^{TP} is indexed according to its position in the parsing tree of φ^{TP}. A diagnostic array *D* of type Boolean and of the size equal to the number of sub-formulas in φ^{TP} is defined in the model. The initial valuation of *D* sets all its elements to *false*. Whenever a model fragment that corresponds to a sub-formula reaches its end state (that is sub-formula satisfaction state), the element in *D* that corresponds to that sub-formula is set to *true*. It means that if the test passes, all elements of *D* are updated to *true*. Otherwise, in case the test fails, those elements of *D* remain *false* which correspond to the sub-formula automata which conditions were not satisfied or reached by the test model run. The updates *D*[*i*]:= *true* of array *D* elements, where *i* is the index of the sup-formula automaton $M^{op}{}_i$, are shown on the edges that enter their *End* locations. The expression automata $M^{op}{}_i$ and their mapping to composition wrapping are shown in Fig. 14.
The test model construction steps can be summarized now as follows:

1. the test purpose is specified as a TDL^{TP} expression φ^{TP};
2. trapsets $TS_1,\ldots, TS_n$ are extracted from φ^{TP} and the ground level TCIs are labelled with elements of non-intersecting trapsets;
3. the parse tree of the *TDL^{TP}* expression φ^{TP} is analysed and each of its sub-formula operator op_i is mapped using the mappings *M*1 to *M*10 to the automaton template $M^{op}{}_i$ that corresponds to the sub-formula operation;

4. the labelling of M^{SUT} with traps is simplified by applying rules in Section 4.6, and M^{SUT} linked with sub-formula automata $M^{op}{}_i$ via wrapping construct that provides channels for signalling about reaching the state where sub-formula are satisfied;
5. finally, the extension for collecting diagnostics is added to automata $M^{op}{}_i$ and the root formula automaton is composed with *Stopwatch* automaton M^{SW} which decides on the test pass or fail.

The total test model is synchronous parallel composition of component models $M^{SUT} || M^{SW} ||_i M^{op}{}_i$.

9. CASE STUDY

To demonstrate the usability of TDL^{TP} the TTU100 satellite testing case study has been chosen. The objective of the TTU100 project is to build a space system consisting of a 1U (10 cm x 10 cm x 10 cm) nanosatellite and a ground station where mission planning and mission control software for scientific experiments is installed. The TTU100 system consists of a Ground Segment and a Space Segment. The Ground Segment communicates, stores and processes data aquired from satellite. The Space Segment is nanosatellite on Earth's Sun Synchronous Orbit (650km altitude). The satellite onboard system consists of smart electrical power supply (EPS), attitude determination and control system (ADCS), on-board computer (OBC), communication system (UHF band, Ku-band) and camera and optics payload.

For TDL^{TP} usability demonstration smart EPS subsystem is selected as a SUT. The test purpose is specified for a test case which demonstrates the TDL^{TP} capability to express combinations of multiple coverage criteria in a single test case. From TDL^{TP} expressions the test models are constructed and the test sequences generated using Uppaal model checker. The section is concluding with comparison of the tests generated with the methods presented in the paper and with those available using ordinary TCTL model checking.

9.1 SYSTEM UNDER TEST MODELLING

EPS receives commands from other system components to change its operation mode and respond with its status information. In the integration level test model we abstract from the concrete content of the commands and responses and describe its interface behavior in response to input commands.

EPS is sampling its input periodically with period 20 time units. EPS wakeup time when detecting a new input command can vary within interval [15, 20] time units after previous sampling. After wakeup it is ready to receive incoming commands. Due to internal maintenance procedures of EPS some of the commands when sent during self-maintenance can be ignored, and need to be repeated later. The command processing after its successful receive takes at most 20 time units. Thereafter, the validity of the command is checked using CRC error-detecting code. If the error is detected the error report will be sent back to EPS output port in *o_response* message. If the received command data is correct, the command is processed and its results returned in the outgoing *o_message*. Since EPS internal processing time is negligible compared to that of input sampling period and wakeup time, all the other locations except *start* and *commandCreated* are modelled as committed locations. The model M^{SUT} of the EPS is depicted in Fig. 16.

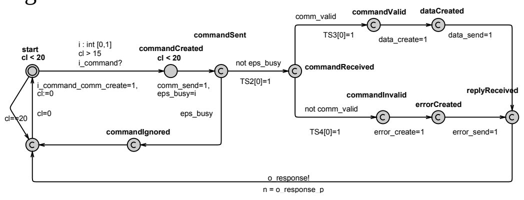


Figure 16 - The model M^{SUT} of the Electrical Power Supply subsystem.

9.2 TEST PURPOSE SPECIFICATION

The goal of test case is to show that after invalid command has been received the valid command can be received correctly and responded with acknowledgement. We specify the test purpose in TDL^{TP} as formula

$$A(TS2; TS4) \sim> E(TS2; TS3), \qquad (25)$$

which expresses that all transition pairs labelled with traps of TS$_2$ and TS$_4$ must lead to some pair of transitions labelled with traps of trapsets TS$_2$ and TS$_3$.

9.3 LABELLING OF M^{SUT}

The labelling of M^{SUT} starts from the ground level trapsets TS$_2$, TS$_3$ and TS$_4$ of the formula (25). These traps guide branching conditions to be satisfied in the test scenario. The labelling is shown in Fig. 16.

Second level labelling results in applying trapset operation next ';' for pairs *TS$_2$;TS$_3$* and *TS$_2$;TS$_4$* which presumes introducing auxiliary variables *fl23* and *fl24* to identify occurrence of traps of TS$_3$ and TS$_4$ right after traps of TS$_2$. Since *TS$_2$;TS$_3$* and *TS$_2$;TS$_4$* are arguments of the upper '*forall*' and '*exists*' formula their occurrence should be signaled respectively to '*forall*' and '*exists*' automata. For this purpose additional committed locations and edges with upwards channels *ch_TS23* and *ch_TS24* are introduced in Fig. 17.

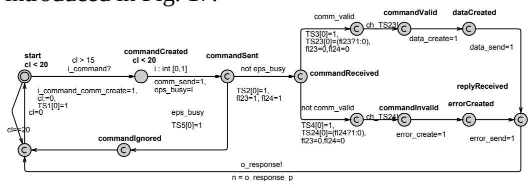


Figure 17 - Marking *TS₂;TS₃* and *TS₂;TS₄* trapsets

9.3 TEST MODEL CONSTRUCTION

When moving upwards in the parse tree of formula (25) the next operators that have $TS_2;TS_4$ and $TS_2;TS_3$ in arguments are forall *A(TS2;TS4)* and exists *E(TS2;TS3)* which automata are depicted in Fig. 18.

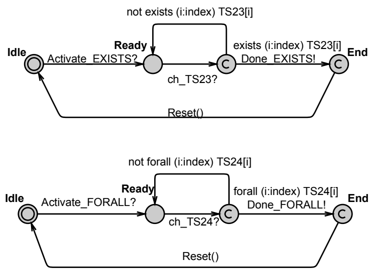


Figure 18 - a) automation that recognizes A(TS2;TS4); b) automation that recognizes E(TS2;TS3) respectively

The root operator in the formula (25) is '*leads to*' the arguments of which are *A(TS2;TS4)* and *E(TS2;TS3)*. The automaton that recognizes *A(TS2;TS4) ~> E(TS2;TS3)* is depicted in Fig. 19.

The full test model for generating test sequences of test scenario *A(TS2;TS4) ~> E(TS2;TS3)* is composed of automations shown in Fig. 17, Fig. 18, Fig.19 and Fig. 20.

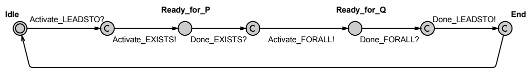


Figure 19 - Recognizing automaton of A(TS2;TS4) ~> E(TS2;TS3)

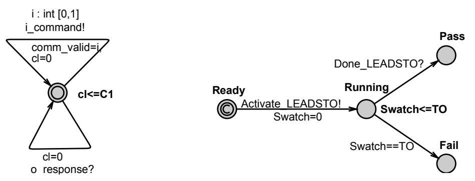


Figure 20 – Automaton for Environment and StopWatch of Test model for implementing test scenario A(TS2;TS4) ~> E(TS2;TS3)

9.4 GENERATING TEST SEQUENCES

The test sequences of the SUT model M^{SUT} shown in Fig. 16 and of the scenario *A(TS2;TS4) ~> E(TS2;TS3)* are generated by running the model checking query *E<> StopWatch.Pass*. There are three options of selecting the trace for test - *shortest*, *fastest*, or *some*. The trace generated with model checking option *shortest* is shown in the Fig.21.

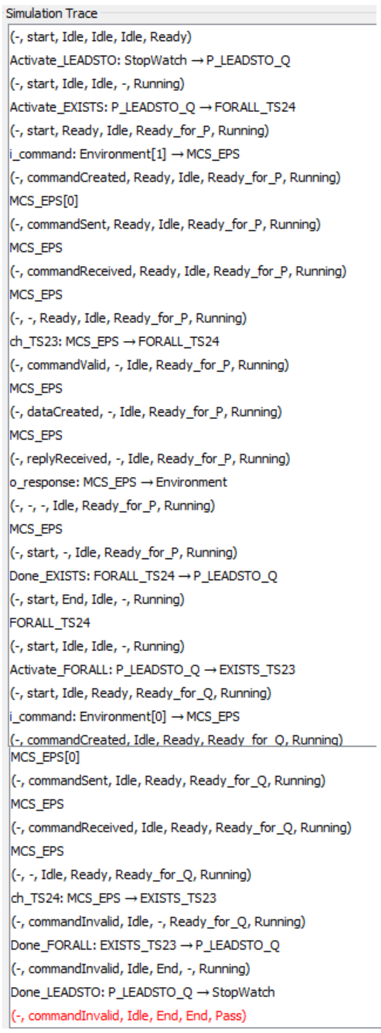

```
Simulation Trace
(-, start, Idle, Idle, Idle, Ready)
Activate_LEADSTO: StopWatch → P_LEADSTO_Q
(-, start, Idle, Idle, -, Running)
Activate_EXISTS: P_LEADSTO_Q → FORALL_TS24
(-, start, Ready, Idle, Ready_for_P, Running)
i_command: Environment[1] → MCS_EPS
(-, commandCreated, Ready, Idle, Ready_for_P, Running)
MCS_EPS[0]
(-, commandSent, Ready, Idle, Ready_for_P, Running)
MCS_EPS
(-, commandReceived, Ready, Idle, Ready_for_P, Running)
MCS_EPS
(-, -, Ready, Idle, Ready_for_P, Running)
ch_TS23: MCS_EPS → FORALL_TS24
(-, commandValid, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, dataCreated, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, replyReceived, -, Idle, Ready_for_P, Running)
o_response: MCS_EPS → Environment
(-, -, -, Idle, Ready_for_P, Running)
MCS_EPS
(-, start, -, Idle, Ready_for_P, Running)
Done_EXISTS: FORALL_TS24 → P_LEADSTO_Q
(-, start, End, Idle, -, Running)
FORALL_TS24
(-, start, Idle, Idle, -, Running)
Activate_FORALL: P_LEADSTO_Q → EXISTS_TS23
(-, start, Idle, Ready, Ready_for_Q, Running)
i_command: Environment[0] → MCS_EPS
(-, commandCreated, Idle, Ready, Ready_for_Q, Running)
MCS_EPS[0]
(-, commandSent, Idle, Ready, Ready_for_Q, Running)
MCS_EPS
(-, commandReceived, Idle, Ready, Ready_for_Q, Running)
MCS_EPS
(-, -, Idle, Ready, Ready_for_Q, Running)
ch_TS24: MCS_EPS → EXISTS_TS23
(-, commandInvalid, Idle, -, Ready_for_Q, Running)
Done_FORALL: EXISTS_TS23 → P_LEADSTO_Q
(-, commandInvalid, Idle, End, -, Running)
Done_LEADSTO: P_LEADSTO_Q → StopWatch
(-, commandInvalid, Idle, End, End, Pass)
```

Figure 21 – Test Case test sequence

The lenght of the trace generated by using TDL^{TP} is 22 transitions and the average lenght generated using ordinary TCTL model checking is 50 transitions.

9. CONCLUSION

In this paper high level test purpose specification language TDL^{TP}, its syntax and semantics have been defined for model-based testing of time critical systems. Based on the semantics proposed in this work a mapping from TDL^{TP} to Uppaal TA formalism has been defined. The mapping is used for automatic construction of test models that are composition of a SUT model and the tester model derived from the test purpose specification in TDL^{TP}. Practical side effect of the proposed test generation technique is the diagnosis capability enabling tracing back the specification sub-formula which violation by SUT behavior could cause the test fail. The application of TDL^{TP} based test generation approach on the TTU100 satellite power supply system case study confirmed also our expectations that complex multi-purpose test goals can be specified using TDL^{TP} in compact and comprehensible way saving from time consuming and error prone manual test scripting.

ACKNOWLEDGEMENT

This research was partially supported by the Estonian Ministry of Education and Research institutional research grant no IUT33-13.

10. REFERENCES

[1] Object Management Group (OMG): CCDL whitepaper.” Razorcat Technical Report, January 2014. [Online]. Available: http://www.razorcat.eu/PDF/Razorcat_Technical_Report_CCDL_Whitepaper_02.pdf .

[2] Robot framework: https://robotframework.org.

[3] T. Pajunen, T. Takala, and M. Katara, “Model-based testing with a general purpose keyword-driven test automation framework,” in 4th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST 2012, IEEE Computer Society, 2011, pp. 242–251.

[4] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, Y. Fusero, and M. Schieber, “A meta-model for tests of avionics embedded systems,” in MODELSWARD 2013 – Proc. of the 1st Int. Conf. on Model-Driven Engineering and Software Development, S. Hammoudi, L. F. Pires, J. Filipe, and R. C. das Neves, Eds. SciTePress, 2013, pp. 5–13.

[5] J. Grossmann and W. Müller, “A formal behavioral semantics for testml,” 2nd Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2006, pp. 441–448, 2006.

[6] “Iso: Road vehicles - open test sequence exchange format - part 3: Standard extensions and requirements.” International ISO multipart standard No. 13209â˘AS3 (2012).

[7] “Iso/iec: Information technology - open systems interconnection – conformance testing methodology and framework - part 1: General concepts.” International ISO/IEC multipart standard No. 9646 (1994/S1998).

[8] ITU Recommendation Z.120: Message sequence chart (MSC), 02/11. [Online] Available: http://www.itu.int/rec/T-REC-Z.120-201102-I/en.

[9] ITU Recommendation Z.120: Annex B: Formal Semantics of Message Sequence Chart (MSC), 04/98. [Online] Available: http://www.itu.int/rec/T-REC-Z.120-199804I! AnnB/en.

[10] “ETSI: TDL” [Online] Available: http://www.etsi.org/deliver/etsi_tr/103100_103199/103119/01.01.01_60/tr_103119v010101p.pdf.

[11] F. Bouquet, C. Grandpierre, B. Legeard, F.Peureux, N. Vacelet, and M. Utting, “A subset of precise UML for model-based testing,” in Proc. of the 3rd WS on Advances in Model Based Testing, A-MOST 2007, co-located with the ISSTA 2007, ACM, 2007, pp. 95–104.

[12] M. P. et al., “Evolving the etsi test description language. in: Grabowski J., Herbold S. (eds) System analysis and modeling. technology-specific aspects of models. SAM 2016. LNCS, vol 9959. Springer.”

[13] “Etsi es 202 553: Methods for testing and specification (mts).” TPLan: A notation for expressing Test Purposes, v1.2.1. ETSI, Sophia-Antipolis, France, June 2009.

[14] A. David, K. G. Larsen, S. Li, and B. Nielsen, “A game-theoretic approach to real-time system testing,” in Design, Automation and Test in Europe, DATE 2008, in: D. Sciuto (Eds.), ACM, 2008, pp. 486–491.

[15] A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, “Uppaal SMC tutorial,” STTT, vol 17, no 4, pp. 397–415, 2015.

[16] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, Springer, Heidelberg (2004), vol. 3098, pp. 87–124.

[17] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, Testing Real-Time Systems Using UPPAAL. In: R. Hierons, J. Bowen, M. Harman (Eds.) LNCS, Springer, Heidelberg (2008), vol. 4949, pp. 77-117.

[18] G. Behrmann, A. David, K. G. Larsen, A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, Springer, Heidelberg (2004), vol. 3185, pp. 200-236.

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

***MSc. Evelin Halling**, has received MSc degree in Computer Science from Tallinn University of Technology in 2011. Currently, PhD student at the Dep. of Software Science, Tallinn University of Technology. Her research interests include formal methods, software testing, model-based testing, robotics and machine learning.*

***Dr. Jüri Vain,** graduated in System Engineering from Tallinn Polytechnic Institute, Estonia in 1979. He received his PhD in Computer Science from the Estonian Academy of Sciences in 1987. Currently, he is Prof. of Computer Science at the Dep. of Software Science, Tallinn University of Technology. His research interests include formal methods, model-based testing, cyber physical systems, human computer interaction, autonomous robotics, and artificial intelligence.*

***Dr. Artem Boyarchuk** is an associate prof. at the computer systems, networks and cybersecurity department of the National aerospace university n. a. N. E. Zhukovsky "Kharkiv Aerospace Institute". He has received M.S. degree in Computer Engineering from the "Kharkiv Aviation Institute" in 2005 and defended a PhD in 2012. His expertise is in models and methods for availability assessment of service-oriented infrastructures for business-critical applications.*

***Dr. Oleg Illiashenko** is a senior lecturer at the computer systems, networks and cybersecurity dep. of the National aerospace university n. a. N. E. Zhukovsky "Kharkiv Aerospace Institute". He has received M.S. degree in Computer Engineering from the Kharkiv Aviation Institute in 2012 and Sp.Ed. in Information and communication systems security from the Kharkiv National University of Radio Electronics, Ukraine in 2014 and defended a PhD in 2018. His expertise is in models, methods and instrumentation tools for information security and cyber security assessment, evaluation and assurance of cyber security of software and hardware, dependability and resilience of embedded, web, cloud and IoT systems.*

Appendix 5

Publication II

J. Vain, A. Anier, and E. Halling. Provably correct test development for timed systems. In H. Haav, A. Kalja, and T. Robal, editors, *Databases and Information Systems VIII - Selected Papers from the Eleventh International Baltic Conference, DB&IS 2014, 8-11 June 2014, Tallinn, Estonia*, volume 270 of *Frontiers in Artificial Intelligence and Applications*, pages 289–302. IOS Press, 2014

Databases and Information Systems VIII
H.-M. Haav et al. (Eds.)
© 2014 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-458-9-289

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. $Inv : L \rightarrow I(Cl,V)$ is a function that assigns an invariant to each location, $I(Cl,V)$ is the set of invariants over clocks Cl and variables V. $T_L \rightarrow \{ordinary, urgent, committed\}$ 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 $val_cl : Cl \to \mathbb{R}_{\geq 0}$ 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.

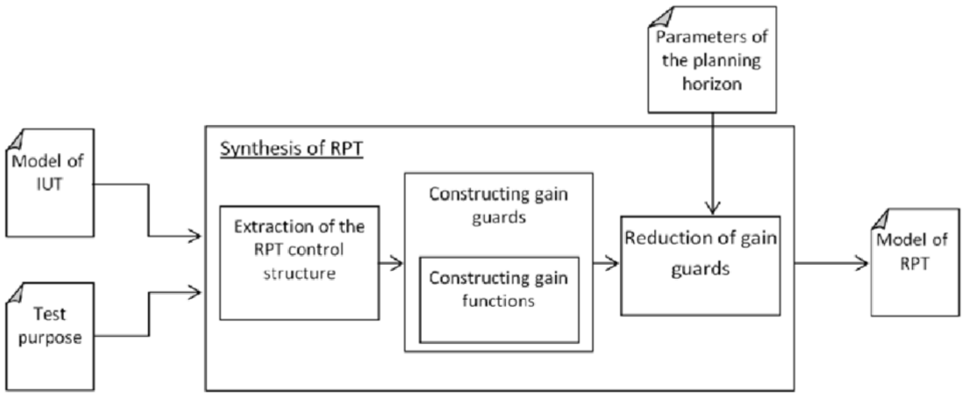

Parameters of the planning horizon
Synthesis of RPT
Model of IUT
Test purpose
Extraction of the RPT control structure
Constructing gain guards
Constructing gain functions
Reduction of gain guards
Model of RPT

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.

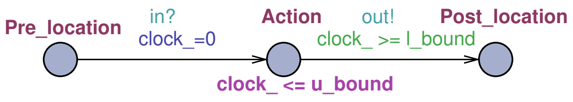


Figure 2. Elementary modelling fragment "Action"

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 "$Pre\_location \to Action$" 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 "$Action \to Post\_location$" 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 $Action \to Post\_location$ 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.

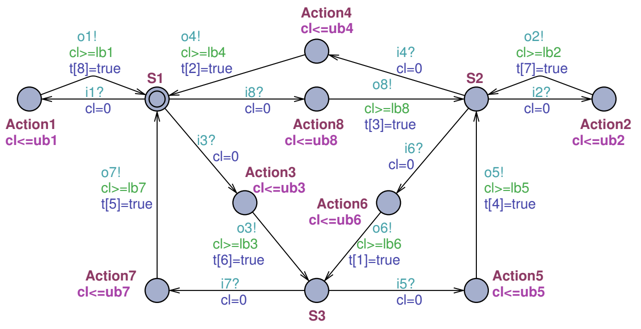


Figure 3. Simple example of well-formed IUT model

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).

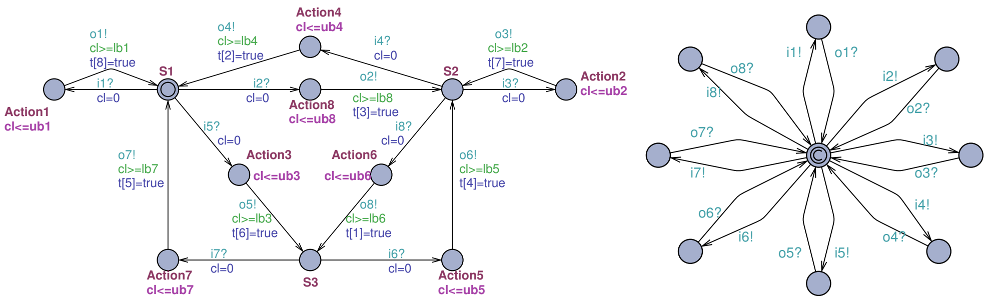


Figure 4. Synchronous parallel composition of a) IUT and b) canonical tester models

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 $I_1 \ldots I_n \in \Sigma$. 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 $I_1 \ldots I_k \in \Sigma^i(Action_i) \subseteq \Sigma$;
- 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: $not(I_1 \vee \ldots \vee I_k)$.

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).

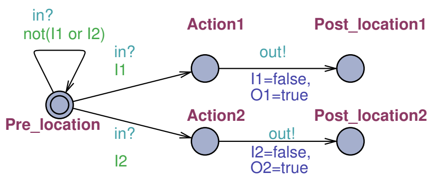


Figure 5. Input enabled fragment

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:

$$A \diamond \forall (i : int[1,n])\, t[i]\,, \tag{2}$$

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

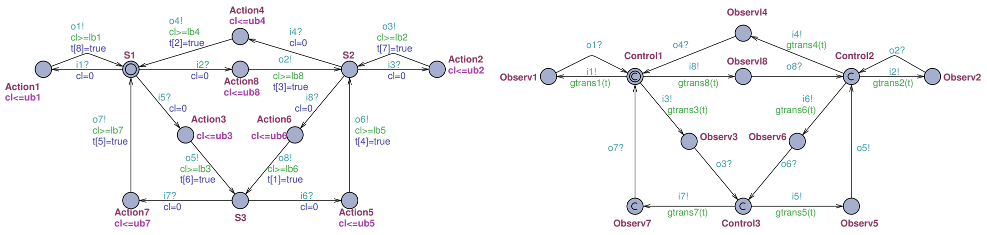


Figure 6. Synchronous parallel composition of IUT and RPT models

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.

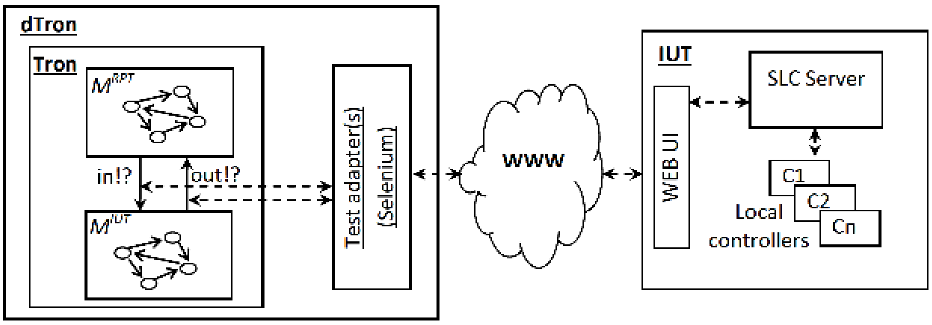


Figure 7. Street light control system test architecture

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.

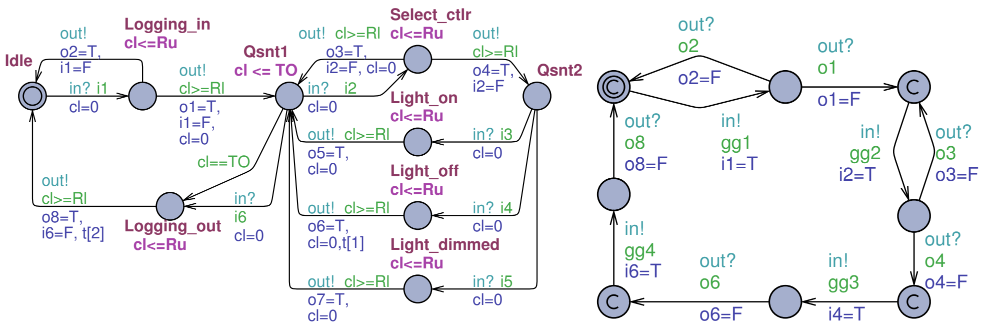


Figure 8. IUT and RPT models

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)

Appendix 6

Publication III

J. P. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ROS packages: A mobile robot case study. In *2015 European Conference on Mobile Robots*, *ECMR 2015*, *Lincoln*, *United Kingdom*, *September 2-4*, *2015*, pages 1–7. IEEE, 2015

Model-based integration testing of ROS packages: a mobile robot case study

Juhan Ernits, Evelin Halling, Gert Kanter and Jüri Vain

***Abstract*—We apply model-based testing – a black box testing technology – to improve the state of the art of integration testing of navigation and localisation software for mobile robots built in ROS. Online model-based testing involves building executable models of the requirements and executing them in parallel with the implementation under test (IUT). In the current paper we present an automated approach to generating a model from the topological map that specifies where the robot can move to. In addition, we show how to specify scenarios of interest and how to add human models to the simulated environment according to a specified scenario. We measure the quality of the tests by code coverage, and empirically show that it is possible to achieve increased test coverage by specifying simple scenarios on the automatically generated model of the topological map. The scenarios augmented by adding humans to specified rooms at specified stages of the scenario simulate the changes in the environment caused by humans. Since we test navigation at coordinate and topological level, we report on finding problems related to the topological map.**

I. Introduction

The software for robots gets increasingly complex as computational resources keep increasing at reduced power budgets. Thus it has become possible to develop software that enables robots to cope in realistic human environments. The current research in, e.g. long term behaviour of mobile robots is concerned with changing environments involving humans and human interaction with robots. The research targets specific scenarios e.g. involving recurring robot behaviour over time in dynamic environments as in [14], [1], and techniques to reason about changing scenes, as in [16], [15]. Such problems stem from the dynamic nature of human environments and the need for robots to cope in them.

While the current robotics research advances the frontiers of what can be achieved by robots, we are aware of relatively moderate amount of work done on how to test robot software to ensure that such solutions are robust and actually work as expected. Evaluation and testing such software is often achieved by running extended tests on real hardware and in simulation. But how much testing is enough? When different development teams develop separate components, how can the influence of a changeset on the overall system behaviour be efficiently evaluated?

Most contemporary software for robots uses some kind of data sharing framework to fascilitate interconnection of sensors, various data processing nodes and actuators. While there exist several such frameworks, we target ROS [18] as a representative and widely used such framework.

Department of Computer Science, Tallinn University of Technology Akadeemia tee 15a, 12618 Tallinn, Estonia `firstname.lastname@ttu.ee`

978-1-4673-9163-4/15/$31.00 © 2015 IEEE

The primary focus of the current paper is how to improve the state of the art of integration testing ROS packages involved in high level robot control, such as e.g. localisation and navigation of mobile robots.

In order not to delve into the very basics of integration testing, it is useful to assume that there is some kind of integration testing system in place, e.g. Jenkins [13]. Jenkins attempts to build all software that gets uploaded to the repository and run the existing tests. When the tests pass, the Jenkins instance can be instructed to upload the binaries to a distribution site. Our goal is to support such integration testing scenario and provide feedback whether some lines of code, e.g. the ones that just got updated, were active in certain test scenarios or not.

We take the approach of Robot Unit Testing [6] and extend it in two ways: first we introduce a white box metric of code coverage, in particular statement and branch coverage, as a quality measure of the tests. Second, we combine a technique called model-based testing [19] into the test setup, that allows us to formalise the requirements of the system into a formal model and check the conformance of the formalised requirements to the *implementation under test* (IUT), in our case the appropriate stack of ROS packages together with either a real or simulated set of sensors and actuators.

The experiments involve modelling and testing the navigation and localisation components of the software stack developed in the STRANDS[1] project. The stack was chosen because it involves multiple layers of functionality on top of the standard ROS `move_base` mobile base package that is responsible for accomplishing navigation, it is open sourced, accessible on GitHub, contains a working simulation environment built using Morse [7], and many existing quality assurance techniques are actively used in the project, including unit tests and a Jenkins based continuous integration system.

A. Test metrics

In order to evaluate and compare different test methods, it is important to quantitatively measure the results. There exist several metrics for software tests, like the number of code errors found, number of test cases per requirement, number of successful test cases etc, that are difficult to apply in our setting where the requirements are not completely clear. Instead we will use the metric of statement and branch coverage of the ROS packages that we are interested in and we prefer tests that excercise a larger percentage of the robot code.

It is important to keep in mind that 100% statement coverage does not guarantee that the code is bug free. On

[1]Spatio-Temporal Representation and Activities for Cognitive Control in Long-Term Scenarios (STRANDS). The project is funded by the EC 7th Frameworks Programme, Cognitive Systems and Robotics, project reference 600623.

the other hand, code coverage is a metric that gives some idea about how much of the code gets touched by the tests.

Another relevant metric in the setting of ROS is performance, i.e. how much CPU and memory resources get used by certain nodes under certain scenarios. We will only use the code coverage metric in the current paper as we have no reference for evaluating the performance results in the context of the chosen case study and monitoring performance has been addressed in e.g. [17] previously.

For computing code coverage of Python modules we use the tool `coverage.py` [4]. The approach is programming language agnostic, for example, `llvm-cov` or `gcov` can be used for measuring C++ code coverage in ROS with little additional effort.

B. *Model-based testing*

Model-based testing is testing on a model that describes how the system is required to behave. The model, built in a suitable machine interpretable formalism, can be used to automatically generate the test cases, either offline or online, and can also be used as the oracle that checks if the IUT passes the tests. Offline test generation means that tests are generated before test execution and executed when needed. In the case of online test generation the model is executed in lock step with the IUT. The communication between the model and the IUT involves controllable inputs of the IUT and observable outputs of the IUT. For example, we can tell the robot to go to a node called Station, and we can observe if and when the robot achieves the goal.

There are multiple different formalisms used for building models of the requirements. Our choice is Uppaal timed automata (TA) [5] because the formalism naturally supports state transitions and time and there exists the Uppaal Tron [11] tool that supports online model-based testing.

II. Related work

A. *Testing in ROS*

Testing is required in the ROS quality assurance process[2], meaning that a package needs to have tests in order to comply with the ROS package quality requirements. However, the requirement is compulsory only for centrally maintained ROS packages and it is up to the particular maintainers to choose their quality assurance process.

The ROS infrastructure supports different levels of testing. The basic testing methodology used in ROS is unit testing. The most used testing tools in ROS unit testing are *gtest* (Google C++ testing framework), *unittest* (Python unit testing framework) and *nosetest* (a more user-friendly Python unit testing framework, which extends the unittest framework). Using the aforementioned tools is not a strict requirement as the tools are agnostic to which testing framework is used. The only requirement is that the used testing framework outputs the test results in a suitable XML format (Ant JUnit XML report format).

ROS has support for higher level integration tests as well. Integration testing can be done using the *rostest* package, which is a wrapper for the `roslaunch` tool. Rostest allows specifying complex configurations of tests, which enables integration testing of different packages.

[2]http://wiki.ros.org/QAProcess

The ROS build tool (`catkin_make`) has built-in support for testing and it is fairly simple to include tests for the package. The main concern, however, is with creating the tests as it is up to the developer to write the tests for their packages. This can be difficult, because many robotics packages deal with dynamic data (e.g. object detection from image stream) and testing with dynamic data is more challenging than unit testing simple functions. For this reason, many developers neglect creating unit tests.

To our knowledge there is relatively little research published on testing robot software in ROS, but we think it deserves further attention, since it is not trivial to apply the testing techniques known in the field of software engineering to robot software. Bihlmaier and Wörn [6] introduced RUT (Robot Unit Testing) methodology to bring modern testing techniques to robotics. They outline the process of testing robots utilizing a simulation environment (e.g. Gazebo or MORSE) and control software to test robot performance and correctness of the control algorithm without actually running the tests on real hardware. Our approach follows theirs, but we have firstly introduced quantified measurement of Python code in the context of ROS and secondly embedded online model-based testing into the ROS framework, that enables not only to drive the system through scenarios deemed interesting by the developers, but also checks if the behaviour conforms to the models, i.e. formalised requirements.

Robotic environments entail uncertainty, and testing in the presence of uncertainty is a hard problem, especially automatically deciding whether a test succeeded or failed. There is an attempt to address the issue in [8] where some ideas in the future handling of uncertainty in testing are outlined. The main emphasis is on using probabilistic models to specify input distributions and to accommodate environment uncertainty in the models. We take a different approach to accommodating uncertainty by abstracting behaviour and measuring whether goals are reached within reasonable time limits.

B. *Robot monitoring and fault detection*

Several fault detection and monitoring approaches in conjunction with robotic frameworks have been proposed, e.g. [10], [12], [17], that enable to detect various faults in robot software. These complement our approach, as we introduce monitoring conformance to certain aspects of specifications that we have encoded into our model, e.g. that the robot makes reasonable progress from topological location to another connected topological location. Our approach differs from the above in the sense that in addition to monitoring, we also provide control inputs to the system. In fact, we get the continuous patrolling feature for free, as we generate the model from the topological map.

III. Modelling robot requirements with timed automata

The overall test setup used in the context of model-based testing with Uppaal Tron as the test engine and dTron as the adapter generation framework is given in Fig. 1. The model contains the formalisation of the requirements of the IUT and the environment. We model the topological map of the environment and encode distances as deadlines. The adapter is responsible for translating messages from the model to postings to appropriate topics in ROS, and vice versa. The dTron layer allows the adapter to be distributed across multiple computers while ensuring that measuring the time stays valid.

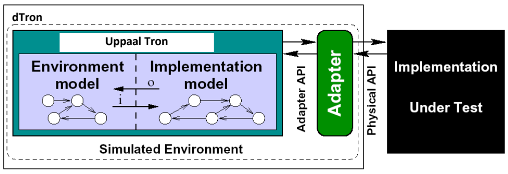


Fig. 1. The test setup involving the Uppaal Tron test engine and the distributed adapter library dTron.

The test configuration used in the current work consists of test execution environment dTron and one or many test adapters that transform abstract input/output symbols of the model to input/output data of the robot. The setup is outlined in Fig. 1. Uppaal Tron is used as a primary test execution engine. Uppaal Tron simulates interactions between the IUT and its environment by having two model components – the *environment* and the *implementation* model. The interactions between these component models are monitored during model execution. When the environment model initiates an input action i Tron triggers input data generation in the adapter and the actual test data is written to the robot interface. In response to that, the robot software produces output data that is transformed back to model output o. Thereafter, the equivalence between the output returned and the output o specified in the model is checked. The run continues if there is no conformance violation, i.e. there exists an enabled transition in the model with parameters equivalent to those passed by the robot. In addition to input/output conformance, the $rtioco_e$ checking supported by Uppaal Tron also checks for timing conformance. We refer the reader to [5] for the details.

A. Uppaal Timed Automata

Uppaal Timed Automata [5] (TA) used for the specification of the requirements are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into a single system by the parallel composition known from the process algebra CCS. An example of a system of two automata comprised of 3 locations and 2 transitions each is given in Fig. 2.

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by an additional circle inside the location.

Synchronous communication between the processes is by hand-shake synchronisation links that are called *channels*. A channel relates a pair of transitions labelled with symbols for input actions denoted by e.g. `chA?` and `chB?` in Fig. 2, and output actions denoted by `chA!` and `chB!`, where `chA` and `chB` are the names of the channels.

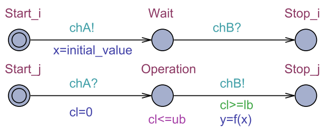


Fig. 2. A sample system with two Uppaal timed automata with synchronisation channels `chA` and `chB`. The automaton at the top denotes Process_i and the one below Process_j. In addition to the automata, the model also includes the declarations of channels `chA` and `chB`, integer constants `lb=1`, `ub=3`, and `initial_value=0`, integer variables `x` and `y`, a clock `cl`, and a function `f(x)` defined in a subset of the C language.

In Fig. 2, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronised transitions. Process_i, initially at location Start_i, initiates the call by executing the send action `chA!` that is synchronised with the receive action `chA?` in Process_j, that is initially at location Start_j. The location Operation denotes the situation where Process_j computes the output `y`. Once done, the control is returned to Process_i by the action `chB!`

The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound ub is given by the *invariant* `cl<=ub`, and the lower bound lb by the *guard condition* `cl>=lb` of the transition Operation $\rightarrow$ Stop_j. The *assignment* `cl=0` on the transition Start_j $\rightarrow$ Operation ensures that the clock cl is reset when the control reaches the location Operation. The global variables `x` and `y` model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function `f(x)` defined in the Uppaal model.

Please note that in the general case these inputs and outputs are between the processes of the model. The inputs and outputs of the test system use channels labelled in a special way described later. Asynchronous communication between processes is modelled using global variables accessible to all processes.

Formally the Uppaal timed automata are defined as follows. Let Σ denote a finite alphabet of actions $a, b, \ldots$ and C a finite set of real-valued variables p, q, r, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $c \in C, \sim \in \{\geq, \leq, =, >, <\}$ and $n \in \mathbb{N}^+$. We use $G(C)$ to denote the set of clock guards. A timed automaton A is a tuple $\langle N, l_0, E, I \rangle$ where N is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n, n \in \mathbb{N}^+$. Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to clock conditions, the propositions on integer variables k are of the form $k \sim n$ for $n \in \mathbb{N}$, and $\sim \in \{\leq, \geq, =, >, <\}$. For the formal definition of Uppaal TA full semantics we refer the reader to [5].

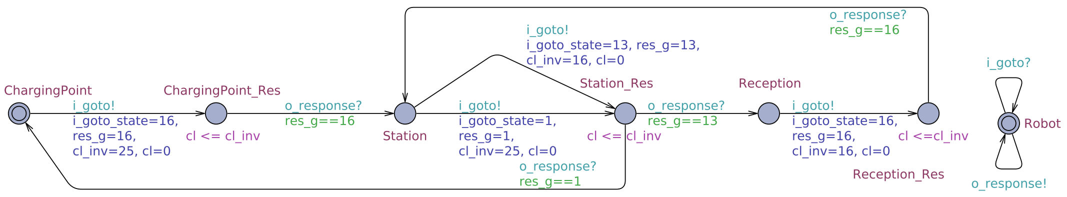


Fig. 3. Automatically generated timed automaton representation of the topological map containing locations “ChargingPoint”, “Station” and “Reception”.

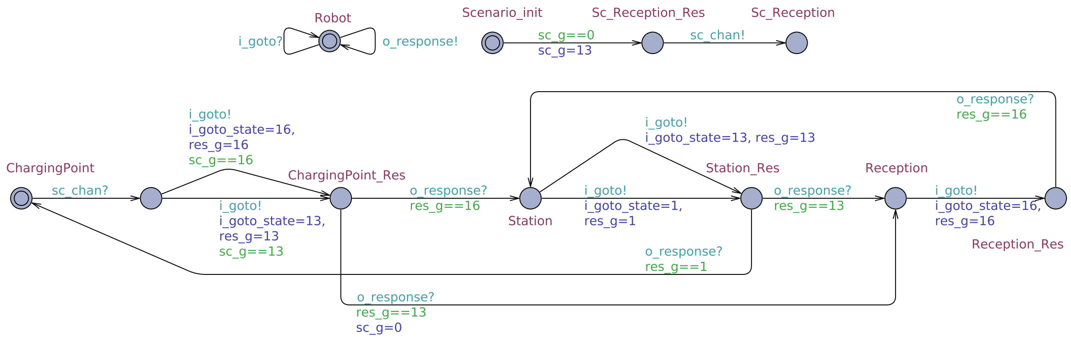


Fig. 4. Automatically generated timed automaton denoting the topological map (below) and the desired scenario (above).

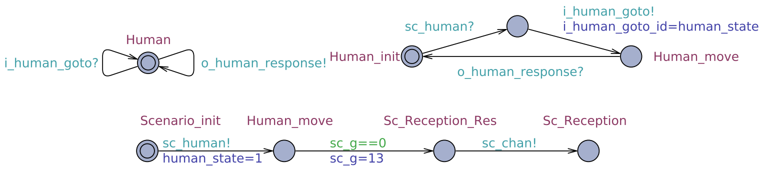


Fig. 5. A scenario involving models of humans

B. *Modelling the topological map*

One of the general requirements of a mobile robot is that it should be able to move around in its environment. We relate the requirement to the topological map and state that *the robot should be able to move to the nodes of the topological map*. The details of how the topological map gets constructed for the particular case study are given in [9], but for the purpose of the current requirements, we assume that each node of the topological map should be reachable by the robot and that from each node, it should be possible to reach adjacent nodes without having to visit any further nodes.

Since the topological map is an artefact frequently present in mobile robots, we chose to automate the translation of the topological map to the TA representation.

In Fig. 3 there is an example of a TA model of the environment of a robot specifying where the robot should be able to move and in what time the robot should be able to complete the moves. The environment stipulates that when the robot moves from the node called ChargingPoint to Station on the topological map, it synchronises on the communication channel `i_goto` with the robot model. This corresponds to passing the command to the robot to go to the state number 16, which denotes the Station node on the topological map. The destination node number is assigned on the transition to a parameter `i_goto_state` that is passed along with the synchronisation command to the test adapter which in turn will pass the command to move to the Station node on to the robot. There is an additional assignment on the transition, `res_g=16` which denotes that the current goal is node 16, i.e. the Station node. The assignment `cl_inv=25` means that the maximum time allowed for the robot to be on its way from ChargingPoint to Station is 25 time units. The clock `cl` is then reset to 0. The automaton transitions to the intermediate location ChargingPoint_Res where it awaits reaching the Station node. When Station is reached, the robot will indicate it to the test adapter which converts the indication to the response `o_response` that is passed back to the model. The guard `res_g==16` only allows the transition to be taken for the goal node 16. While on the second transition the guard does not influence the behaviour as there is no other value `res_g` can have taken, there is a choice on transitions starting from the Station node. It is possible either to go back to the ChargingPoint node when taking the transition below with the assignment `res_g=1` or go to the Reception node by taking the transition above with the assignment `res_g=13`. Then, the guards `res_g==1` and `res_g==13` will restrict which will be the valid transition after the robot has reached the goal. In this way the model will be able to distinguish which node it started off to. If the robot for some reason wonders to a wrong node or is kidnapped on the way without covering the sensors, it will be detected as a conformance failure. Also, if the robot takes too much time to reach another node, the model will trigger an error. The time restrictions are enforced by invariants `cl<=cl_inv` at the intermediate states. The robot is modelled as an automaton with a single location and the edges synchronising on the IUT input and output messages – those denoted by the `i_...` and `o_...` channels. The model of the robot is input enabled, i.e. it does not restrict any behaviour, it is up to the implementation – the robot software.

In Fig. 4 there is a model where the behaviour is restricted with a *scenario*. The model contains an automaton corresponding to the topological map, an automaton corresponding to a scenario above it, and an input enabled single location automaton denoting the robot. Now there is one additional intermediate state to facilitate synchronisation with the scenario over the `sc_chan` channel. It is important to note that when synchronisation over the `sc_chan` takes place, an integer variable `sc_g` is assigned the value 13. After such synchronisation, when the map model is at ChargingPoint_Res location, the only option to proceed is to the Reception location. In this way a goal is set that is not an immediate neighbour of the current node on the map.

If multiple such combinations of pairs are enabled, one is chosen either randomly or according to some test coverage criterion involving model elements. We have omitted the clock resets and invariants from Fig. 4 for brevity. The automaton with a single location denotes the model of the robot.

In our approach both models are automatically generated from the topological map file in the yaml format. The time delays allowed to transition from a node to node on the topological map are computed based on the distances along the edges between nodes and are computed with a margin to accommodate time spent on turning. The resulting model is able to detect situations when the robot gets stuck for example when moving too close to a low wall in simulation or when there is a link on the topological map that is not present in the environment. In the case of scenarios we compute the length of the shortest path between each pair of nodes specified in the scenario and add appropriate time restrictions to the invariant of the intermediate location introduced between the nodes on the topological map automaton stipulated by the scenario.

In Fig. 5 we add the human factor to the environment. As the simulator, Morse, supports models of humans, we can create scenarios where humans are either moved around or “teleported” to desired locations. We leave the actual locations to be specified at the adapter level and specify in the scenario automaton when which configuration of humans should be present in certain rooms. This way we can change the environment in chosen rooms and emulate varying patterns involving humans.

We can think of the scenarios as *high level test cases* in the context of integration testing. In the example above there is a test case for moving along a predefined adjacent set of nodes, one for moving to a remote location on the topological map, and one moving throught rooms with humans present.

In addition to actual testing, such scenarios run in cycles can be used for generating data, e.g. representing long term behaviour in simulation. As we can leave certain parts of the scenario less strictly specified, the model-based testing tool will vary the scenarios randomly.

Once a test failure is encountered, the search for the causes is currently left to the user. If the problem is repeatable after a certain patch set, the problem obviously may be related to the patch set, but it may also be the problem with the simulator or wrong estimates of deadlines in the model. The process of getting the requirements related to the model right requires work, that can be considered the overhead in our approach.

IV. Test execution

In order to connect the model to the robot we need an *adapter* (sometimes called *harness*) that connects the abstract messages from the model to the concrete messages comprehensible by the robot and vice versa. We use the dTron tool to fascilitate connecting our ROS adapter to Uppaal Tron. We refer to the dTron setup as the *tester*.

A. dTron

dTron[3] [3] extends the functionality of Uppaal Tron by enabling distributed and coordinated execution of tests across a computer network. It relies on Network Time Protocol (NTP) based clock corrections to give a global timestamp (t_1) to events arriving at the IUT adapter. These events are then globally serialized and published to other subscribers using the Spread toolkit [2]. Subscribers can be other IUT adapters, as well as dTron instances. Subscribers that have clocks synchronised with NTP also timestamp the event received message (t_2) to compute and if necessary and possible, compensate for the messaging overhead $D = t_2 - t_1$. The parameter D is essential in real-time executions to compensate for messaging delays in test verdict that may otherwise lead to false-negative non-conformance results for the test-runs.

B. ROS test adapter

We have created a test adapter that can be used as a template for test adapters in any similar dTron-based MBT setup. The template adapter is implemented in C++ and the specific case study inherits from the template adapter and implements the one specific to the case study. The template adapter has the implementations for receiving and sending messages between the tester and the adapter – a generic prerequisite for performing model-based testing with dTron.

In essence, the test adapter specifies what is to be done when a synchronisation message is received from the tester. In the mobile robot case, the model specifies the goal waypoint where the robot must travel. Upon receiving the waypoint information via a synchronisation message (`i_...` channel), it is either passed as a goal to the standard ROS `move_base` action server or the action server responsible for topological navigation – `topological_navigation`, depending on which level we choose to run the tests. In the implementation the goal topic is specified in the configuration of the adapter.

[3]http://www.cs.ttu.ee/dtron

After publishing the goal, the adapter waits for the action server to return a result for the action (i.e. whether it was successful or not). In case the action was successful, the adapter sends a synchronisation message (`o_...` channel) back to the tester and waits for a new goal to be passed on to the action server. If the goal was not achieved, the adapter does not send a synchronisation to the tester and the tester will detect a time-out and report the test failure.

V. Experimental results

We specified different scenarios, i.e. high level test cases, and ran the tests in two different simulated environments[4] and repeated same scenarios by sending the goals to the `move_base` and `topological_navigation` action servers. The results are summarised in Table I and the highest coverage results for the particular test case for each package are highlighted in cases they can be distinguished. The "Total" columns represent total number of statements of Python code in the package, "Missed" columns represent the number of statements missed, and the "%" columns represent a combined statement and branch coverage percentage. That is why there are same statement counts but different percentages in the results of e.g. the `localisation` node.

Initially we tested the code coverage by manually specifying a neigbouring node on the topological map to `move_base`. Then we proceeded giving the same topological node to the robot as a goal via the `topological_navigation` action server. These are the rows marked by *manual goal*.

Then we repeated the same neighbouring node goals, but controlled the robot from the model (rows marked by *model*). It is expected that the code coverage is very close in these cases.

Next we specified a scenario with the goal node not being a neighbour. It can be seen from the results that significantly more code was exercised in the topological navigation node in the case of scenarios with goals passed to the `topological_navigation` action server.

Next we tested the scenario when human models are moved to different locations in a room and robot is given tasks to enter and leave the room. We managed to run the tests when passing goals to `move_base`, but the `topological_navigation` case failed because the robot got stuck with "DWA planner failed to produce path" error from `move_base`. We cannot confirm the problem to be a code error, as it can also be related to a local simulator configuration. But we have successfully demonstrated that it is possible to produce code coverage results in cases where the tests succeed and point out scenarios where the goals are not reached.

We also augmented the topological map with a transition through a wall. The test system yielded a test failure based on exceeding the deadline for reaching the node.

We used different versions of code in C1 and C2, that is why there is are slightly different statement counts in similar components. C1 experiments were done on STRANDS packages taken from the GIT repository while C2 experiments were done using current versions of packages available from the Ubuntu Strands deb package repository. In the case of the `actionlib` package, it appears that more code is utilized in the case of topological navigation. The `strands_navigation` package contains only messages in Python, thus there are very little differences in coverage. In the `topological_navigation` module utilisation there is clear correlation between the use of topological goals and code coverage. The `localisation` node uses practically the same amount of code regardless of the scenario. In the case of the `navigation` node the difference is the largest and there is clear correlation between larger code coverage and harder navigation tasks[5].

When interpreting the results it is important to keep in mind that the coverage numbers are for *single high level test cases*. When analysing e.g. the `navigation` package coverage, then the code covered in the `move_base` case is a subset of the coverage in the `topological_navigation` case. Developing a test suite with a higher total code coverage is an iterative process of running the tests and looking at what code has still been missed. In the current case study, the test suite needs to be extended with a scenario with a goal that is the same as the current node, there need to be scenarios triggering the preemting of goals, and triggering several different exceptions. Such behaviour requires extending the model, and perhaps the adapter. While the coverage numbers of the reported scenarios are below 100%, the added value provided by the current approach lies in clear feedback, either in the form of test failure, or in the case of success, what code was used in the particular set of scenarios and what was missed.

VI. Conclusion

We presented a case study of applying model-based testing in ROS and evaluating the results in terms of code coverage of code related to topological navigation of mobile robots. Relying on the empirical evidence, we conclude that the proposed automatic generation of models from topological maps and defining scenarios as sequences of states provides a valuable tool of exercising the system with the purpose to achieve high code coverage. By performing the tests on the `move_base` coordinate level and topological navigation level, we showed that our approach can also be used to validate and discover problems in configurations, such as topological maps. We also showed how to build models of the environment involving human models in simulation. Similar scenarios can be carried out also in real life, but then the test adapter needs to be changed to give humans instructions when and where to move to, and when humans are in place, the adapter can return to giving the robot next goals.

The future work on the model and adapter side involves extending the dynamic reconfiguration of the environment, e.g. connecting collision detection probes in Morse with the test adapter and introducing natural human movement. Improving the code coverage requires insight into the packages and manual extension of model and the adapter to support triggering various exceptions and other specific actions.

[4]C1 corresponds to AAF simulation environment and C2 to Bham SoCS building ground floor simulation environment.

[5]The code and detailed coverage statistics is available at http://cs.ttu.ee/staff/juhan/mobile_robot_mbt/.

TABLE I. THE EXPERIMENTAL RESULTS OF MODEL-BASED TESTING A MOBILE ROBOT IN SIMULATION IN TWO DIFFERENT VIRTUAL ENVIRONMENTS, C1 AND C2.

	actionlib			strands_navigation			topological_navigation			localisation node			navigation node		
	Total	Missed	%	Total	Missed	%	Total	Missed	%	Total	Missed	%	Total	Missed	%
C1 manual goal move_base	1347	733	46	13024	10969	16	1954	1502	23	154	37	72	349	251	24
C2 manual goal move_base	1347	872	35	13024	10902	16	1789	1479	17	154	37	73	344	247	24
C1 model goal move_base	1347	733	46	13024	10969	16	1954	1502	23	154	37	73	349	251	24
C2 model goal move_base	1347	872	35	13024	10902	16	1789	1479	17	154	37	73	344	247	24
C1 manual goal topo-nav	1347	565	58	13024	10832	17	1954	1335	32	154	37	72	349	99	64
C2 manual goal topo-nav	1347	678	50	13024	10864	17	1789	1346	25	154	37	73	344	159	48
C1 model goal topo-nav	1347	598	56	13024	10832	17	1954	1335	32	154	37	73	349	97	65
C2 model goal topo-nav	1347	615	54	13024	10749	17	1789	1311	27	154	37	73	344	98	64
C1 scenario topo-nav	1347	598	56	13024	10832	17	1954	1315	33	154	37	73	349	77	72
C2 scenario move_base	1347	872	35	13024	10902	16	1789	1475	18	154	37	73	344	247	24
C2 scenario topo-nav	1347	613	54	13024	10749	17	1789	1286	28	154	37	73	344	73	73
C2 scenario with humans move_base	1347	871	35	13024	10902	16	1789	1479	17	154	37	73	344	247	24

Acknowledgements

We would like to thank Nick Hawes, Marc Hanheide and Jaime Pulido Fentanes for useful discussions and support in setting up a local copy of the STRANDS simulation environment. We also thank the anonymous referees for their constructive comments. This work was partially supported by the European Union through the European Regional Development Fund via the competence centre ELIKO.

REFERENCES

[1] Rares Ambrus, Nils Bore, John Folkesson, and Patric Jensfelt. Meta-rooms: Building and maintaining long term spatial models in a dynamic world. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 1854–1861, 2014.

[2] Yair Amir, Michal Miskin-Amir, Jonathan Stanton, and John Schultz et al. The Spread toolkit, 2015. http://spread.org/, accessed in May 2015.

[3] Aivo Anier and Jüri Vain. Model based continual planning and control for assistive robots. In Emmanuel Conchon, Carlos Manuel B. A. Correia, Ana L. N. Fred, and Hugo Gamboa, editors, *HEALTHINF 2012 - Proceedings of the International Conference on Health Informatics, Vilamoura, Algarve, Portugal, 1 - 4 February, 2012.*, pages 382–385. SciTePress, 2012.

[4] Ned Batchelder and Gareth Rees. Coverage.py – code coverage testing for Python, 2015. http://nedbatchelder.com/code/coverage/, accessed in April 2015.

[5] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[6] Andreas Bihlmaier and Heinz Wörn. Robot unit testing. In Davide Brugali, Jan F. Broenink, Torsten Kroeger, and Bruce A. MacDonald, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 8810 of *Lecture Notes in Computer Science*, pages 255–266. Springer International Publishing, 2014.

[7] Gilberto Echeverria, Séverin Lemaignan, Arnaud Degroote, Simon Lacroix, Michael Karg, Pierrick Koch, Charles Lesire, and Serge Stinckwich. Simulating complex robotic scenarios with MORSE. In *SIMPAR*, pages 197–208, 2012.

[8] Sebastian G. Elbaum and David S. Rosenblum. Known unknowns: testing in the presence of uncertainty. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 833–836. ACM, 2014.

[9] Jaime Pulido Fentanes, Bruno Lacerda, Tomás Krajník, Nick Hawes, and Marc Hanheide. Now or later? Predicting and maximising success of navigation actions from long-term experience. In *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, pages 1112–1117, 2015.

[10] Raphael Golombek, Sebastian Wrede, Marc Hanheide, and Martin Heckmann. Online data-driven fault detection for robotic systems. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, pages 3011–3016. IEEE, 2011.

[11] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[12] Hengle Jiang, Sebastian G. Elbaum, and Carrick Detweiler. Reducing failure rates of robotic systems though inferred invariants monitoring. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*, pages 1899–1906. IEEE, 2013.

[13] Kohsuke Kawaguchi, Andrew Bayer, and R.Tyler Croy. Jenkins – an extensible open source continuous integration server, 2015. http://jenkins-ci.org, accessed in April 2015.

[14] Tomás Krajník, Jaime Pulido Fentanes, Óscar Martínez Mozos, Tom Duckett, Johan Ekekrantz, and Marc Hanheide. Long-term topological localisation for service robots in dynamic environments using spectral maps. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 4537–4542, 2014.

[15] Lars Kunze, Chris Burbridge, Marina Alberti, Akshaya Thippur, John Folkesson, Patric Jensfelt, and Nick Hawes. Combining top-down spatial reasoning and bottom-up object class recognition for scene understanding. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14-18, 2014*, pages 2910–2915, 2014.

[16] Lars Kunze, Keerthi Kumar Doreswamy, and Nick Hawes. Using qualitative spatial relations for indirect object search. In *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*, pages 163–168. IEEE, 2014.

[17] Valiallah Monajjemi, Jens Wawerla, and Richard T. Vaughan. Drums: A middleware-aware distributed robot monitoring system. In *Canadian Conference on Computer and Robot Vision, CRV 2014, Montreal, QC, Canada, May 6-9, 2014*, pages 211–218. IEEE Computer Society, 2014.

[18] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[19] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach.* Morgan Kaufmann, 2007.

Appendix 7

Publication IV
J. Vain, E. Halling, G. Kanter, A. Anier, and D. Pal. Model-based testing of real-time distributed systems. In G. Arnicans, V. Arnicane, J. Borzovs, and L. Niedrite, editors, *Databases and Information Systems - 12th International Baltic Conference*, *DB&IS 2016*, *Riga*, *Latvia*, *July 4-6*, *2016*, *Proceedings*, volume 615 of *Communications in Computer and Information Science*, pages 272–286. Springer, 2016

Databases and Information Systems IX
G. Arnicans et al. (Eds.)
© 2016 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/978-1-61499-714-6-297

Automatic Distribution of Local Testers for Testing Distributed Systems

Jüri VAIN [1], Evelin HALLING, Gert KANTER, Aivo ANIER and Deepak PAL
Department of Computer Science, Tallinn University of Technology, Estonia
http://cs.ttu.ee/

Abstract. *Low-latency systems* where reaction time is primary success factor and design consideration, are serious challenge to existing integration and system level testing techniques. Modern cyber physical systems have grown to the scale of global geographic distribution and latency requirements are measured in nanoseconds. While existing tools support prescribed input profiles they seldom provide enough reactivity to run the tests with simultaneous and interdependent input profiles at remote front ends. Additional complexities emerge due to severe timing constraints the tests have to meet when test navigation decision time ranges near the message propagation time. Sufficient timing conditions for remote online testing have been proposed in remote Δ-testing method recently. We extend the Δ-testing by deploying testers on fully distributed test architecture. This approach reduces the test reaction time by almost a factor of two. We validate the method on a distributed oil pumping SCADA system case study.

Keywords. model-based testing, distributed systems, low-latency systems

1. Introduction

Modern large scale cyber-physical systems have grown to the size of global geographic distribution and their latency requirements are measured in microseconds or even nanoseconds. Such applications where latency is one of the primary design considerations are called *low-latency systems* and where it is of critical importance – to *time critical systems*. A typical example of distributed time critical system is smart energy grid (SEG) where delayed control signals can cause overloads and blackouts of whole regions. Thus, the proper timing is the main measure of success in SEG and often the hardest design concern.

Since large SEG-s systems are mostly distributed systems (by distributed systems we mean the systems where computations are performed on multiple networked computers that communicate and coordinate their actions by passing messages), their latency dynamics is influenced by many technical and non-technical factors. Just to name a few, energy consumption profile look up time (few milliseconds) may depend on the load profile, messaging middleware and the networking stacks of operating systems. Similarly, due to cache miss, the caching time can grow from microseconds to about hundred

[1]Corresponding Author: Jüri Vain; Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, 19086 Tallinn, Estonia; E-mail: juri.vain@ttu.ee

milliseconds [1]. Reaching sufficient feature coverage by integration testing of such systems in the presence of numerous latency factors and their interdependences, is out of the reach of manual testing. Obvious implication is that scalable integration and system level testing presumes complex tools and techniques to assure the quality of the test results [2]. To achieve the confidence and trustability, the test suites need to be either correct by construction or verified against the test goals after they are generated. The need for automated test generation and their correctness assurance have given raise to model based testing (MBT) and the development of several commercial and academic MBT tools. In this paper, we interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviors described by the system requirements model with the observed behaviors of an actual implementation (implementation under test). For detailed overview of MBT and related tools we refer to [3] and [4].

2. Related Work

Testing distributed systems has been one of the MBT challenges since the beginning of the 90s. An attempt to standardize the test interfaces for distributed testing was made in ISO OSI Conformance Testing Methodology [5]. A general distributed test architecture, containing distributed interfaces, has been presented in Open Distributed Processing (ODP) Basic Reference Model (BRM), which is a generalized version of ISO distributed test architecture. First MBT approaches represented the test configurations as systems that can be modeled by finite state machines (FSM) with several distributed interfaces, called ports. An example of abstract distributed test architecture is proposed in [6]. This architecture suggests the Implementation Under Test (IUT) contains several ports that can be located physically far from each other. The testers are located in these nodes that have direct access to ports. There are also two strongly limiting assumptions: (i) the testers cannot communicate and synchronize with one another unless they communicate through the IUT, and (ii) no global clock is available. Under these assumptions a test generation method was developed in [6] for generating synchronizable test sequences of multi-port finite state machines. However, it was shown in [7] that no method that is based on the concept of synchronizable test sequences can ensure full fault coverage for all the testers. The reason is that for certain testers, given a FSM transition, there may not exist any synchronizable test sequence that can force the machine to traverse this transition. This is generally known as *controllability* and *observability* problem of distributed testers. These problems occur if a tester cannot determine either when to apply a particular input to IUT, or whether a particular output from IUT is generated in response to a specific input [8]. For instance, the controllability problem occurs when the tester at a port p_i is expected to send an input to IUT after IUT has responded to an input from the tester at some other port p_j, without sending an output to p_i. The tester at p_i is unable to decide whether IUT has received that input and so cannot know when to send its input. Similarly, the observability problem occurs when the tester at some port p_i is expected to receive an output from IUT in response to a given input at some port other than p_i and is unable to determine when to start and stop waiting. Such observability problems can introduce fault masking.

In [8], it is proposed to construct test sequences that cause no controllability and observability problems during their application. Unfortunately, offline generation of test sequences is not always applicable. For instance, when the model of IUT is non-deterministic it needs instead of fixed test sequences online testers capable of handling non-deterministic behavior of IUT. But even this is not always possible. An alternative is to construct testers that includes external coordination messages. However, that creates communication overhead and possibly the delay introduced by the sending of each message. Finding an acceptable amount of coordination messages depends on timing constraints and finally amounts to finding a tradeoff between the controllability, observability and the cost of sending external coordination messages.

The need for retaining the timing and latency properties of testers became crucial natively when time critical cyber physical and low-latency systems were tested. Pioneering theoretical results have been published on test timing correctness in [9] where a remote abstract tester was proposed for testing distributed systems in a centralized manner. It was proven that if IUT ports are remotely observable and controllable then 2Δ-condition is sufficient for satisfying timing correctness of the test. Here, Δ denotes an upper bound of message propagation delay between tester and IUT ports. However, this condition makes remote testing problematic when 2Δ is close to timing constraints of IUT, e.g. the length of time interval when the test input has to reach port has definite effect on IUT. If the actual time interval between receiving an IUT output and sending subsequent test stimulus is longer than 2Δ the input may not reach the input port in time and the test goal cannot be reached.

In this paper we focus on distributed online testing of low latency and time-critical systems with distributed testers that can exchange synchronization messages that meet Δ-delay condition. In contrast to the centralized testing approach, our approach reduces the tester reaction time from 2Δ to Δ. The validation of proposed approach is demonstrated on a distributed oil pumping SCADA system case study.

3. Preliminaries

3.1. Model-Based Testing

In model-based testing, the formal requirements model of implementation under test describes how the system under test is required to behave. The model, built in a suitable machine interpretable formalism, can be used to automatically generate the test cases, either offline or online, and can also be used as the oracle that checks if the IUT behavior conforms to this model. Offline test generation means that tests are generated before test execution and executed when needed. In the case of online test generation the model is executed in lock step with the IUT. The communication between the model and the IUT involves controllable inputs of the IUT and observable outputs of the IUT.

There are multiple different formalisms used for building conformance testing models. Our choice is Uppaal timed automata (TA) [10] because the formalism is designed to express the timed behavior of state transition systems and there exists a family of tools that support model construction, verification and online model-based testing [11].

3.2. Uppaal Timed Automata

Uppaal Timed Automata [10] (UTA) used for the specification of the requirements are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into a single system by the parallel composition known from the process algebra CCS. An example of a system of two automata comprised of 3 locations and 2 transitions each is given in Figure 1.

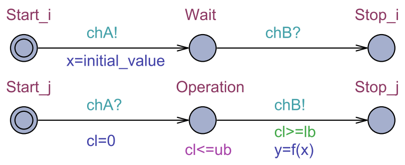


Figure 1. A parallel composition of Uppaal timed automata

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by an additional circle inside the location.

Synchronous communication between the processes is by hand-shake synchronization links that are called *channels*. A channel relates a pair of edges labeled with symbols for input actions denoted by e.g. `chA?` and `chB?` in Figure 1, and output actions denoted by `chA!` and `chB!`, where `chA` and `chB` are the names of the channels.

In Figure 1, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronized transitions. Process_i, initially at location Start_i, initiates the call by executing the send action `chA!` that is synchronized with the receive action `chA?` in Process_j, that is initially at location Start_j. The location Operation denotes the situation where Process_j computes the output `y`. Once done, the control is returned to Process_i by the action `chB!`

The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound *ub* is given by the *invariant* `cl<=ub`, and the lower bound *lb* by the *guard condition* `cl>=lb` of the transition Operation $\rightarrow$ Stop_j. The *assignment* `cl=0` on the transition Start_j $\rightarrow$ Operation ensures that the clock *cl* is reset when the control reaches the location Operation. The global variables `x` and `y` model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function `f(x)` defined in the declarations section of the Uppaal model.

The inputs and outputs of the test system are modeled using channels labeled in a special way described later. Asynchronous communication between processes is modeled using global variables accessible to all processes.

Formally the Uppaal timed automata are defined as follows. Let Σ denote a finite alphabet of actions $a, b, \ldots$ and C a finite set of real-valued variables p, q, r, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $p \in C, \sim \in \{\geq, \leq, =, >, <\}$ and $n \in \mathbb{N}^+$. We use $G(C)$ to denote the set of clock guards. A timed automaton A is a tuple $\langle N, l_0, E, I \rangle$ where N is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n, n \in \mathbb{N}^+$. Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to clock conditions, the propositions on integer variables k are of the form $k \sim n$ for $n \in \mathbb{N}$, and $\sim \in \{\leq, \geq, =, >, <\}$. For the formal definition of Uppaal TA full semantics we refer the reader to [12] and [10].

4. Remote Testing

The test purpose most often used in MBT is conformance testing. 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 [13] is to check if the behavior 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 behavior 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 [14]) 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 behavior of the IUT and from the coverage criteria that constrain the behavior defined in IUT model with only those addressed by the test purpose. In our approach Uppaal Timed Automata (UTA) [10] are used as a formalism for modeling IUT behavior. 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 behavior 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 [15], 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.

4.1. Centralized Remote Testing

Let us first consider a centralized tester design case. In the case of centralized tester, all test inputs are generated by a single monolithic tester. This means that the centralized tester will generate an input for the IUT, waits for the result and continues with the next set of inputs and outputs until the test scenario has been finished. Thus, the tester has to wait for the duration it takes the signal to be transmitted from the tester to the IUT's ports and the responses back from ports to the tester. In the case of IUT being distributed in a way that signal propagation time is non-negligible, this can lead into a situation where the tester is unable to generate the necessary input for the IUT in time due to message propagation latency. These timing issues can render testing an IUT impossible if the IUT is a distributed real-time system.

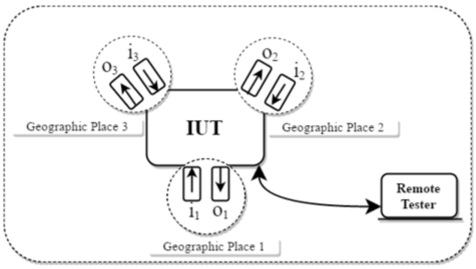


Figure 2. Remote tester communication architecture

To be more concrete, let us consider the remote testing architecture depicted in Figure 2 and the corresponding model depicted in Figure 3 and 4. In this case the IUT has 3 ports (p_1, p_2, p_3) in geographically different places to interact within the system, inputs i_1, i_2 and i_3 at ports p_1, p_2 and p_3 respectively and outputs o_1 at port p_1, o_2 at port p_2, o_3 at port p_3.

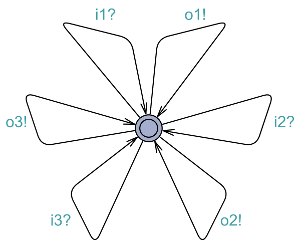


Figure 3. IUT model

We model a multi-ports timed automata by splitting the edges with multiple communication actions to a sequence of edges each labeled with exactly one action and connected via committed locations, so that all ports of such group are updated at the same time. In Figure 4 the labels on the edges represent the transitions and the transition tuple (`L0`, `L1`, $i_1!/(o_1?, o_2?)$) is represented by sequence of edges each labeled with exactly one action and connected via committed locations. For example the sequence of edges from location `L0` to `L1` with labels $i_1!$, $o_1?$ and $o_2?$ represents the multiple communication actions where the input $i_1!$ at port p_1 in location `L0` being able to trigger a transition that leads to the output $o_1?$ and $o_2?$ at ports p_1, p_2 respectively and the location becoming `L1`.

Using such splitting of edges with committed locations, we model a three port automata shown in Figure 4 where the tester sends an input i_1 to the port p_1 at `Geographic Place 1` and receives a response or outputs o_1 and o_2 from IUT at `Geographic Place 1` and `Geographic Place 2` respectively. After receiving the result, the tester is in location `L1`, it gets both i_3 on port p_3 and i_2 on port p_2. Then, either it follows the intended

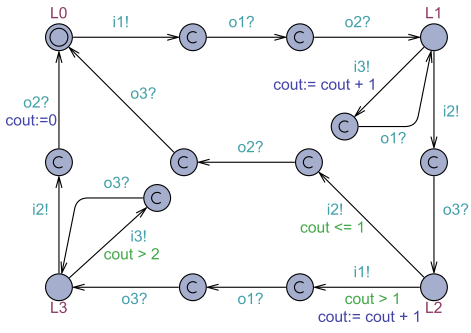


Figure 4. Remote Tester model

path sending i_3 before i_2, or it sends i_2 before i_3. If tester decides to send i_3 before i_2 it receives an output o_1 at port p_1 and returns to location `L1`. The transition is a self loop if its start and end locations are the same. If tester decides to send i_2, the IUT responds with an output o_3 at port p_3. Now, the tester is in location `L2`, it gets both i_1 on port p_1 and i_2 on port p_2. Based on guard condition and previously triggered inputs and received outputs the next input is sent to IUT and tester continues with the next set of inputs and outputs until the test scenario has been finished.

The described IUT is a real-time distributed system, which means that it has strict timing constraints for messaging between ports. More specifically, after sending the first input i_1 to port p_1 at `Geographic Place 1` and after receiving the response o_1 and o_2 at `Geographic Place 1` and `Geographic Place 2` respectively, the tester needs to decide and send the next input i_2 to port p_2 at `Geographic Place 2` or input i_3 to port p_3 at `Geographic Place 3` in Δ time. But, due to the fact that the tester is not at the same geographical place as the distributed IUT, it is unable to send the next input in time as the time it takes to receive the response and send the next input amounts to 2Δ, which is double the time allotted for the next input signal to arrive.

Consequently, the centralized remote testing approach is not suitable for testing a real-time distributed system if the system has strict timing constraints with non negligible signal propagation times between system ports. To overcome this problem, the centralized tester is decomposed and distributed as described in the next section.

5. Distributed Testing

The shortcoming of the centralized remote testing approach is mitigated with extending the Δ-testing idea by decomposing the monolithic remote tester into multiple local testers. These local testers are directly attached to the ports of the IUT. Thus, instead of bidirectional communication between a remote tester and the IUT, only unidirectional synchronization between the local testers is required. The local testers are generated in two steps: at first, a centralized remote tester is generated by applying the reactive planning online-tester synthesis method of [16], and second, a set of synchronizing local testers is derived by decomposing the monolithic tester into a set of location specific tester instances. Each tester instance needs to know now only the occurrence of i/o events

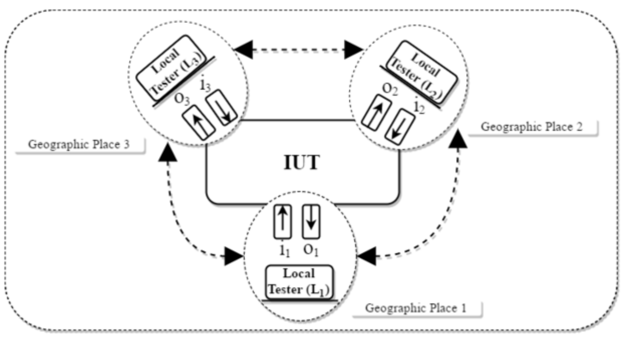


Figure 5. Distributed Local testers communication architecture

at other ports which determine its behavior. Possible reactions of the local tester to these events are already specified in its model and do not need further feedback to the event sender. The decomposing preserves the correctness of testers so that if the monolithic remote tester meets 2Δ requirement then the distributed testers meet (one) Δ-controllability requirement.

We apply the algorithm described in 5.1 to transform the centralized testing architecture depicted in Figure 2 into a set of communicating distributed local testers, the architecture of which is shown in Figure 5. After applying the algorithm, the message propagation time between the local tester and the IUT port has been eliminated because the tester is attached directly to the port. This means that the overall testing response time is also reduced, because previously the messages had to be transmitted over a channel with latency bidirectionally. The resulting architecture mitigates the timing issue by replacing the bidirectional communication with a unidirectional broadcast of the IUT output signals between the distributed local testers. The generated local tester models are shown in Figure 6, Figure 7, Figure 8 and Figure 9.

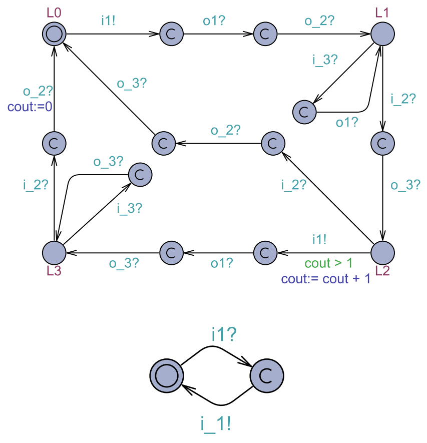


Figure 6. Local tester at Geographic Place 1

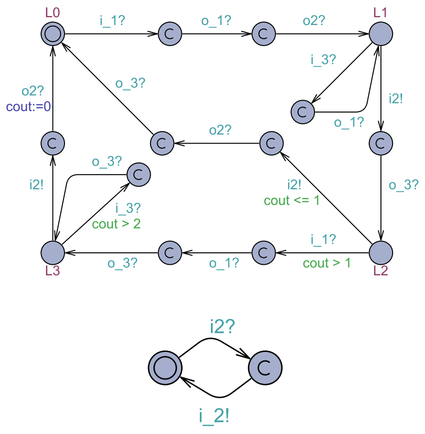


Figure 7. Local tester at Geographic Place 2

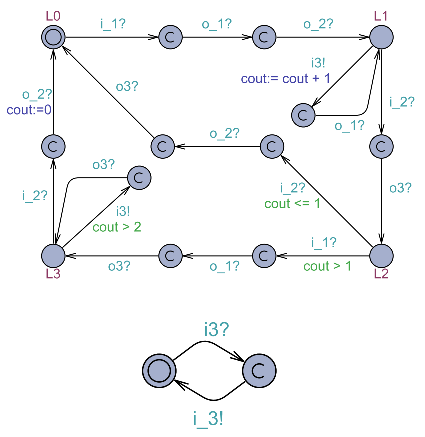


Figure 8. Local tester at Geographic Place 3

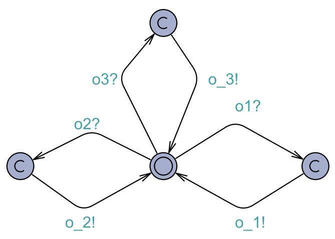


Figure 9. Output Event Synchronizer

5.1. Tester Distribution Algorithm

Let M^{MT} denote a monolithic remote tester model generated by applying the reactive planning online-tester synthesis method [16]. $Loc(IUT)$ denotes a set of geographically different port locations of IUT. The number of locations can be from 1 to n, where $n \in \mathbb{N}$ i.e. $Loc(IUT) = \{l_n | n \in \mathbb{N}\}$. Let P_{l_n} denotes a set of ports accessible in the location l_n.

1. For each l, $l \in Loc(IUT)$ we copy M^{MT} to M^l to be transformed to a location specific local tester instance.
2. For each M^l we go through all the edges in M^l. If the edge has a synchronizing channel and the channel does not belong to the the set of ports P_{l_n}, we do the following:

   - if the channel's action is *send*, we replace it with the co-action *receive*.
   - if the channel's action is *receive*, we do nothing.

3. For each M^l we add one more automaton that duplicates the input signals from M^l to IUT, attached to the set of ports P_{l_n} and broadcasts the duplicates to other local testers to synchronize the test runs at their local ports. Similarly the IUT local output event observations are broadcast to other testers for synchronization purposes like automaton in Figure 9.

6. Correctness of Tester Distribution Algorithm

To verify the correctness of distributed tester generation algorithm we check the bi-simulation equivalence relation between the model of monolithic centralized tester and that of distributed tester. For that the models are composed by parallel compositions so that one has a role of words generator on i/o alphabet and other the role of words acceptor machine. If the i/o language acceptance is established in one direction then the roles of models are reversed. Since the i/o alphabets of remote tester and distributed tester differ due to synchronizing messages of distributed tester the behaviors are compared based on the i/o alphabet observable on IUT ports only. Second adjustment of models to be made for bi-simulation analysis is the reduction of message propagation delays to uniform basis either on Δ or 2Δ in both models. Assume (due to closed world assumption used in MBT):

- the centralized remote tester model: $M^{remote} = TA^{IUT} \parallel TA^{r-TST}$
- the distributed tester model: $M^{distrib} = TA^{IUT} \parallel \shortparallel_i TA_i^{d-TST}$ $i = [1, n]$, n - number of ports locations.
- to unify the timed words $TW(M^{remote})$ and $TW(M^{distrib})$ the communication delay between IUT and Tester is assumed.

Definition (correctness of tester distribution mapping): The mapping $M^{remote} \xmapsto{Algorithm} M^{distrib}$ is correct if TA^{r-TST} and $\shortparallel_i TA_i^{d-TST}$ are observation bisimilar, i.e. if TA^{r-TST} and $\shortparallel_i TA_i^{d-TST}$ are respectively generating and accepting automata on common i/o alphabet $\Sigma^i \cup \Sigma^o$ then all timed words $TW(TA^{r-TST})$ are recognizable by $\shortparallel_i TA_i^{d-TST}$ and all timed words $TW(\shortparallel_i TA_i^{d-TST})$ are recognizable by TA^{r-TST}.

Here, alphabet $\Sigma^i \cup \Sigma^o$ includes i/o symbols used at IUT-TESTER interfaces of M^{remote} and $M^{distrib}$.

Correctness verification of the distribution mapping:

Step 1: (Constructing generating-accepting automata synchronous composition):

- label each output action of TA^{r-TST} with output symbol a! and its co-action in $\shortparallel_i TA_i^{d-TST}$ with input symbol a?;
- define parallel composition $TA^{r-TST} \parallel \shortparallel_i TA_i^{d-TST}$ with synchronous i/o actions.

Step 2: (Bisimilarity proof by model checking): TA^{r-TST} and $\shortparallel_i TA_i^{d-TST}$ are observation bisimilar if following holds: $M^{remote} \models \texttt{not deadlock} \wedge M^{distrib} \models \texttt{not deadlock} \Rightarrow TA^{r-TST} \parallel \shortparallel_j TA_j^{d-TST} \models \texttt{not deadlock}$ $j = [1, n]$, n - number of local testers , i.e. the composition of bisimilar testers must be non-blocking if the testers composed with IUT model separately are non-blocking.

7. Case Study

7.1. Use Case

The benefit of using the proposed method is demonstrated in the use case of an EMS (Energy Management System) which is integrated into the SCADA (Supervisory Control And Data Acquisition) system of an industrial consumer. An EMS is essentially a load balancing system. The target of the balancing system is the load on power supplies called feeders to an industrial consumer. These industrial power consumers have multiple feeders to power the devices required for their operations (e.g., pumps and pipeline heating systems). The motivation for balancing the power consumption between the feeders stems from the fact that the power companies can enforce fines on the industrial consumers if the power consumption exceeds certain thresholds due to safety considerations and possible damage to the equipment. Therefore, the consumer is motivated to share the power consuming devices among the feeders minimize or eliminate such energy consumption spikes completely.

Let us consider a use case in which an oil terminal has two feeders and multiple power consuming devices (consumers). The number of consumers can range from some to many. In our use case we have 32 consumers, but in other cases it can be more. These consumers are both pumps and pipeline heating systems. The pumps have a high surge power consumption when starting up which must be taken into consideration when designing an EMS. The EMS monitors the current consumption by polling the consumers via a communication system (e.g., PROFIBUS, CAN bus or Industrial Ethernet). The PROFIBUS communication system is standardized in EN 50170 international standard.

Because the oil terminal stores oil it is considered an explosion hazard area and therefore, a special communication system that is certified for explosive areas - PROFIBUS PA (Process Automation) is used. PROFIBUS PA meets the 'Intrinsically Safe' (IS) and bus-powered requirements defined by IEC 61158-2. The maximum transfer rate of PROFIBUS PA is 31.25 kbit/s which can limit the system response speed if there are many devices connected to the PROFIBUS bus and each device has significant input and output data load.

The EMS is able to switch devices from being supplied from either feeder. Ideally, the power consumption is shared equally among both feeders at all times. This means that the EMS monitors the devices and switches devices over to other feeders if the power consumption is unbalanced among the feeders. In normal operation, the feeder loads are kept sufficiently low to accommodate new devices starting up in a way that the surge consumption will not exceed the threshold power of the feeders.

The EMS polls every power consumer periodically and updates the total consumption. Based on this total consumption, the EMS will command the power distribution devices to switch around from first feeder to the second in case the load on the first feeder is higher than on the second and vice versa.

In our use case we simulate the power consumption of the devices as the input to the IUT. The tester monitors the output (the EMS feeder load values). The test purpose is to verify that neither of the power loads exceed the specified threshold. Exceeding this limit might cause equipment damage and the power company can impose fines upon violating this limit.

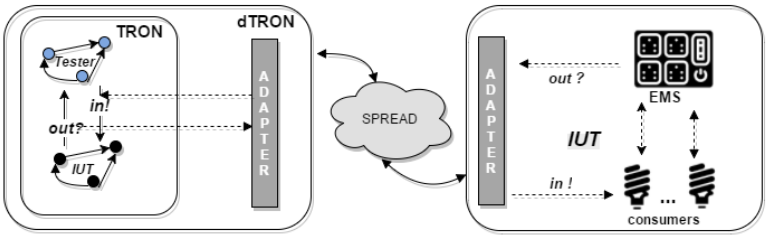


Figure 10. Case Study Test Architecture

The test architecture is depicted in Figure 10. In the right side of the figure, we can see the EMS and consumers as the implementation under test. The test model and test runner is on the left side. The test is executed via DTRON, which transmits the inputs and outputs via Spread. In the IUT and tester models we are going to introduce, the signals prefixed with $i\_$ or $o\_$ are synchronizing signals sent through Spread message serialization service . The signals without the aforementioned prefixes are internal signals which are not published to the Spread network. The input to the IUT is provided by the remote tester model is depicted in Figure 13 which simulates the device power consumption levels and creates challenging scenarios for the EMS. The EMS queries the consumers which are modeled in Figure 12 and balances the load between the feeders based on the total power consumption monitoring data . The EMS model is shown in Figure 11 which displays the querying loop. The querying is performed in a loop due to the semi-duplex nature of communication in PROFIBUS networks. The EMS also takes the maximum power limit into account as the total power consumption must not exceed this level. This can be seen in the remote tester model shown in Figure 13. Remote tester nondeterministically selects a consumer and sends the level of energy consumption for that particular device to the input port of the IUT. Then the remote tester waits s time units before requesting the current feeder energy levels. On the model, it is indicated as *i_get_line_balance*!. After receiving the current values the tester will check whether they are within allowed range. If the values exceed the limit the test verdict is fail. Otherwise the tester will continue with the next iteration.

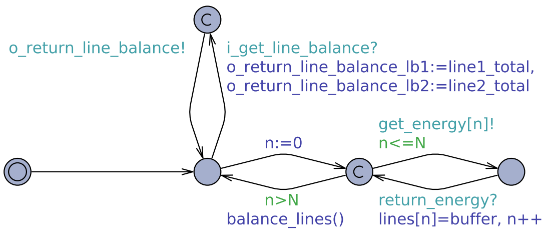


Figure 11. Energy Management System model

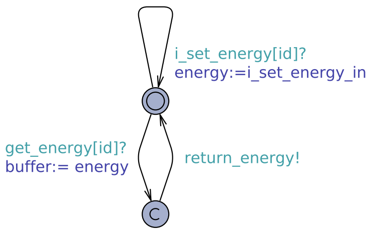


Figure 12. Consumer model

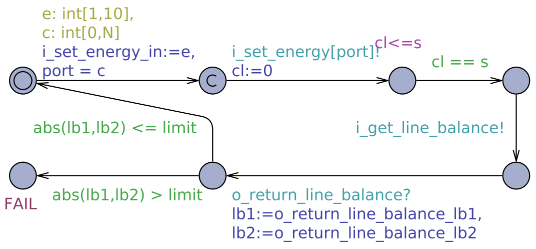


Figure 13. Remote Tester model

The communication delay between receiving the signal from EMS with the current feeder energy levels and sending input to the IUT is 2Δ. According to the specification the system must stabilize the load between feeders in stabilization time limit s after receiving the input. If Δ is very close to system stabilization time limit s indicated in the remote tester model in Figure 13 the remote tester fails to send the signal in time to the IUT.

For this reason, we introduce the distributed tester Figure 14 where each local component of the tester is closely coupled to the IUT input ports. As shown in chapter 5 this approach reduces the delay by up to Δ. This guarantees that after receiving the output from EMS we can send new input to IUT within less than s time units.

7.2. Test Execution Environment DTRON

Uppaal TRON [11] is a testing tool, based on Uppaal engine, suited for black-box conformance testing of timed systems [11]. DTRON [13] extends this enabling distributed

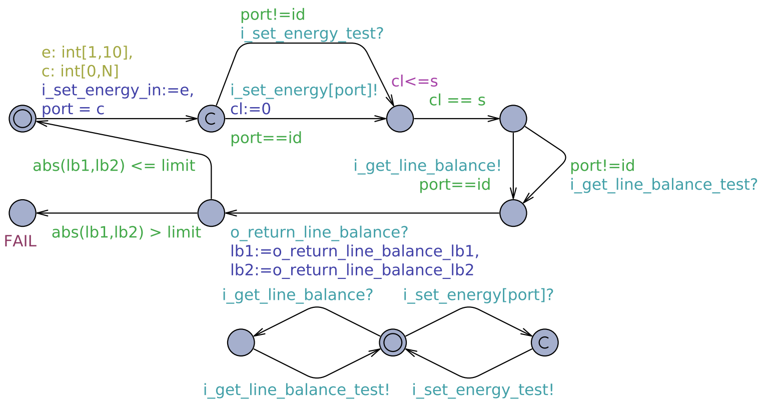


Figure 14. Parametrized local tester template for distributed testing

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 [18]. Subscribers can be other IUT 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 verification of the deployed test configuration we make corresponding time parameter adjustments in the IUT model.

8. Conclusion

We extend the Δ-testing method proposed originally for single remote tester by introducing multiple local testers on fully distributed test architecture where testers are attached directly to the ports of IUT. Thus, instead of bidirectional communication between a remote tester and IUT only unidirectional synchronization between the local testers is needed in given solution. A constructive algorithm is proposed to generate local testers in two steps: at first, a monolithic remote tester is generated by applying the reactive planning online-tester synthesis method of [16], and second, a set of synchronizing local testers is derived by partitioning the monolithic tester into a set of location specific tester instances. The partitioning preserves the correctness of testers so that if the monolithic remote tester meets 2Δ requirement then the distributed testers meet (one) Δ-controllability requirement. Second contribution of the paper is that distributed testers are generated as Uppal Timed Automata. According to our best knowledge the real time distributed testers have not been constructed automatically in this formalism yet. As for method implementation, the local testers are executed and communicating via distributed test execution environment DTRON [13]. We demonstrate that the distributed deployment architecture supported by DTRON and its message serialization service allows reducing the total test reaction time by almost a factor of two. The validation of proposed approach is demonstrated on an Energy Management System case study.

References

[1] A. Brook, Evolution and Practice: Low-latency Distributed Applications in Finance. Queue - Distributed Computing, ACM, New York (2015), vol. 13, no. 4, pp. 40-53.
[2] G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow, Applying formal software engineering techniques to smart grids. In Software Engineering for the Smart Grid (SE4SG), 2012 International Workshop, IEEE (2012), pp. 50-56.
[3] M. Utting, A. Pretschner, and B. Legeard, A taxonomy of Model-based Testing. Software Testing, Verification & Reliability, John Wiley and Sons Ltd., Chichester, UK (2012), vol. 22, iss. 5, pp. 297-312.
[4] J. Zander, I. Schieferdecker, P. J. Mosterman (eds), Model-Based Testing for Embedded Systems. CRC Press (2011).
[5] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework - Parts 1-5. International Standard IS-9646. ISO, Geneve (1991).
[6] G. Luo, R. Dssouli, G. v. Bochmann, P. Venkataram, A. Ghedamsi, Test generation with respect to distributed interfaces. Computer Standards & Interfaces, Elsevier (1994), vol. 16, iss. 2, pp.119-132.
[7] B. Sarikaya, G. v. Bochmann, Synchronization and specification issues in protocol testing. In: IEEE Trans. Commun., IEEE Press, New York (1984), pp. 389-395.
[8] R. M. Hierons, M. G. Merayo, M. Núñez, Implementation relations and test generation for systems with distributed interfaces. Springer-Verlag (2012), Distributed Computing, vol. 25, no. 1, pp. 35-62.
[9] A. David, K. G. Larsen, M. Mikuionis, O. L. Nguena Timo, A. Rollet, Remote Testing of Timed Specifications. In: Proceedings of the 25th IFIP International Conference on Testing Software and Systems (ICTSS 2013), Springer, Heidelberg (2013), pp. 65-81.
[10] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, Springer, Heidelberg (2004), vol. 3098, pp. 87–124.
[11] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, Testing Real-Time Systems Using UPPAAL. In: Hierons, R. M., Bowen, J. P., Harman, M. (eds.) Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. LNCS, Springer, Heidelberg (2008), vol. 4949, pp. 77-117.
[12] G. Behrmann, A. David, K. G. Larsen, A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, Springer, Heidelberg (2004), vol. 3185, pp. 200-236.
[13] DTRON - Extension of TRON for distributed testing, http://www.cs.ttu.ee/dtron.
[14] J. Tretmans: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools, Springer-Verlag (1996), vol. 17, no. 3, pp. 103-120.
[15] R. Segala, Quiescence, fairness, testing, and the notion of implementation. In: Best, E. (eds.) 4th Intrenational Conference on Concurrency Theory (CONCUR'93). LNCS, Springer, Heidelberg (1993), vol. 715, pp. 324-338.
[16] J. Vain, M. Kääramees, M. Markvardt: Online testing of nondeterministic systems with reactive planning tester. In: Petre, L., Sere, K., Troubitsyna, E. (eds.) Dependability and Computer Engineering: Concepts for Software-Intensive Systems, IGI Global, Hershey (2012), pp. 113-150.
[17] A. Anier, J. Vain, Model based Continual Planning and Control for Assistive Robots. In: Proceedings of the International Conference on Health Informatics, SciTePress, Setúbal (2012), pp. 382-385.
[18] The Spread Toolkit, http://spread.org/.

Appendix 8

Publication V
J. Vain and E. Halling. Constraint-based testing scenario description language. *Proceedings of 13th Biennial Baltic Electronics Conference, BEC 2012*, IEEE:89–92, 2012

Databases and Information Systems IX
G. Arnicans et al. (Eds.)

doi:10.3233/978-1-61499-714-6-297

Automatic Distribution of Local Testers for Testing Distributed Systems

Jüri VAIN [1], Evelin HALLING, Gert KANTER, Aivo ANIER and Deepak PAL
Department of Computer Science, Tallinn University of Technology, Estonia
http://cs.ttu.ee/

Abstract. *Low-latency systems* where reaction time is primary success factor and design consideration, are serious challenge to existing integration and system level testing techniques. Modern cyber physical systems have grown to the scale of global geographic distribution and latency requirements are measured in nanoseconds. While existing tools support prescribed input profiles they seldom provide enough reactivity to run the tests with simultaneous and interdependent input profiles at remote front ends. Additional complexities emerge due to severe timing constraints the tests have to meet when test navigation decision time ranges near the message propagation time. Sufficient timing conditions for remote online testing have been proposed in remote Δ-testing method recently. We extend the Δ-testing by deploying testers on fully distributed test architecture. This approach reduces the test reaction time by almost a factor of two. We validate the method on a distributed oil pumping SCADA system case study.

Keywords. model-based testing, distributed systems, low-latency systems

1. Introduction

Modern large scale cyber-physical systems have grown to the size of global geographic distribution and their latency requirements are measured in microseconds or even nanoseconds. Such applications where latency is one of the primary design considerations are called *low-latency systems* and where it is of critical importance – to *time critical systems*. A typical example of distributed time critical system is smart energy grid (SEG) where delayed control signals can cause overloads and blackouts of whole regions. Thus, the proper timing is the main measure of success in SEG and often the hardest design concern.

Since large SEG-s systems are mostly distributed systems (by distributed systems we mean the systems where computations are performed on multiple networked computers that communicate and coordinate their actions by passing messages), their latency dynamics is influenced by many technical and non-technical factors. Just to name a few, energy consumption profile look up time (few milliseconds) may depend on the load profile, messaging middleware and the networking stacks of operating systems. Similarly, due to cache miss, the caching time can grow from microseconds to about hundred

[1]Corresponding Author: Jüri Vain; Department of Computer Science, Tallinn University of Technology, Akadeemia tee 15A, 19086 Tallinn, Estonia; E-mail: juri.vain@ttu.ee

milliseconds [1]. Reaching sufficient feature coverage by integration testing of such systems in the presence of numerous latency factors and their interdependences, is out of the reach of manual testing. Obvious implication is that scalable integration and system level testing presumes complex tools and techniques to assure the quality of the test results [2]. To achieve the confidence and trustability, the test suites need to be either correct by construction or verified against the test goals after they are generated. The need for automated test generation and their correctness assurance have given raise to model based testing (MBT) and the development of several commercial and academic MBT tools. In this paper, we interpret MBT in the standard way, i.e. as conformance testing that compares the expected behaviors described by the system requirements model with the observed behaviors of an actual implementation (implementation under test). For detailed overview of MBT and related tools we refer to [3] and [4].

2. Related Work

Testing distributed systems has been one of the MBT challenges since the beginning of the 90s. An attempt to standardize the test interfaces for distributed testing was made in ISO OSI Conformance Testing Methodology [5]. A general distributed test architecture, containing distributed interfaces, has been presented in Open Distributed Processing (ODP) Basic Reference Model (BRM), which is a generalized version of ISO distributed test architecture. First MBT approaches represented the test configurations as systems that can be modeled by finite state machines (FSM) with several distributed interfaces, called ports. An example of abstract distributed test architecture is proposed in [6]. This architecture suggests the Implementation Under Test (IUT) contains several ports that can be located physically far from each other. The testers are located in these nodes that have direct access to ports. There are also two strongly limiting assumptions: (i) the testers cannot communicate and synchronize with one another unless they communicate through the IUT, and (ii) no global clock is available. Under these assumptions a test generation method was developed in [6] for generating synchronizable test sequences of multi-port finite state machines. However, it was shown in [7] that no method that is based on the concept of synchronizable test sequences can ensure full fault coverage for all the testers. The reason is that for certain testers, given a FSM transition, there may not exist any synchronizable test sequence that can force the machine to traverse this transition. This is generally known as *controllability* and *observability* problem of distributed testers. These problems occur if a tester cannot determine either when to apply a particular input to IUT, or whether a particular output from IUT is generated in response to a specific input [8]. For instance, the controllability problem occurs when the tester at a port p_i is expected to send an input to IUT after IUT has responded to an input from the tester at some other port p_j, without sending an output to p_i. The tester at p_i is unable to decide whether IUT has received that input and so cannot know when to send its input. Similarly, the observability problem occurs when the tester at some port p_i is expected to receive an output from IUT in response to a given input at some port other than p_i and is unable to determine when to start and stop waiting. Such observability problems can introduce fault masking.

In [8], it is proposed to construct test sequences that cause no controllability and observability problems during their application. Unfortunately, offline generation of test sequences is not always applicable. For instance, when the model of IUT is non-deterministic it needs instead of fixed test sequences online testers capable of handling non-deterministic behavior of IUT. But even this is not always possible. An alternative is to construct testers that includes external coordination messages. However, that creates communication overhead and possibly the delay introduced by the sending of each message. Finding an acceptable amount of coordination messages depends on timing constraints and finally amounts to finding a tradeoff between the controllability, observability and the cost of sending external coordination messages.

The need for retaining the timing and latency properties of testers became crucial natively when time critical cyber physical and low-latency systems were tested. Pioneering theoretical results have been published on test timing correctness in [9] where a remote abstract tester was proposed for testing distributed systems in a centralized manner. It was proven that if IUT ports are remotely observable and controllable then 2Δ-condition is sufficient for satisfying timing correctness of the test. Here, Δ denotes an upper bound of message propagation delay between tester and IUT ports. However, this condition makes remote testing problematic when 2Δ is close to timing constraints of IUT, e.g. the length of time interval when the test input has to reach port has definite effect on IUT. If the actual time interval between receiving an IUT output and sending subsequent test stimulus is longer than 2Δ the input may not reach the input port in time and the test goal cannot be reached.

In this paper we focus on distributed online testing of low latency and time-critical systems with distributed testers that can exchange synchronization messages that meet Δ-delay condition. In contrast to the centralized testing approach, our approach reduces the tester reaction time from 2Δ to Δ. The validation of proposed approach is demonstrated on a distributed oil pumping SCADA system case study.

3. Preliminaries

3.1. Model-Based Testing

In model-based testing, the formal requirements model of implementation under test describes how the system under test is required to behave. The model, built in a suitable machine interpretable formalism, can be used to automatically generate the test cases, either offline or online, and can also be used as the oracle that checks if the IUT behavior conforms to this model. Offline test generation means that tests are generated before test execution and executed when needed. In the case of online test generation the model is executed in lock step with the IUT. The communication between the model and the IUT involves controllable inputs of the IUT and observable outputs of the IUT.

There are multiple different formalisms used for building conformance testing models. Our choice is Uppaal timed automata (TA) [10] because the formalism is designed to express the timed behavior of state transition systems and there exists a family of tools that support model construction, verification and online model-based testing [11].

3.2. Uppaal Timed Automata

Uppaal Timed Automata [10] (UTA) used for the specification of the requirements are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into a single system by the parallel composition known from the process algebra CCS. An example of a system of two automata comprised of 3 locations and 2 transitions each is given in Figure 1.

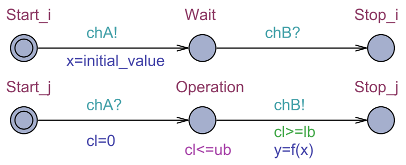


Figure 1. A parallel composition of Uppaal timed automata

The nodes of the automata are called *locations* and the directed edges *transitions*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. The initial locations of the automata are graphically denoted by an additional circle inside the location.

Synchronous communication between the processes is by hand-shake synchronization links that are called *channels*. A channel relates a pair of edges labeled with symbols for input actions denoted by e.g. `chA?` and `chB?` in Figure 1, and output actions denoted by `chA!` and `chB!`, where `chA` and `chB` are the names of the channels.

In Figure 1, there is an example of a model that represents a synchronous remote procedure call. The calling process Process_i and the callee process Process_j both include three locations and two synchronized transitions. Process_i, initially at location Start_i, initiates the call by executing the send action `chA!` that is synchronized with the receive action `chA?` in Process_j, that is initially at location Start_j. The location Operation denotes the situation where Process_j computes the output `y`. Once done, the control is returned to Process_i by the action `chB!`

The duration of the execution of the result is specified by the interval $[lb, ub]$ where the upper bound ub is given by the *invariant* `cl<=ub`, and the lower bound lb by the *guard condition* `cl>=lb` of the transition Operation $\rightarrow$ Stop_j. The *assignment* `cl=0` on the transition Start_j $\rightarrow$ Operation ensures that the clock cl is reset when the control reaches the location Operation. The global variables `x` and `y` model the input and output arguments of the remote procedure call, and the computation itself is modelled by the function `f(x)` defined in the declarations section of the Uppaal model.

The inputs and outputs of the test system are modeled using channels labeled in a special way described later. Asynchronous communication between processes is modeled using global variables accessible to all processes.

Formally the Uppaal timed automata are defined as follows. Let Σ denote a finite alphabet of actions $a, b, \ldots$ and C a finite set of real-valued variables p, q, r, denoting clocks. A guard is a conjunctive formula of atomic constraints of the form $p \sim n$ for $p \in C, \sim \in \{\geq, \leq, =, >, <\}$ and $n \in \mathbb{N}^+$. We use $G(C)$ to denote the set of clock guards. A timed automaton A is a tuple $\langle N, l_0, E, I \rangle$ where N is a finite set of locations (graphically denoted by nodes), $l_0 \in N$ is the initial location, $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges (an edge is denoted by an arc) and $I : N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $p \leq n$ or $p < n, n \in \mathbb{N}^+$. Without the loss of generality we assume that guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints on integer variables. Similarly to clock conditions, the propositions on integer variables k are of the form $k \sim n$ for $n \in \mathbb{N}$, and $\sim \in \{\leq, \geq, =, >, <\}$. For the formal definition of Uppaal TA full semantics we refer the reader to [12] and [10].

4. Remote Testing

The test purpose most often used in MBT is conformance testing. 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 [13] is to check if the behavior 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 behavior 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 [14]) 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 behavior of the IUT and from the coverage criteria that constrain the behavior defined in IUT model with only those addressed by the test purpose. In our approach Uppaal Timed Automata (UTA) [10] are used as a formalism for modeling IUT behavior. 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 behavior 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 [15], 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.

4.1. Centralized Remote Testing

Let us first consider a centralized tester design case. In the case of centralized tester, all test inputs are generated by a single monolithic tester. This means that the centralized tester will generate an input for the IUT, waits for the result and continues with the next set of inputs and outputs until the test scenario has been finished. Thus, the tester has to wait for the duration it takes the signal to be transmitted from the tester to the IUT's ports and the responses back from ports to the tester. In the case of IUT being distributed in a way that signal propagation time is non-negligible, this can lead into a situation where the tester is unable to generate the necessary input for the IUT in time due to message propagation latency. These timing issues can render testing an IUT impossible if the IUT is a distributed real-time system.

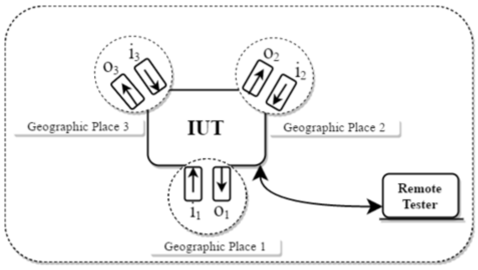


Figure 2. Remote tester communication architecture

To be more concrete, let us consider the remote testing architecture depicted in Figure 2 and the corresponding model depicted in Figure 3 and 4. In this case the IUT has 3 ports (p_1, p_2, p_3) in geographically different places to interact within the system, inputs i_1, i_2 and i_3 at ports p_1, p_2 and p_3 respectively and outputs o_1 at port p_1, o_2 at port p_2, o_3 at port p_3.

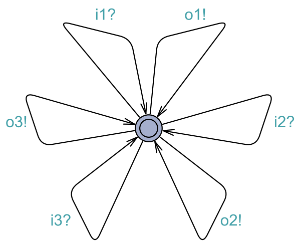


Figure 3. IUT model

We model a multi-ports timed automata by splitting the edges with multiple communication actions to a sequence of edges each labeled with exactly one action and connected via committed locations, so that all ports of such group are updated at the same time. In Figure 4 the labels on the edges represent the transitions and the transition tuple (`L0`, `L1`, $i_1!/(o_1?, o_2?)$) is represented by sequence of edges each labeled with exactly one action and connected via committed locations. For example the sequence of edges from location `L0` to `L1` with labels $i_1!$, $o_1?$ and $o_2?$ represents the multiple communication actions where the input $i_1!$ at port p_1 in location `L0` being able to trigger a transition that leads to the output $o_1?$ and $o_2?$ at ports p_1, p_2 respectively and the location becoming `L1`.

Using such splitting of edges with committed locations, we model a three port automata shown in Figure 4 where the tester sends an input i_1 to the port p_1 at `Geographic Place 1` and receives a response or outputs o_1 and o_2 from IUT at `Geographic Place 1` and `Geographic Place 2` respectively. After receiving the result, the tester is in location `L1`, it gets both i_3 on port p_3 and i_2 on port p_2. Then, either it follows the intended

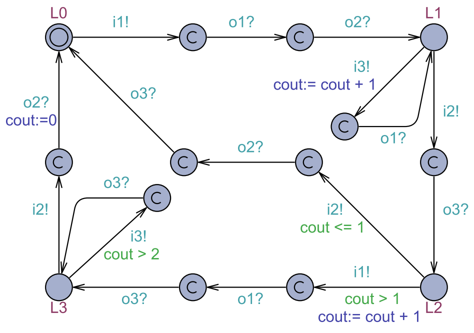


Figure 4. Remote Tester model

path sending i_3 before i_2, or it sends i_2 before i_3. If tester decides to send i_3 before i_2 it receives an output o_1 at port p_1 and returns to location `L1`. The transition is a self loop if its start and end locations are the same. If tester decides to send i_2, the IUT responds with an output o_3 at port p_3. Now, the tester is in location `L2`, it gets both i_1 on port p_1 and i_2 on port p_2. Based on guard condition and previously triggered inputs and received outputs the next input is sent to IUT and tester continues with the next set of inputs and outputs until the test scenario has been finished.

The described IUT is a real-time distributed system, which means that it has strict timing constraints for messaging between ports. More specifically, after sending the first input i_1 to port p_1 at `Geographic Place 1` and after receiving the response o_1 and o_2 at `Geographic Place 1` and `Geographic Place 2` respectively, the tester needs to decide and send the next input i_2 to port p_2 at `Geographic Place 2` or input i_3 to port p_3 at `Geographic Place 3` in Δ time. But, due to the fact that the tester is not at the same geographical place as the distributed IUT, it is unable to send the next input in time as the time it takes to receive the response and send the next input amounts to 2Δ, which is double the time allotted for the next input signal to arrive.

Consequently, the centralized remote testing approach is not suitable for testing a real-time distributed system if the system has strict timing constraints with non negligible signal propagation times between system ports. To overcome this problem, the centralized tester is decomposed and distributed as described in the next section.

5. Distributed Testing

The shortcoming of the centralized remote testing approach is mitigated with extending the Δ-testing idea by decomposing the monolithic remote tester into multiple local testers. These local testers are directly attached to the ports of the IUT. Thus, instead of bidirectional communication between a remote tester and the IUT, only unidirectional synchronization between the local testers is required. The local testers are generated in two steps: at first, a centralized remote tester is generated by applying the reactive planning online-tester synthesis method of [16], and second, a set of synchronizing local testers is derived by decomposing the monolithic tester into a set of location specific tester instances. Each tester instance needs to know now only the occurrence of i/o events

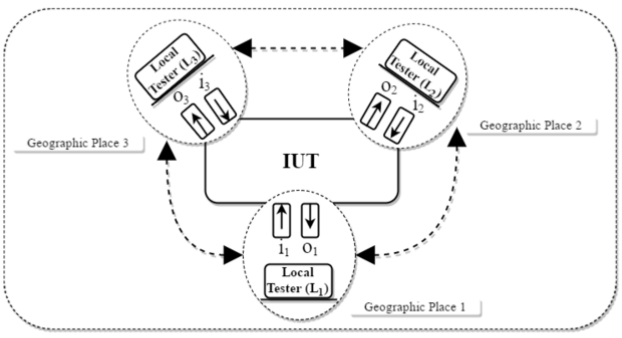


Figure 5. Distributed Local testers communication architecture

at other ports which determine its behavior. Possible reactions of the local tester to these events are already specified in its model and do not need further feedback to the event sender. The decomposing preserves the correctness of testers so that if the monolithic remote tester meets 2Δ requirement then the distributed testers meet (one) Δ-controllability requirement.

We apply the algorithm described in 5.1 to transform the centralized testing architecture depicted in Figure 2 into a set of communicating distributed local testers, the architecture of which is shown in Figure 5. After applying the algorithm, the message propagation time between the local tester and the IUT port has been eliminated because the tester is attached directly to the port. This means that the overall testing response time is also reduced, because previously the messages had to be transmitted over a channel with latency bidirectionally. The resulting architecture mitigates the timing issue by replacing the bidirectional communication with a unidirectional broadcast of the IUT output signals between the distributed local testers. The generated local tester models are shown in Figure 6, Figure 7, Figure 8 and Figure 9.

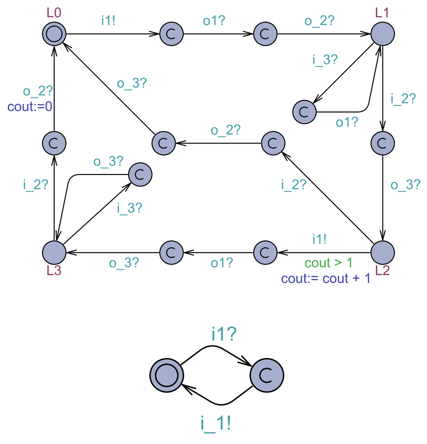


Figure 6. Local tester at Geographic Place 1

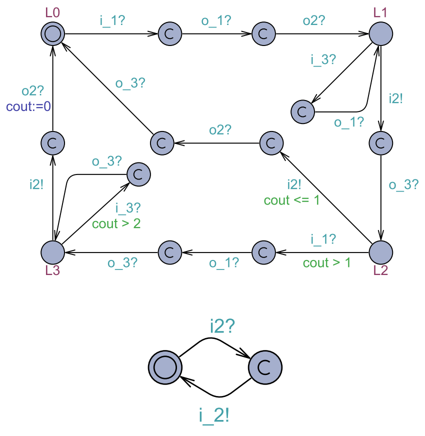


Figure 7. Local tester at Geographic Place 2

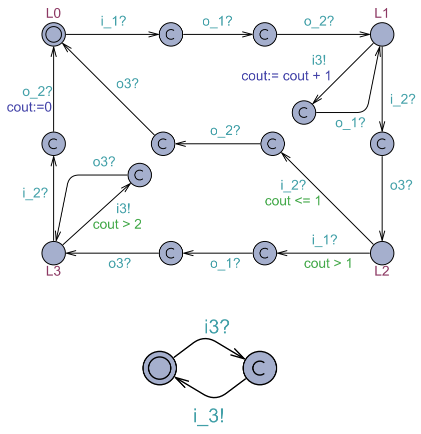


Figure 8. Local tester at Geographic Place 3

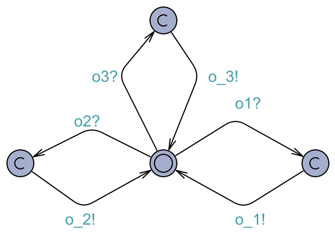


Figure 9. Output Event Synchronizer

5.1. Tester Distribution Algorithm

Let M^{MT} denote a monolithic remote tester model generated by applying the reactive planning online-tester synthesis method [16]. $Loc(IUT)$ denotes a set of geographically different port locations of IUT. The number of locations can be from 1 to n, where $n \in \mathbb{N}$ i.e. $Loc(IUT) = \{l_n | n \in \mathbb{N}\}$. Let P_{l_n} denotes a set of ports accessible in the location l_n.

1. For each l, $l \in Loc(IUT)$ we copy M^{MT} to M^l to be transformed to a location specific local tester instance.
2. For each M^l we go through all the edges in M^l. If the edge has a synchronizing channel and the channel does not belong to the the set of ports P_{l_n}, we do the following:
   - if the channel's action is *send*, we replace it with the co-action *receive*.
   - if the channel's action is *receive*, we do nothing.
3. For each M^l we add one more automaton that duplicates the input signals from M^l to IUT, attached to the set of ports P_{l_n} and broadcasts the duplicates to other local testers to synchronize the test runs at their local ports. Similarly the IUT local output event observations are broadcast to other testers for synchronization purposes like automaton in Figure 9.

6. Correctness of Tester Distribution Algorithm

To verify the correctness of distributed tester generation algorithm we check the bi-simulation equivalence relation between the model of monolithic centralized tester and that of distributed tester. For that the models are composed by parallel compositions so that one has a role of words generator on i/o alphabet and other the role of words acceptor machine. If the i/o language acceptance is established in one direction then the roles of models are reversed. Since the i/o alphabets of remote tester and distributed tester differ due to synchronizing messages of distributed tester the behaviors are compared based on the i/o alphabet observable on IUT ports only. Second adjustment of models to be made for bi-simulation analysis is the reduction of message propagation delays to uniform basis either on Δ or 2Δ in both models. Assume (due to closed world assumption used in MBT):

- the centralized remote tester model: $M^{remote} = TA^{IUT} \parallel TA^{r-TST}$
- the distributed tester model: $M^{distrib} = TA^{IUT} \parallel \shortparallel_i TA_i^{d-TST}$ $i = [1, n]$, n - number of ports locations.
- to unify the timed words $TW(M^{remote})$ and $TW(M^{distrib})$ the communication delay between IUT and Tester is assumed.

Definition (correctness of tester distribution mapping): The mapping $M^{remote} \xmapsto{Algorithm} M^{distrib}$ is correct if TA^{r-TST} and $\shortparallel_i TA_i^{d-TST}$ are observation bisimilar, i.e. if TA^{r-TST} and $\shortparallel_i TA_i^{d-TST}$ are respectively generating and accepting automata on common i/o alphabet $\Sigma^i \cup \Sigma^o$ then all timed words $TW(TA^{r-TST})$ are recognizable by $\shortparallel_i TA_i^{d-TST}$ and all timed words $TW(\shortparallel_i TA_i^{d-TST})$ are recognizable by TA^{r-TST}.

Here, alphabet $\Sigma^i \cup \Sigma^o$ includes i/o symbols used at IUT-TESTER interfaces of M^{remote} and $M^{distrib}$.

Correctness verification of the distribution mapping:

Step 1: (Constructing generating-accepting automata synchronous composition):

- label each output action of TA^{r-TST} with output symbol a! and its co-action in $\shortparallel_i TA_i^{d-TST}$ with input symbol a?;
- define parallel composition $TA^{r-TST} \parallel \shortparallel_i TA_i^{d-TST}$ with synchronous i/o actions.

Step 2: (Bisimilarity proof by model checking): TA^{r-TST} and $\shortparallel_i TA_i^{d-TST}$ are observation bisimilar if following holds: $M^{remote} \models \texttt{not deadlock} \wedge M^{distrib} \models \texttt{not deadlock} \Rightarrow TA^{r-TST} \parallel \shortparallel_j TA_j^{d-TST} \models \texttt{not deadlock}$ $j = [1, n]$, n - number of local testers , i.e. the composition of bisimilar testers must be non-blocking if the testers composed with IUT model separately are non-blocking.

7. Case Study

7.1. Use Case

The benefit of using the proposed method is demonstrated in the use case of an EMS (Energy Management System) which is integrated into the SCADA (Supervisory Control And Data Acquisition) system of an industrial consumer. An EMS is essentially a load balancing system. The target of the balancing system is the load on power supplies called feeders to an industrial consumer. These industrial power consumers have multiple feeders to power the devices required for their operations (e.g., pumps and pipeline heating systems). The motivation for balancing the power consumption between the feeders stems from the fact that the power companies can enforce fines on the industrial consumers if the power consumption exceeds certain thresholds due to safety considerations and possible damage to the equipment. Therefore, the consumer is motivated to share the power consuming devices among the feeders minimize or eliminate such energy consumption spikes completely.

Let us consider a use case in which an oil terminal has two feeders and multiple power consuming devices (consumers). The number of consumers can range from some to many. In our use case we have 32 consumers, but in other cases it can be more. These consumers are both pumps and pipeline heating systems. The pumps have a high surge power consumption when starting up which must be taken into consideration when designing an EMS. The EMS monitors the current consumption by polling the consumers via a communication system (e.g., PROFIBUS, CAN bus or Industrial Ethernet). The PROFIBUS communication system is standardized in EN 50170 international standard.

Because the oil terminal stores oil it is considered an explosion hazard area and therefore, a special communication system that is certified for explosive areas - PROFIBUS PA (Process Automation) is used. PROFIBUS PA meets the 'Intrinsically Safe' (IS) and bus-powered requirements defined by IEC 61158-2. The maximum transfer rate of PROFIBUS PA is 31.25 kbit/s which can limit the system response speed if there are many devices connected to the PROFIBUS bus and each device has significant input and output data load.

The EMS is able to switch devices from being supplied from either feeder. Ideally, the power consumption is shared equally among both feeders at all times. This means that the EMS monitors the devices and switches devices over to other feeders if the power consumption is unbalanced among the feeders. In normal operation, the feeder loads are kept sufficiently low to accommodate new devices starting up in a way that the surge consumption will not exceed the threshold power of the feeders.

The EMS polls every power consumer periodically and updates the total consumption. Based on this total consumption, the EMS will command the power distribution devices to switch around from first feeder to the second in case the load on the first feeder is higher than on the second and vice versa.

In our use case we simulate the power consumption of the devices as the input to the IUT. The tester monitors the output (the EMS feeder load values). The test purpose is to verify that neither of the power loads exceed the specified threshold. Exceeding this limit might cause equipment damage and the power company can impose fines upon violating this limit.

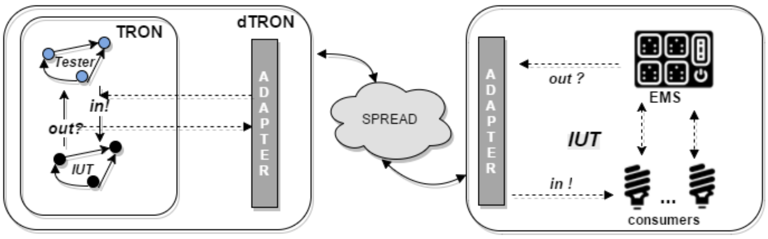


Figure 10. Case Study Test Architecture

The test architecture is depicted in Figure 10. In the right side of the figure, we can see the EMS and consumers as the implementation under test. The test model and test runner is on the left side. The test is executed via DTRON, which transmits the inputs and outputs via Spread. In the IUT and tester models we are going to introduce, the signals prefixed with $i\_$ or $o\_$ are synchronizing signals sent through Spread message serialization service . The signals without the aforementioned prefixes are internal signals which are not published to the Spread network. The input to the IUT is provided by the remote tester model is depicted in Figure 13 which simulates the device power consumption levels and creates challenging scenarios for the EMS. The EMS queries the consumers which are modeled in Figure 12 and balances the load between the feeders based on the total power consumption monitoring data . The EMS model is shown in Figure 11 which displays the querying loop. The querying is performed in a loop due to the semi-duplex nature of communication in PROFIBUS networks. The EMS also takes the maximum power limit into account as the total power consumption must not exceed this level. This can be seen in the remote tester model shown in Figure 13. Remote tester nondeterministically selects a consumer and sends the level of energy consumption for that particular device to the input port of the IUT. Then the remote tester waits s time units before requesting the current feeder energy levels. On the model, it is indicated as *i_get_line_balance*!. After receiving the current values the tester will check whether they are within allowed range. If the values exceed the limit the test verdict is fail. Otherwise the tester will continue with the next iteration.

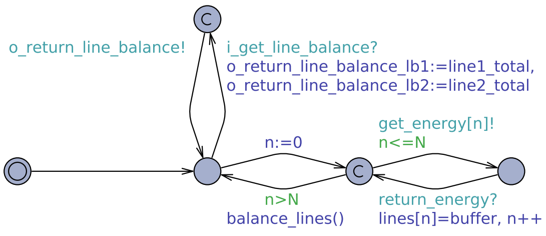


Figure 11. Energy Management System model

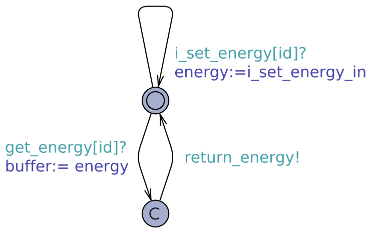


Figure 12. Consumer model

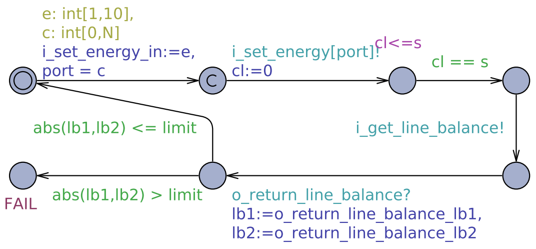


Figure 13. Remote Tester model

The communication delay between receiving the signal from EMS with the current feeder energy levels and sending input to the IUT is 2Δ. According to the specification the system must stabilize the load between feeders in stabilization time limit s after receiving the input. If Δ is very close to system stabilization time limit s indicated in the remote tester model in Figure 13 the remote tester fails to send the signal in time to the IUT.

For this reason, we introduce the distributed tester Figure 14 where each local component of the tester is closely coupled to the IUT input ports. As shown in chapter 5 this approach reduces the delay by up to Δ. This guarantees that after receiving the output from EMS we can send new input to IUT within less than s time units.

7.2. Test Execution Environment DTRON

Uppaal TRON [11] is a testing tool, based on Uppaal engine, suited for black-box conformance testing of timed systems [11]. DTRON [13] extends this enabling distributed

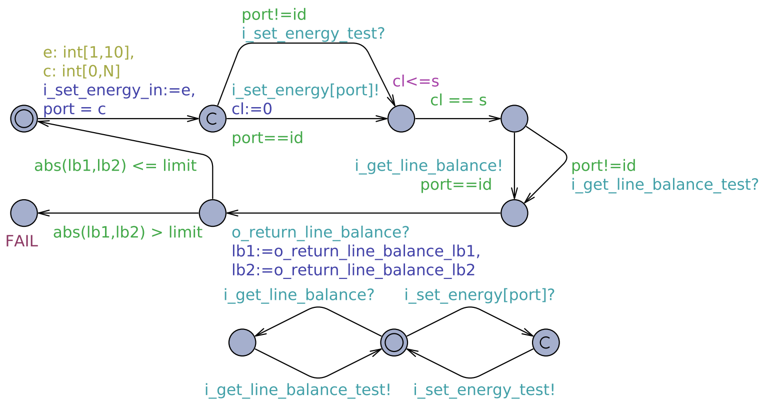


Figure 14. Parametrized local tester template for distributed testing

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 [18]. Subscribers can be other IUT 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 verification of the deployed test configuration we make corresponding time parameter adjustments in the IUT model.

8. Conclusion

We extend the Δ-testing method proposed originally for single remote tester by introducing multiple local testers on fully distributed test architecture where testers are attached directly to the ports of IUT. Thus, instead of bidirectional communication between a remote tester and IUT only unidirectional synchronization between the local testers is needed in given solution. A constructive algorithm is proposed to generate local testers in two steps: at first, a monolithic remote tester is generated by applying the reactive planning online-tester synthesis method of [16], and second, a set of synchronizing local testers is derived by partitioning the monolithic tester into a set of location specific tester instances. The partitioning preserves the correctness of testers so that if the monolithic remote tester meets 2Δ requirement then the distributed testers meet (one) Δ-controllability requirement. Second contribution of the paper is that distributed testers are generated as Uppal Timed Automata. According to our best knowledge the real time distributed testers have not been constructed automatically in this formalism yet. As for method implementation, the local testers are executed and communicating via distributed test execution environment DTRON [13]. We demonstrate that the distributed deployment architecture supported by DTRON and its message serialization service allows reducing the total test reaction time by almost a factor of two. The validation of proposed approach is demonstrated on an Energy Management System case study.

References

[1] A. Brook, Evolution and Practice: Low-latency Distributed Applications in Finance. Queue - Distributed Computing, ACM, New York (2015), vol. 13, no. 4, pp. 40-53.

[2] G. Hackenberg, M. Irlbeck, V. Koutsoumpas, and D. Bytschkow, Applying formal software engineering techniques to smart grids. In Software Engineering for the Smart Grid (SE4SG), 2012 International Workshop, IEEE (2012), pp. 50-56.

[3] M. Utting, A. Pretschner, and B. Legeard, A taxonomy of Model-based Testing. Software Testing, Verification & Reliability, John Wiley and Sons Ltd., Chichester, UK (2012), vol. 22, iss. 5, pp. 297-312.

[4] J. Zander, I. Schieferdecker, P. J. Mosterman (eds), Model-Based Testing for Embedded Systems. CRC Press (2011).

[5] ISO. Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework - Parts 1-5. International Standard IS-9646. ISO, Geneve (1991).

[6] G. Luo, R. Dssouli, G. v. Bochmann, P. Venkataram, A. Ghedamsi, Test generation with respect to distributed interfaces. Computer Standards & Interfaces, Elsevier (1994), vol. 16, iss. 2, pp.119-132.

[7] B. Sarikaya, G. v. Bochmann, Synchronization and specification issues in protocol testing. In: IEEE Trans. Commun., IEEE Press, New York (1984), pp. 389-395.

[8] R. M. Hierons, M. G. Merayo, M. Núñez, Implementation relations and test generation for systems with distributed interfaces. Springer-Verlag (2012), Distributed Computing, vol. 25, no. 1, pp. 35-62.

[9] A. David, K. G. Larsen, M. Mikuionis, O. L. Nguena Timo, A. Rollet, Remote Testing of Timed Specifications. In: Proceedings of the 25th IFIP International Conference on Testing Software and Systems (ICTSS 2013), Springer, Heidelberg (2013), pp. 65-81.

[10] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets: Advances in Petri Nets. LNCS, Springer, Heidelberg (2004), vol. 3098, pp. 87–124.

[11] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, Testing Real-Time Systems Using UPPAAL. In: Hierons, R. M., Bowen, J. P., Harman, M. (eds.) Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. LNCS, Springer, Heidelberg (2008), vol. 4949, pp. 77-117.

[12] G. Behrmann, A. David, K. G. Larsen, A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, Springer, Heidelberg (2004), vol. 3185, pp. 200-236.

[13] DTRON - Extension of TRON for distributed testing, http://www.cs.ttu.ee/dtron.

[14] J. Tretmans: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools, Springer-Verlag (1996), vol. 17, no. 3, pp. 103-120.

[15] R. Segala, Quiescence, fairness, testing, and the notion of implementation. In: Best, E. (eds.) 4th Intrenational Conference on Concurrency Theory (CONCUR'93). LNCS, Springer, Heidelberg (1993), vol. 715, pp. 324-338.

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

[17] A. Anier, J. Vain, Model based Continual Planning and Control for Assistive Robots. In: Proceedings of the International Conference on Health Informatics, SciTePress, Setúbal (2012), pp. 382-385.

[18] The Spread Toolkit, http://spread.org/.

Curriculum Vitae

1. Personal data

Name	Evelin Halling
Date and place of birth	4 October 1978 Tartu, Estonia
Nationality	Estonian

2. Contact information

Address	Tallinn University of Technology, Faculty of Information Technology, Department of Software Science, Ehitajate tee 5, 19086 Tallinn, Estonia
Phone	+372 620 2325
E-mail	evelin.halling@ttu.ee

3. Education

2011–. . .	Tallinn University of Technology, Department of Software Science, Computer Science, PhD studies
2008–2011	Tallinn University of Technology, Faculty of Information Technology, Informatics, MSc
2008–2011	The Estonian Information Technology College, IT Systems Development, BSc

4. Language competence

Estonian	native
English	fluent

5. Professional employment

2012 –	Tallinn University of Technology, Lecturer
2009 – 2011	Airem OÜ, Software architect
2001 – 2009	Tieto Estonia AS, Software developer
1998 – 2000	MicroLink AS, IT support
1996 – 1997	Miksike OÜ, IT specialist

6. Field of research

- Formal methods
- Model-based testing

Papers

1. Vain, Jüri, and Halling, Evelin. Constraint-based testing scenario description language. BEC 2012 : 2012 13th Biennial Baltic Electronics Conference [Proceedings, Tallinn University of Technology, October 3-5, 2012, Tallinn, Estonia]. Piscataway, NJ: IEEE (2012): 89-92. (ETIS:3.1)

2. 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). (ETIS:3.1)

3. Ernits, J.; Halling, E.; Kanter, G.; Vain, J. (2015). Model-based integration testing of ROS packages: a mobile robot case study. 2015 IEEE European Conference on Mobile Robots : Lincoln, UK, September 2-4, 2015, Proceedings. Lincoln: IEEE, [1-7]. (ETIS:3.1)

4. Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Automatic Distribution of Local Testers for Testing Distributed Systems. In: Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. (Ed.). Databases and Information Systems IX : Selected Papers from the Twelfth International Baltic Conference, DB&IS 2016 (297-310). Amsterdam: IOS Press. (Frontiers in Artificial Intelligence and Applications; 291). (ETIS:3.1)

5. Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Model-based testing of real-time distributed systems. Databases and Information Systems : 12th International Baltic Conference, DB&IS 2016, Riga, Latvia, July 4-6, 2016, Proceedings. Ed. Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. Cham: Springer, 271-286. (Communications in Computer and Information Science; 615) (ETIS:3.1)

Elulookirjeldus

1. Isikuandmed

Nimi	Evelin Halling
Sünniaeg ja -koht	04.10.1978, Tartu, Eesti
Kodakondsus	Eesti

2. Kontaktandmed

Aadress	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Tarkvarateaduse Instituut, Ehitajate tee 5, 19086 Tallinn, Estonia
Telefon	+372 620 2325
E-post	evelin.halling@ttu.ee

3. Haridus

2011–. . .	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvutiteadus, doktoriõpe
2008–2011	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Informaatika, MSc
2001–2006	Eesti Infotehnoloogia Kolledz, IT süsteemide arendus, BSc

4. Keelteoskus

eesti keel	emakeel
inglise keel	kõrgtase

5. Teenistuskäik

2012 –	Tallinna Tehnikaülikool, Lektor
2009 – 2011	Airem OÜ, Tarkvara arhitekt
2001 – 2009	Tieto Estonia AS, Tarkvara arendaja
1998 – 2000	MicroLink AS, IT tugi
1996 – 1997	Miksike OÜ, IT spetsialist

6. Teadustöö põhisuunad

- Formaalsed meetodid
- Mudelipõhine testimine

7. Teadustegevus

Teadusartiklite, konverentsiteeside ja konverentsiettekannete loetelu on toodud ingliskeelse elulookirjelduse juures.