TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
19/2020

Model-Based Testing Framework for Autonomous Multi-Robot Systems

GERT KANTER

PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science
The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Informatics on June 15, 2020

Supervisor: Professor Jüri Vain
Department of Software Science
School of Information Technologies
Tallinn University of Technology
Tallinn, Estonia

Opponents: Anatoliy Gorbenko, PhD
Leeds Beckett University
Leeds, United Kingdom

Dragos Truscan, PhD
Åbo Akademi University
Turku, Finland

Defence of the dissertation: July 3, 2020, Tallinn

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

Gert Kanter

signature

European Union
European Regional
Development Fund

Investing
in your future

Copyright: Gert Kanter, 2020
ISSN 2585-6898 (publication)
ISBN 978-9949-83-558-4 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-559-1 (PDF)
Printed by Koopia Niini & Rauam

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
19/2020

Mudelipõhine testimisraamistik autonoomsetele multirobotsüsteemidele

GERT KANTER

TALTECH KIRJASTUS

Contents

List of Publications

Publication I: G. Kanter and J. Vain. Model-based testing of autonomous robots using TestIt. *Journal of Reliable Intelligent Environments*, 6(1):1–17, 2020 (ETIS category 1.1)

Publication II: G. Kanter and J. Vain. Testit: an open-source scalable long-term autonomy testing toolkit for ros. In *Proceedings of the 10th International Conference Dependable Systems, Services and Technologies, DESSERT'2019*, pages 45–50, 2019 (ETIS category 3.1)

Publication III: G. Kanter, J. Vain, S. Srinivasan, and S. Ramaswamy. Provably correct configuration management of precision feeding in agriculture4.0. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 1631–1637, 2019 (ETIS category 3.1)

Publication IV: J. Vain, G. Kanter, and A. Anier. Learning timed automata from interaction traces. In *14th IFAC Symposium on Analysis, Design, and Evaluation of Human Machine Systems, HMS 2019*, volume 52-19, pages 205–210, 2019 (ETIS category 3.1)

Publication V: J. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ros packages: a mobile robot case study. In *2015 IEEE European Conference on Mobile Robots*, pages 1–7. IEEE, 2015 (ETIS category 3.1)

Publication VI: J. Vain, G. Kanter, and S. Srinivasan. Model based testing of distributed time critical systems. In *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 99–105. IEEE, 2017 (ETIS category 3.1)

Author’s Contributions to the Publications

I In Publication I, I was the main author, developed the model-based testing toolkit TestIt, carried out the experiments, analysed the results, prepared the figures, and wrote the manuscript.

II In Publication II, I was the main author, developed the presented toolkit, carried out the experiments, analysed the results, prepared the figures, and wrote the manuscript

III In Publication III, I developed and validated the provably correct test configuration management on autonomous multi-robot case study (farm feeding system).

IV In Publication IV, I defined the learning context and the extraction of model learning assumptions, validated the learning algorithm on IEEE1394 leader election protocol.

V In Publication V, I developed the framework for connecting Dtron and ROS, conducted simulation experiments, specified the test coverage criteria, validated the test configuration.

VI In Publication VI, I developed the correctness criteria for test deployment in distributed architectures.

Foreword

The field of autonomous systems research and development is expanding rapidly. The level of autonomy exhibited by such systems is also increasing, perhaps only surpassed by the expectation for them by their potential users. Besides the expectations of relieving human labor from routine or dangerous jobs, there is also an acute need for safety and trust around these autonomous systems. This is especially true as these systems are beginning to become more commonplace in environments where they are supposed to collaborate safely and seamlessly with non-robotic specialists and bystanders. Ensuring that the participants in the highly dynamic environments come to no harm relies directly on the quality assurance of autonomous systems.

A need for safe and correct behavior is further exacerbated by the fact that the likelihood of multiple autonomous systems encountering each other is also increasing. This adds a layer of complexity in ensuring autonomous systems operate safely and correctly even in dynamic multi-party situations.

The advances in autonomy are largely thanks to improvements in software which in turn has been enabled by more capable and affordable hardware. As part of quality assurance process, the autonomous system behavior powered by its software needs to be tested and validated according to much higher standards than any common office software or web application.

This dissertation is focused on model-based testing of autonomous robotic systems with the aim of improving their quality assurance process and providing novel methods and tools for its automation.

Abbreviations

AAL	Adapter Action Language
API	Application Programming Interface
AS	Autonomous System
AWS	Amazon Web Services
AWS EC2	Amazon Web Services Elastic Compute Cloud
BDI	Belief-Desire-Intention
CADP	Construction and Analysis of Distributed Processes
CAN	Controller Area Network
CI	Continuous Integration
CLI	Command Line Interface
CPS	Cyber-Physical System
DREAM	Distributed Real-time Embedded Analysis Method
DTRON	Distributed Testing Real-time systems ONline
EFSM	Extended Finite State Machine
fMBT	Free Model-Based Testing
HAROS	High Assurance ROS
HRI	Human-Robot Interaction
ioco	Input Output Conformance
IOTS	Input-Output Transition System
JSXM	Java Stream X-Machines
MARL	Multi-Agent Reinforcement Learning
MBT	Model-Based Testing
MRS	Multi-Robot System
OPRoS	Open Platform for Robotic Services
PCD	Provably Correct Development
QA	Quality Assurance
REST	Representational State Transfer
ROS	Robot Operating System
rtioco	Relativized Timed Input Output Conformance
SDLC	Software Development Life Cycle
SMACH	State MACHine
SUT	System Under Test

SXM	Stream X-Machines
SysML	Systems Modeling Language
TDD	Test-Driven Development
TRON	Testing Real-time systems ONline
TTrS	Symbolic Timed Trace
UML	Unified Modeling Language
UTA	Uppaal Timed Automata

Terms

Container	A container is a standard unit of software that packages up software and all its dependencies so that the application runs from one computing environment to another. [7]
Cyber-Physical System	A cyber-physical system consists of a collection of computing devices communicating with one another and interacting with the physical world via sensors and actuators in a feedback loop. [8]
Multi-Robot Systems	A collection of two or more autonomous robots working together to achieve some well defined goals [9].
Provably correct development	Development process that can be proved to be correct using formal methods and mathematical proofs.
Real-Time System	A Real-time system is a system that is subjected to guaranteeing a response within a specified timing constraint. The timing constraint can specify a hard real-time system that is never allowed to miss a deadline and soft real-time system that can occasionally miss a deadline without disastrous consequences.
Robot	A robot is an automatic or semiautomatic machine capable of purposeful motion in response to its surroundings in an unstructured environment. [10]
Robotics	The science and technology of robots. [11]
Simulator	Software that is designed to computationally reproduce the expected physical behavior of the CPS.
Software Defect	An error in coding or logic that causes a program to malfunction or to produce incorrect results. [12]
System Under Test	SUT is the software component or the robotic system that is being tested. The SUT can also be a collection of systems under test as is the case in multi-robot systems.
Testing	The process of investigating that CPS or software meets the specified requirements, responds correctly to all system inputs, performs its functions within acceptable time and is usable to the required level.
Validation	The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies requirements. [13]
Verification	The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [13]

Verification & Validation	The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements. [13]

List of Figures

1 Introduction

1.1 Chapter overview

In this chapter, the context of the dissertation is introduced and the motivation for developing automated testing tools for autonomous robotic systems is described.

1.2 Terminology

This dissertation contains several terms that are closely related: cyber-physical systems (CPS), autonomous systems (AS), and multi-robot systems (MRS). CPS are an umbrella term for a large set of systems in which physical and virtual (i.e., software) components are combined. Examples of CPS categories include among other things smart power grid, industrial process control, aerospace, smart homes and buildings, medical and health-care systems, and robotics [14]. This categorization establishes robots as an instance of CPS. Robots can be divided into automated (e.g., automated guided vehicles [15]) and autonomous. Finally, robot systems can be divided based on the size of the collective into single robot or multi-robot systems [16].

This dissertation focuses on autonomous MRS testing but since a vast amount of theory and techniques have been developed by the research community for wider areas such as AS or CPS which are applicable to a more narrow area such as MRS as well. Therefore, these terms are used where appropriate.

1.3 Motivation

CPS testing is critical as defects have had fatal outcomes. At the time of preparing the dissertation, there have been five traffic accidents with driver fatalities with Tesla Autopilot system and one accident in which a pedestrian was killed [17].

The damages that software defects cause can also be financial. For example, Ariane 5 launcher failure in which a float number conversion caused a disaster which resulted in a destruction of a 500 million USD payload. These facts exemplify the importance of testing CPS on the whole but great care must be taken with software testing since not only hardware failures can have far-reaching and fatal consequences.

According to Gartner's assessment on autonomous robotics in its report [18] within the next 10 years autonomous vehicles and robot systems will conquer the public space and dominate in most of the economic sectors. The term autonomous systems covers systems which involve most diverse forms of vehicles, such as transport, robots, or devices that are capable of motion in the physical world in a self-controlling manner without direct human intervention. As detailed in [19], an AS is characterized by the fact that it is capable of independently heading for and achieving a given mission goal. AS functions can be grouped as follows:

- Sensing: the system can capture the signals and data relevant to its mission and occurring in its environment.
- Perceiving: the system can recognize patterns or situations based on signals and data.
- Analyzing: the system can identify options for action appropriate to the respective situation.
- Planning: the system can select appropriate or best options for action.

- Acting: the system can implement the chosen action correctly and on time.

Regardless the attractive perspective of using AS, the major public concern and challenge for AS developers is to ensure that these systems are sufficiently safe and reliable so that they are verified and can be approved for use in public space.

Testing AS requires careful balancing of the design complexity between the hardware and software components. A recent evaluation of industrial and societal trends reported in [20] that AB Volvo estimates software driving 70 percent of all innovation in its trucks and Volvo Cars estimates that electronics and software drive 80 to 90 percent of its innovation. The evaluation also reports that telecommunications company Ericsson's focus has shifted, with more than 80 percent of its research and development budget being dedicated to software. This shift in focus makes software testing increasingly important as the importance of software in not only CPS but in all other areas is growing.

A recent survey on the current state of the art focusing on formal specification and verification of autonomous robotic systems [21] identifies and categorizes the challenges posed by the formal approaches for the specification and verification of autonomous robots. The authors find that model checking is by far the most popular formal verification approach in verification of autonomous robotic systems. The study shows that formal software frameworks, integrated formal methods, theorem proving and runtime monitoring are less favoured by the research community. The authors of [21] conclude that field tests and using simulations are both useful tools for robotic system development, but stress that formal verification including formal model-based testing are crucial, especially at the early stages of development when field tests of the control software are infeasible or dangerous. They identify a strong need for tool support which supports a combination or integration of formal methods since no single formalism is capable of adequately capturing that all aspects of a robotic system behave as expected. A comprehensive analysis of the state of the art is presented in Section 2.3.1.

As modern software development practices favor agile software development life cycle (SDLC) [22] supported by continuous integration (CI) methodology, the modern toolkits for testing must also be designed with such practices in mind. The SDLC and CI aspects constitute a second design requirement for modern testing tools.

Model-based testing (MBT) has been shown to be effective in AS verification (see 2.5 for more details). MBT uses models of the system to generate test cases and is used to verify model and actual behavior conformance (i.e., input output conformance). Model-based testing is discussed in detail in Section 2.4.

1.4 Research questions

RQ1 How is the CPS software testing process improved using novel Model-Based Testing (MBT) methods and tools?

The answer to the posed question is sought in Publication I, Publication II, Publication III and Publication V. The publications propose that using an incremental model-based approach improves the quality of tests and reduces the cost of software development by tool supported earlier discovery and fixing of software defects. The model-based testing improvements are discussed in Section 2.4.

RQ2 How to realize the MBT productivity methods?

The productivity and performance questions are explored in Publication I, Publication III and Publication VI. The publications show that by utilizing parallelism and test distribution it is possible to increase the effectiveness and performance of testing. I present the toolkit named TestIt that demonstrates an improvement in these aspects. TestIt toolkit is presented in Chapter 3.

RQ3 How to decrease validation time and design space exploration for complex multi-robot systems operating in dynamic environments?

Multi-robot systems being the object of study in this dissertation, belong to the class of Cyber-Physical Systems (CPS). CPS design validation time and design space exploration are important criteria for development cost optimization. The process of validation and its stages are presented in Publication III. The validation process is described in Section 2.9.

RQ4 How to overcome the main bottleneck of model-based methods, i.e., model construction, namely, how to accelerate the test model construction by automated model learning (based on system logs) and model generation (based on specification)?

One of the primary bottlenecks in model-based validation and verification of autonomous robot systems is the process of model construction. Publication IV and Publication V address this issue by proposing algorithms of model construction from recorded input-output observation logs. Information about SUT model construction is presented in Section 4.5.

RQ5 Can model-based scenario specification improve model-based test quality and reduce test development time?

Test development time is an important factor in CPS testing cost. Test quality is also very important as low quality tests waste resources and can yield incorrect results. These factors are discussed in Publication V. Details about scenario description in Section 4.4.

RQ6 Can the test correctness proof developed for MBT technique be extended to distributed time-critical systems?

The need for proving correctness in distributed time-critical cyber-physical systems arises due to the fact that modern CPS have distributed architecture. Publication VI proposes a technique of deriving and proving the correctness of distributed tests. Provably correct test development is discussed in Section 2.9.2.

1.5 Hypotheses

The solutions of the dissertation's research questions are founded on the hypotheses H1 to H6. These hypothesis will be validated using the theoretical results and by demonstrating the usability of proposed toolkit TestIt.

H1 MBT-based approach provides improvements in testing efficiency and performance (RQ1).

H2 Test process parallelization by using pipelines increases test efficiency (RQ2).

H3 Incremental validation of test development steps improves the quality and trust of test results and decreases the design validation time, design space exploration effort and total cost of CPS development (RQ3).

H4 Automatic model learning and model generation decreases the model construction effort that is a key drawback in applying all model-based techniques (RQ4).

H5 Optimizing test scenarios narrows down design space exploration which, in turn, improves test quality and testing time (RQ5).

H6 Generating distributed tests from centralized remote tests is algorithmically feasible and the correctness of test distribution result can be proved by showing the bisimilarity of the centralized and distributed test models constructed by alternative methods (RQ6).

1.6 Research contributions

The main contribution of the dissertation is the testing framework for autonomous robots featuring a provably correct test development process. In addition to the research contributions presented in the dissertation publications listed below, the dissertation clarifies and adds details about the testing framework for AS.

Contributions in Publication I:

RC1 The proposed testing toolkit initially presented in Publication II is improved with an online test runner. The results are validated on a robotic intruder detection system.

RC2 An adaptive test optimization technique is proposed that takes advantage of the multi-pipeline architecture where testing threads can communicate and coordinate the test runs based on their cooperatively collected test performance data.

Contributions in Publication II:

RC3 The main novelty of presented solution is the scalable multi-pipeline testing architecture that enables incorporation of multi-purpose testing tools including those used in state-of-the-art model-based testing.

Contributions in Publication III:

RC4 The investigation presents an incremental model-based approach for robot farm runtime configuration management and its model-based testing. It has been demonstrated how configuration parameters are generated automatically using Uppaal model checker and the synthesized farm configuration feasibility is checked. Then simulation based verification by introducing low-level operational details and finally, operational correctness in real exploitation conditions is tested against the simulation.

RC5 Results show that the proposed incremental parametric configuration synthesis validation and the model-based generation of autonomous multi-robot system coordination plan minimizes the validation time, design space exploration and cost during early stages of design, rather than during operations which could lead to significant cost.

Contributions in Publication IV:

RC6 Novel algorithm for automatic learning a subclass of Uppaal timed automata (TA) models from the system and its environment interaction logs has been developed.

Contributions in Publication V:

RC7 An automated approach to generating an Uppaal TA model from the topological map that specifies where the robot can move to.

RC8 The specification of interesting scenarios including adding human models to the simulated environment according to a specified scenario.

RC9 Test code coverage measurement shows empirically that it is possible to achieve increased test coverage by specifying simple scenarios on the automatically generated model of the topological map.

Contributions in Publication VI:

RC10 Demonstration of Uppaal TA models and related tool family supporting the development and verification of test models that serve as abstract representations of tests.

RC11 Present the test controllability criteria for remote and distributed testing.

RC12 An algorithm for deriving distributed remote tests from centralized remote testing model is provided to improve the test performance.

RC13 A technique for proving the correctness of derived distributed tests in terms of bisimulation equivalence between the remote and distributed tests is proposed.

1.6.1 Industry application

The testing framework proposed in the dissertation and the TestIt toolkit has been used in the development of an autonomous feeding farm robot in collaboration with the industry partner Norcar AB. TestIt was used in this project to test the autonomous navigation and feeding operations separately. This approach increased the effectiveness of testing by allowing the software development teams to work in parallel and test their respective algorithms without being dependent on the other team after the application programming interface (API) contracts had been specified.

The toolkit has also been used in the industry project "Applied research on system of sensors and software algorithms for safety and driver assistance on remotely operated ground vehicles for off-road applications" [23] in collaboration with the industry partner AS Milrem. In the scope of that project, TestIt was used to test the localization algorithm for the unmanned ground vehicle and hyperparameter optimization for the localization component.

1.7 Structure of the dissertation

The dissertation is structured as follows:

Chapter 2 gives an overview of the background and presents the related work. This chapter introduces the specifics of cyber-physical systems (CPS) testing and robot systems testing. Model-based testing (MBT) as mainstream technique to tackle the complexity and heterogeneity issues of CPS testing is discussed. The CPS MBT is elaborated further by introducing simulation-based testing as an important approach to reduce the testing effort of autonomous mobile robot systems and its benefits are explained. The rationale behind selecting Robot Operating System (ROS) as the default middleware around which the TestIt MBT methodology has been built is presented. Modeling, model-based testing of autonomous multi-robot systems is described. A thorough account of the related tools is given. The background information about Uppaal TA is given and the process of applying

Uppaal TA for conformance testing is detailed. The chapter concludes with the description of the model-based test development process for autonomous multi-robot systems. The steps for test development are shown as well as the full cycle of MBT CPS testing process.

Chapter 3 provides a detailed overview of the testing toolkit TestIt. The chapter begins with the presentation of the design considerations. The architecture of the toolkit is explained next. Finally, the TestIt toolkit functional and non-functional features are presented.

Chapter 4 exemplifies the usability of TestIt toolkit, namely how the toolkit is applied to a practical case study - robotic intruder detection system.

The dissertation ends with the conclusion, reviewing the contributions of the dissertation and outlining the ideas for future work.

2 Preliminaries

2.1 Chapter overview

This chapter gives the fundamentals and an overview of background information related to the work presented in this dissertation. The chapter investigates the research questions RQ1, RQ3 and RQ6.

2.2 Software testing methods

Software testing is defined as "the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component" [13]. This definition states that in order to verify the software correctness the software should be executed to detect defects. In contrast to testing, static analysis is a verification technique in which the software is not executed but its code is inspected and analysed to detect defects instead. While testing is a partial system state space exploration method, static analysis is an exhaustive exploration method. Therefore, static analysis faces severe problems of scalability when applied to a full suite of robot software with many integrated parts and the complex component interactions. Autonomous robotic systems typically comprise multiple actively interacting software and hardware components working under non-trivial timing, computation resource, energy, and algorithmic constraints.

Among several AS quality assurance techniques, such as code inspection, risk assessment and defect detection, Test-Driven Development (TDD) is considered to be one of the most promising trends. In [24], Beck defines TDD as a method of software development driven by automated tests. In TDD, new source code is written only if an automated test has failed or duplication needs to be eliminated. According to [24], TDD promises to manage and reduce uncertainty during programming by incorporating software tests even before writing the source code.

Autonomous robots are inherently cyber-physical systems, meaning that their development and testing techniques can be partitioned based on what component they are targeted to. This dissertation focuses on robot software testing although in integration level testing the borderline between hardware and software becomes hard to distinguish.

The two of the most prevalent testing strategies are white-box and black-box testing [25]. White-box testing (also called logic-driven testing) involves examining the internal structure of the program. This approach allows generating test cases from the program's logic. The weakness of white-box testing is that by definition it requires access to the internal structure of the program and access to all components of a MRS might not be available due to intellectual property rights. Another major concern is the complexity (i.e., the number of parallel components) of MRS which makes white-box integration testing very difficult.

An alternative to white-box testing is black-box testing. Black-box testing has become an important concept to abstract irrelevant implementation details of software components and tackle only with the components' interface behavior. Black-box testing is a testing technique used when the implementation of the system is not known to the tester or the SUT externally observable behavior is verified. The tests are carried out by sending stimuli and observing the behavior of the system. Black-box testing is typically applied to verify whether the system conforms to the specification. This is known as input output conformance (ioco) testing [26]. Black-box ioco technique is a key concept for the testing framework proposed in the dissertation that is focused on testing autonomous robotic systems.

2.3 Cyber-physical systems and autonomous multi-robot systems

2.3.1 Cyber-physical systems testing challenges

Testing robotic systems or any CPS is labour intensive and expensive constituting up to 70 percent of the development costs according to [27]. These high costs are associated with testing due to requiring a physical robot or a CPS to be available for the tester. Physical dedicated testbeds are also necessary to carry out tests on robotic systems in the real world. In addition to that, testing such systems requires significant human expertise and effort which can also be exceedingly expensive ([28], [29]).

Running tests on real implementations is not expensive only because of substantial allocation of time and human resources but also because the access to the SUT may be limited or applying tests may be unsafe to the system itself or its environment. Therefore, simulation-based testing is often used to reduce the cost of testing and the risk of damaging the robotic systems during testing.

A study on robot software defect detection using simulation has been studied in [30]. The authors conclude that many defects do not require the reproduction of complex physical phenomena to be revealed. In fact, the authors reported that only a single defect was not detected out of 21 potentially detectable defects. Using low-fidelity simulation, the authors managed to detect 11 defects.

There are criticisms of simulation-based testing presented in [31]. The paper presents a qualitative study based on interviews with practitioners. The authors find based on the interviews that there is some distrust of simulation even though the participants acknowledged the theoretical benefits of simulation-based testing. The main concerns raised by the participants were the distrust of the simulation accuracy and the validity of the simulated operations.

Another paper [32] analyzes testing robots in virtual worlds using simulators with a presentation of preliminary work for an agricultural robot. The author argues that part of the software validation could use simulation means.

The authors of [33] test autonomous robot software using procedural content generation. The study details an automated approach to testing by generating 500 randomised situations to test. The prototype tool simulated and rated them. The highest rated situations were analyzed in depth which revealed weaknesses in the robot control algorithm.

Automatic testing of self-driving cars with search-based procedural content generation is studied in [34]. The authors combine procedural content generation with search-based testing to create challenging scenarios for testing self-driving car software. The presented tool was not only more efficient, but also caused up to twice as many lane departures.

Experience of autonomous transportation systems [35] has proven that simulation is often drastically more cost-effective than testing such complex robotic systems in the real world. Simulation is also considered as the primary means to scale the tests by using more computational resources instead of requiring more physical robots and testbeds. Simulation-based testing is also encouraged in [36] as physical real-world deployment is often too late in the development stage to start testing. This can result in significant savings in product development.

In order to verify that the software behaves correctly it must be subjected to different input data. This input data can be generated randomly (i.e., random testing) or by following certain test scenarios. In random testing, the programs are tested by randomly generated independent inputs. The output is compared to software specifications to verify whether the test passed or failed. This testing method can be improved upon by using a model of the system to generate the input data more systematically. Using a model of the system and its environment allows generating test sequences rather than mere isolated test cases. This approach is called model-based testing (MBT) and this approach is often applied in combination with scenario-based testing. In scenario-based testing, the test case is defined as a concrete scenario that the SUT has to perform (e.g., a robot has to move to a location, perform a task and return to the starting position). Such scenarios can be designed manually but in order to minimize manual labour, it is desirable to automate scenario generation using models. The scenario-based testing has been shown to be more effective than random testing and other testing approaches ([37], [38], [28], [29]).

According to [39], the relative cost of fixing software defects grows immensely from design phase to maintenance phase. The authors of [39] claim that a defect fixed at implementation phase is 6.5 times more expensive to fix than a defect fixed in the design phase. Fixing a defect at the acceptance testing phase increases the cost to 15 times. This cost grows to 100 times if the defect would be fixed at the maintenance phase. Such a large cost increase motivates finding software defects at the earliest possible phase. Using model-based testing approach supports early detection of defects since it is possible to start testing early as model development can occur in parallel with software development rather than after development has been finished. In test-driven development the test design can start even earlier, as soon as requirements specification is made available, i.e., even before software implementation.

Last but not least, model-based approaches reduce the complexity of specifications by removing unnecessary details and focusing on more significant parts of the system description. These are the main motivators why in complex software systems development processes the use of model-based methods become more extensively accepted ([40], [41], [42]). Addressing the challenges of autonomous multi-robot system quality assurance issues using model-based methods, in particular, those that need advancing automated test and verification methods and tools is the main topic of this dissertation. The general considerations discussed above lead to concrete research questions outlined in Section 1.4.

Cyber-Physical Systems (CPS) have been defined in recent literature in [43] as systems of collaborating computational entities which are in intensive connection with the surrounding physical world and its on-going processes, providing and using, at the same time, data-accessing and data-processing services available on the internet. In other words, CPS can be generally characterized as “physical and engineered systems whose operations are monitored, controlled, coordinated, and integrated by a computing and communicating core”.

Testing cyber-physical systems is more challenging compared to pure software testing due to several factors discussed below.

The challenges of CPS, including robots and AS, testing have been mapped and analyzed by multiple authors ([31], [44], [19], [45], [46], [47], [48]). Here, just an extract of them are presented to exemplify the growing interest towards the topic. In [31], robotic systems testing issues have been examined and categorized in three major themes: real-world complexities, community and standards, and component integration. The authors call attention to the fact that very little of the work on testing takes into account the physical aspects of the problem (i.e., abstraction of the environment is exceedingly difficult and testing in physical environments requires more resources). The second theme brought out the issue of diversity among developers and the lack of standards in testing robotic systems. The final factor examined in [31] showed that testing integrated hardware and software is highly complicated since this integration increases the complexity of the system, the cost associated with testing and introduces complications when defining test oracles.

Further, the issues of testing specifically autonomy related functionalities have been discussed in [44] and [19]. Here, alongside with showing major challenges that appear with testing autonomy and highly dynamic nature of systems behavior, prospectives on how to apply MBT combined with simulations have been suggested by the authors. In the following, some of the challenges are discussed.

Cyber-physical systems are systems of systems due to the different technologies used in CPS integrating on multiple scales. In [49], CPS are defined as integrations of computation and physical processes. This means there are usually several hardware components with each having software layers. Numerous interconnected components make testing complex. Testing CPS requires focusing on specific aspects and layers separately and together to verify that the integration of components is working correctly.

In addition to architectural integration complexity, there is a high level of concurrency with complex computations and interactions where timing and physical location are critical. Timing issues are challenging in testing as the timing deadlines (hard, firm and soft RT systems) can affect the SUT behavior and have been studied in Publication VI. Timing correctness is an important aspect to be tested and verified under the various load, security and safety constraints. CPS quality assurance practice relies on integration and system level testing in addition to relevant to specific components formal design and verification methods. This is because, in CPS, there are many levels of integration (hardware and software). Non-determinism problems stemming from communication latency and race conditions show up sporadically in the course of system execution making them very hard to predict, find and reproduce. CPS testing challenges are covered in more detail in Section 2.5.

CPS testing usually mandates using black-box testing method because testing all levels of technology integrations as white-box testing becomes prohibitively expensive and complex. Black-box testing meshes well with model-based testing since it is based on the specification of the system. Model-based testing checks the conformance of the SUT against an abstract behavioral model. Abstract tests hide irrelevant implementation details, allow automatic design and execution of tests, provide systematic coverage and measure coverage of model and requirements. The drawback of model-based testing is the modeling overhead. This overhead can be substantial in large-scale projects but for CPS operating in proximity to humans the benefits of MBT can easily outweigh the overhead costs [50].

2.3.2 Autonomous multi-robot systems and system validation

The terms group behavior robotics [51], collective behavior robotics [52], cooperative behavior robotics [53], swarm robotics [54] and multi-robot systems [55] have been used rather interchangeably by researchers over several decades to refer to the same phenomenon. Authors of a review of research in multi-robot systems [9] offer the definition of multi-robot systems as a collection of two or more autonomous mobile robots working together to achieve some well defined goals. Multiple robots instead of a single robot add additional complexities to motion coordination, communication, object manipulation, reconfiguration, task planning, control, localization, mapping, exploration and learning. All the listed areas become more intertwined in heterogeneous multi-robot systems (MRS) which in turn makes their validation exceedingly difficult. The autonomy related functionality (self-learning, strategy adaptation), as already pointed out above, adds even more complexity to the system design and testing challenge. In order to reduce the complexity of designing and testing such systems, it is possible to employ robot software middlewares to simplify the multi-component design complexity. This will be discussed in the following subsection.

2.3.3 Robot software development

Software paradigms are varied as described in [56] but, in essence, software can be categorized as having a single large monolithic architecture or contrastingly as being composed of multiple smaller interconnected modules. Both general approaches have their strengths and weaknesses. According to [57], the monolithic approach has the benefit of being easier to debug and test. Monolithic software can also be simpler to develop as there are less cross-cutting concerns that affect the whole application. In addition, an analysis on monolithic architecture performance [58] showed that it is possible to achieve better memory performance compared to multi-component (microservice) architecture. Also, a monolithic architecture does not require an extra data distribution layer for communication between its components since it is a single monolithic block. On the other hand, monolithic architecture creates a single point of failure since all of the components are packaged into a single application. If this monolithic component fails, the robot system would fail and would need to be restarted. Such complete failures can be catastrophic in robotic systems (e.g., fast moving mobile robots). Recovery time from failure is also longer in monoliths since the component itself is larger and takes longer to restart.

Alternatively, the robot software can be developed as multiple smaller components working in parallel as distributed applications [56]. This approach has certain distinct advantages over a monolithic architecture. Firstly, structuring the robot software as a set of small components more readily supports fault-tolerant design. Fault-tolerant design allows the system to continue operation with degraded performance rather than completely failing [59]. If the robot software is designed as modules the non-critical components can fail while the robot might still retain some essential functionality to fail safely. This is not the case with monolithic architecture as a critical failure (e.g., a memory access violation causing a segmentation fault) would result in complete robot failure. The second key advantage is the fact that clear partitioning into smaller components allows simultaneous work on different components by development teams with different competences in parallel [60]. Finally, smaller components also promote code reuse which allows the development team to focus on innovation [61]. These advantages are part of the reasoning why autonomous robots are often composed of multiple software components rather than a massive monolithic block.

Developing modular software for autonomous robots is considerably less complicated when the software takes advantage of some existing solutions that provide functionality for inter-process data exchange. This functionality is provided by software called middleware.

A comparison study between robot middlewares in [62] covers ROS, RT-Middleware, OPRoS and Orocos as being the most popular and widely adopted platforms for robot software development. The authors conclude that ROS having the biggest ecosystem of users and components can be the best choice for a robot middleware but other options may have some specific advantages depending on the requirements for the concrete use case.

According to [63], ROS is an open-source, meta-operating system for robots. It provides the services commonly expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

The TestIt toolkit presented in this dissertation is generally robot software implementation agnostic as it can be augmented with additional specific middleware support but support for ROS based robots has been developed as the default supported middleware. ROS [64] has been selected as the initial support target due to its popularity both in academia and industry.

Another comprehensive literature survey [65] gives also a detailed overview of a large number of middlewares. There are many options for robot middleware that makes the informed choice complex and requires in-depth analysis. A selection of features [66] provided by ROS middleware are highlighted as follows:

Popularity. The yearly report on ROS [67] stresses a steady growth of documented ROS robots which has grown to over a hundred robots. This list includes both experimental and commercially available robot platforms. Additionally, the number of papers citing the original ROS publication has increased 22% [67] over the previous year. These factors indicate that the growth trend continues for the ROS community.

Hardware support. An often overlooked aspect for middleware choice is hardware support. Numerous sensors have ROS hardware drivers readily available for use which makes their integration to the robot platform much less time consuming and allows the development team to focus on other matters [66]. Thus, also the number of ROS-supported hardware components is a strong argument for selecting ROS as the primary support target for the testing toolkit TestIt.

Data sharing. The key feature that is required from a robot software development standpoint is data sharing between software components. ROS offers a message passing interface providing inter-process communication [66]. ROS has a built-in and well-tested messaging system that saves development time substantially by managing the details of communication between components. This is achieved by the anonymous publish and subscribe mechanism. Clear communication interfaces improve component encapsulation and enables better scalability. Data transmission is peer-to-peer which minimises the overhead.

Message recording and playback. Another key feature that advocates for using ROS is the message recording and playback [66]. The message recording is naturally supported due to the publish and subscribe system which is anonymous and asynchronous. Data can easily be captured and replayed without changes to software components. This improves the complex task of robot software debugging as it allows the developers to reproduce defects from recorded data more easily.

Synchronous interprocess communication. ROS features synchronous request and response interactions between processes [66]. This feature is useful in situations when software components need to communicate synchronously. Using synchronous communication is achieved by using ROS services which enable guaranteed but blocking responses to requests.

Global data sharing. ROS provides a way for software components to share information through a global key-value store [66]. This allows the components to easily modify task settings and even allows to reconfigure these settings dynamically by other components.

Since TestIt toolkit presented in the dissertation aims to improve testing workflow for as many robot software engineers as possible, it has made the choice in favor of ROS as the default middleware a natural decision.

A recent study [68] surveyed the architectures of ROS-based systems. This study is founded on analysis of numerous repositories and provided evidence-based guidelines for architecting robot software. The study details that ROS-based systems are predominantly composed of numerous components working in parallel. According to [69], the number of parallel interdependent components can easily reach hundreds of individual components. This motivates the need for advanced integration testing tools which the toolkit proposed in this dissertation aims to satisfy.

2.4 Model-based testing and modeling

In [70], model-based testing is defined via four main approaches:

1. Generation of test input data from a domain model.
2. Generation of test cases from an environment model.
3. Generation of test cases with oracles from a behavior model.
4. Generation of test scripts from abstract tests.

Each of these approaches has a different meaning. The first approach generates test input from a specific domain model but does not specify how to determine whether to consider the test to pass or fail. The second approach uses a model of the expected environment of the SUT. Such an approach allows automated generation of the environment variables for testing the SUT (e.g., operation frequencies, data value distributions) but also lacks the possibility to easily determine whether a test passed or not. The third approach uses behavior models of the SUT to generate test cases for the SUT but also declare the expected result of each test case. This approach is far more complex than simply generating SUT inputs without checking the actual output values of the SUT. The fourth approach is different as the aim is to generate concrete test cases from concise abstract description of test cases. The approach referred to as MBT in this dissertation is the third approach. Taking this into account, MBT can be defined as the automation of the design of black-box tests [70].

The main motivation behind using model-based testing is that it aims to reduce the cost of testing, enables early detection of requirements defects, allowing traceability and generating less overlap among tests. The respondents in a MBT survey [71] reported that thanks to MBT, test design becomes more efficient, testing becomes more effective, models help manage the complexity of the system with respect to testing, models improve communication between stakeholders and help start testing design earlier.

The criticisms of MBT in modern software development processes are listed in [72]. The report outlines slow modeling speed, competence development issues, concerns about return on investment and about complexity along with expense of model-based tools as main criticisms of MBT. According to [72], there is a negative attitude towards modeling after the demise of the waterfall software development life cycle (see Section 2.9 for more information) which is making some developers see modeling as a chore and something that does not provide value. Manual modeling is time consuming and prone to errors. Modeling usually requires background knowledge about formal methods and modeling which can also be an issue. Due to these factors, MBT has still not reached ubiquitous adoption and is still mostly used for only critical components (e.g., aerospace and space technology). These concerns must be taken into account when developing new MBT tools if the tool is to be adopted by the community.

The advantages of model-based testing get the best exposure while the models are concise and they are created at a suitable abstraction level for the desired test scope. Different granularity models are required for verification of the system at different levels and the chosen abstraction level can have a significant impact on the scalability of verification [73]. If the model is too detailed it will contain superfluous information and details that increase the complexity of the model. Conversely, models with too little information are also not very useful for generating interesting test scenarios. Therefore, it is important to determine the relevant abstraction level for the model used in MBT.

Generally, the model should only include the operations that need to be tested to avoid the state explosion problem [74]. It is inadvisable to add data fields that are not required for testing the operations. To keep the model concise, complex data fields can be simplified with an enumeration or generated by an external algorithm that does not necessarily have to be included in the model itself [73]. The operations themselves can be modeled at an abstraction level that is not exactly the same as it is in the SUT (i.e., not all states and transitions are listed explicitly). This is especially true in case of automatically learned models where only behavior observed during learning is represented in the model. As a result of automatic model learning, the model can not include information the system did not exhibit during learning ([75], [76]).

Another aspect of models' relevance is their expressive power. There are multiple notations for modeling the SUT. Different notations have different strengths and weaknesses. Notations have been grouped in [77] into the following classes: history-based specification, state-based specification, transition-based specification, functional specification and operational specification.

Besides expressive power, the modeling formalisms are most often categorized by the different aspects they are focused on, such as scope, characteristics, paradigm, test selection criteria, technology and whether they are designed for online or offline use [78]. Though de facto standard modeling languages such as UML, SysML, and their numerous profiles have been widely used in the software industry due to their intuitive nature and mature tool support they have not been designed originally specifically for robotics [79].

The formalism used in this dissertation is a state machine based formalism called Uppaal timed automata (TA) [80]. This formalism has some distinct benefits over some other notations for modeling and verification of CPS as outlined in [81]. Uppaal TA will be covered in more detail in Section 2.7.

Model-based testing has been mapped into different categories in [78]. The taxonomy is created based on different aspects of the MBT process. One of the key aspects is whether the tests can be executed online or offline. The toolkit presented in this dissertation supports online testing which is desirable for testing autonomous systems because of inherently non-deterministic nature of interaction between the robots and their environment.

2.5 Model-based testing of autonomous multi-robot systems

As MRS are instances of CPS, CPS testing steps described in [82] are generally also applicable to MRS. According to [82], the CPS testing is divided into the following levels:

1. Hardware components testing, including testing of component functionality w.r.t. system requirements.

2. Structural and computation testing with focus on the design and structure of the program.

3. Extra-functional properties testing with non-functional requirements correctness testing (e.g., temperature, power consumption).

4. Network testing for testing communication flow among multiple devices and users.

5. Integration testing of individual software modules.

6. System level testing of the full system composed of both hardware and software.

As the TestIt toolkit presented in this dissertation is primarily focused on integration testing of software, only the integration and system-level testing efforts will be covered in more detail. The authors of [82] direct their attention to several efforts by researchers targeted at integration and system testing in order to verify CPS. Notably, they point out [83] as an example of system level testing approach of a CPS. In this paper, the authors investigate an automotive development platform with limited source code access and the test interface is only using controller area network (CAN) bus messages. They report successful monitoring of a hardware-in-the-loop vehicle simulator and analysis of the prototype vehicle log data to detect violation of high-level critical properties.

The authors of [82] conclude that little research has focused on testing and validation for CPS where suitable testing method can arguably have profound impact in preventing costly and possibly fatal system failures. The authors find that using model-based testing has potential and should be explored further.

Using MBT for testing CPS specifically was covered in [84]. In this publication, the author used data-driven approaches to obtain values of features and variables from a passenger lift CPS. However, this case study explores a relatively small example where complexity issues are not significantly exposed. The sensors used in the process only provided low-level system information such as acceleration and air pressure.

MBT efficiency has been studied in several papers and the results have shown significant improvement in testing time and efficiency ([37], [38], [28], [29]). Timed automata and Belief-Desire-Intention (BDI) automata were compared to pseudorandom testing in [37]. The authors measure code coverage which is defined as the number of lines executed as a result of stimuli sent by running the model. The goal is to achieve the largest code coverage in the shortest amount of time. The results of the study showed a significant improvement in code coverage accumulation speed for model-based approaches over the basic pseudorandom method.

Another paper [85] investigated the BDI agent model-based approach further in the domain of human-robot interaction (HRI). Their paper covers three test case generation methods (manual, pseudorandom and reinforcement learning based). The authors demonstrate the results using simulation. The paper concludes that using models for test generation clearly outperforms existing approaches in terms of coverage, test diversity and the level of automation that can be achieved. The motivation for using simulation for testing software comes from the fact that large-scale physical test in the real world in the early stage of testing is unfeasible due to costs and time. A recent study focusing on autonomous driving [86] found that to fully demonstrate the vehicle reliability in terms of fatalities and injuries the vehicle would have to be driven hundreds of millions of kilometres and in some cases even hundreds of billions of kilometres. This is clearly not feasible as it would take the existing autonomous cars fleets tens and sometimes hundreds of years to drive such vast distances.

Grieskamp et al. [38] applied MBT methods to Microsoft products and reported average testing effort per requirement to be 1.39 person-days while traditional testing required 2.37 person-days (41% improvement in testing time).

Large-scale technology evaluation study [28] of 13 industry cases from the transportation domain shows verification and validation cost reduction between 29% and 34%. This study was conducted over a three-year time period and featured projects from the transportation domain (automotive, avionics, rail system). In addition to the improvement in verification and validation, the study reports less significant improvements regarding test coverage (8%), number of remaining defects (13%) and time to market (8%).

Felderer et al. [29] studied defect taxonomies (ESA multi-mission user services infrastructure testing) and their return on investment depending on several parameters like the average test design time or the number test cycles and experience values of a test organization. The authors find that MBT requires a potentially larger resource investment in the beginning of the project compared to traditional methods. But after the initial investment, the incremental effort required for testing using MBT is drastically lower compared to manual testing.

These results lead to the conclusion that applying MBT methods to CPS brings significant improvements over traditional testing methods.

2.6 Related tools

As detailed in Publication I and Publication II there are other frameworks and tools that are targeted to robot software testing in different capacities and model-based testing in general. Each of the analysed tools has had some limitations which reduce its usability, effectiveness, focus or scope compared to the toolkit presented in this dissertation.

A generic testing framework for TDD of robotic systems has been presented in [87]. This framework provides functionalities for developing and running unit tests in a language and middleware independent manner by allowing users to use independent plugins in their code. The framework does not feature integrated model-based testing support and is targeted towards unit test level testing.

One of the first attempts to provide a unified, tool-supported methodology for CPS testing and optimization is presented in [88] where the authors consider a black-box approach to perform test input sequence optimization by testing the input-output behavior of the CPS. They claim their tool is the first CPS testing tool that supports Bayesian optimization. It is also the first attempt to employ fully automated dimensionality reduction techniques for CPS testing.

Using simulation in testing has been proposed in [89] where the authors presented a test harness that allows initialization of the simulation in specified conditions based on previous computation. However, this harness is not specifically designed to support ROS-based robots nor does it aim to aggregate and automate the debugging process for the accidents the authors managed to detect using their proposed approach.

Another approach is proposed in [90]. The authors focus mainly on testing autonomous vehicles (AVs) by introducing a risk-based framework. It uses the cross-entropy algorithm to produce so called rare events which have considerably higher probabilities to lead to an AV crash. The generative model for such behavior of the vehicles is trained using imitation learning based on the public traffic data collected by the US Department of Transportation. The approach described in the paper does not utilize model-based testing methods but relies on the rare-event probability evaluation of the data to generate specific highway scenarios.

The concept of utilizing simulation in a massively parallel configuration has been used before for example in [91]. In the publication, the authors describe how they used Apache Spark and ROS bag files to train deep-learning models for autonomous driving. The presented architecture is used for model training and not for discovering scenarios and optimizing tests.

Using simulation to uncover software defects has been shown successfully in [92]. The authors propose a high-level framework for automatically testing robotic systems.

Robot system specific testing tools have more narrow focus and most of them are adjusted to ROS-based software testing. At present, there is no known significant effort made in the development of model-based testing toolkits designed specifically for autonomous multi-robot systems and their application.

There are several general purpose testing tools that allow test parallelization but they do not employ the model-based testing of ROS-based autonomous robots. An example of such general purpose testing tool is a continuous integration (CI) platform Testributor [93].

Testributor is an open-source continuous integration platform that reduces building times by slicing up the test suite and runs the slices in parallel. This platform is not specifically designed for testing ROS software components and it does not support model-based testing natively.

An alternative to Testributor is a ROS specific automated test framework (ATF)[94] which has been developed specifically for ROS applications. ATF framework supports executing integration and system tests and running benchmarks. Unfortunately, it is not readily scalable and is designed to run on a single machine. ATF framework also only provides the execution of the test suite but does not offer tools to create or optimize the test suite itself.

In addition to test execution tools outlined above, numerous general purpose model-based testing tools have been developed which focus on the testing process itself without providing the required supporting functions such as infrastructure allocation, environment and test suite launching and test results aggregation.

The following list is limited to recently updated projects with open source or a free academic license. Comparison to Uppaal TA is provided for pertinent tools. This analysis was taken into account when selecting the initial formalism to include in the toolkit presented in this dissertation.

GraphWalker [95] is an open-source tool based on finite state machines. It offers model verification by running test path generations (offline and online path generation as REST or a WebSocket service). GraphWalker is essentially a test generation tool but it lacks the support for testing timing constraints which is important in testing time-critical systems. Another important limitation is the absence of test execution support which means that the tool does not interact with the SUT itself and requires additional effort to integrate the tool to the test framework. Finally, GraphWalker does not include a model checker and hence does not support test model verification.

Free Model-Based Testing (fMBT) [96] is another open-source tool which generates test cases from models written in the AAL/Python pre/postcondition language. fMBT provides a set of tools for test generation and execution. This tool is similar to GraphWalker in the sense that it generates tests based on the model but it also lacks the notion of time that is present in Uppaal TA and lacks model checking capability.

The tool named 4Test [97] is a commercial tool with limited free version which uses a combinatorial approach called constraint driven testing to select test cases from textual models specified in a syntax inspired by the Gherkin language. The main demonstration use cases for this tool are website testing use cases and the tests do not include time as opposed to Uppaal TA which is a limiting factor for CPS testing.

JSXM [98], [99] is a model animation and test generation tool. This tool uses a special kind of extended finite-state machine (EFSM) called Stream X-machines (SXM) as its underlying data structure. The tool supports generation of concrete test cases from abstract test cases from SXM models in the implementation language of the SUT. The main area of use for JSXM is web services. This tool lacks timed automata support like the previous tools.

The tool Modbat [100] is a model-based testing tool that is based on EFSMs. Modbat is specialized to testing the application programming interfaces (API) of software. This tool provides a domain-specific modeling language with features for probabilistic and non-deterministic transitions, component models with inheritance, and exceptions. Modbat suffers from lack of model checking and timed automata support.

MoMut [101] is a free for academic-use family of automated, model-based test case generation tools. The tools can work with different inputs: UML state charts, action systems, timed automata and assume-guarantee contracts. It features a fault-based test case generation strategy using mutation operators. Compared to Uppaal TA, this tool does not support model checking and lacks concurrency support.

OSMO model-based testing tool [102], [103] is an open-source test case generation and execution tool. The test models are expressed as Java programs which the test generator executes based on annotations defined in the model. This tool does not support model concurrency and model checking.

Tcases[104] is an open-source model-based test case generator. Tcases is a combinatorial testing tool that can generate n-wise (generalized pair-wise) or randomized test suites. This tool does not support model concurrency and model checking.

TorXakis [105] is an open-source model-based testing tool that generates test cases based on the composition of process instances that model the behavior of the SUT. TorXakis uses the ioco theory to check the actual behavior against the specified externally observable behavior. This tool lacks model checking and timed automata support.

There is also a cluster of other tools that provide model checking functionality (e.g., CADP [106] [107], DREAM [108], Romeo [109]) but they lack other properties such as timing constraints support, native test execution support, have limited monitoring, graphical specification or lack counterexample visualization support.

Based on the analysis of related tools and the formalisms they are based upon, Uppaal family and Uppaal TA was selected as the initial tool and formalism to be supported in the toolkit presented in this dissertation. More information about Uppaal TA are presented in Section 2.7 and its application in CPS conformance testing in Section 2.8.

2.7 Uppaal timed automata

In order to perform model-based testing, the SUT must first be modeled using some formalism. Testing robotic systems is well suited for modeling notation enabling modeling of both inputs and outputs to and from the SUT (i.e., the stimuli and the response of the robot). In addition, the formalism needs to have tool support (e.g, online execution) to be practically usable for automated testing of robotic systems. These constraints eliminate formalisms such as Markov chains [110] and Event-Flow graphs [111] as these notations are more relevant for environment specification. Event-B [112], Finite state machines [113] and State charts [114] all have tool support but in their original forms lack another important aspect for robotic system testing, namely, the notion of time. Timed Petri Net and Timed Automata are formalisms with the support of time. The expressiveness and properties of Timed Automata and timed extensions of Petri Nets are compared in [115]. The author of [115] brings attention to the fact that for Timed Petri Nets reachability is not decidable which can be a concern in verification of MRS. Translation between variants of these two formalisms have been shown to be possible as shown in [115] giving indication that both formalisms could be used via automatic translation under some conditions. According to the analysis in [115], Timed Automata are a suitable formalism for modelling of systems of industrial sizes and the tool support is sufficiently mature. The analysis in [115] also concludes that Timed Petri Nets are too expressive and are therefore unsuitable for automatic verification. Considering these aspects, Timed Automata was selected as the initial formalism to be supported in the toolkit presented in this dissertation. Timed Automata formalism is supported by the tool Uppaal TA tool family.

According to [116], a timed automaton is a finite-state machine extended with clock variables. The automaton uses a dense-time model. As outlined in Publication II, Uppaal TA are defined as a closed network of timed automata extended with integer variables, structured data types, and channel synchronisation [80]. These timed automata are combined into a network by parallel composition. An automaton consists of vertices called locations (similar to vertices in graph notation) and edges (directed arcs in graph notation) between the locations.

The set of variables associated with an automaton have valuations that are called states and the configuration of model consists of current control location of each automaton and assignments to all model data variables and clocks [80]. The automata in the network can be synchronized using synchronization links named channels between edges [80]. In this work, the channel names follow naming convention due to their specific use in test models. The channels that prefixes i and o in their names are used for sending commands to the SUT and receiving feedback from the SUT respectively.

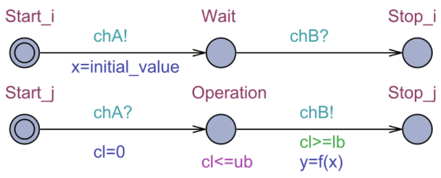


Figure 1: Uppaal sample.

An example of Uppaal TA with two simple automata template instances called $Process1$ and $Process2$, respectively, and composed in parallel is depicted in Figure 1. Both processes start from their initial location (similar to vertex in graph notation) $Start\_i$ and $Start\_j$, respectively. At its first move from location $Start\_i$ to $Wait$, the $Process1$ synchronizes with $Process2$ via channel chA that labels edge ($Start\_j$, $Operation$). At the same time, $Process1$ updates variable x with the value of constant $initial\_value$ and $Process2$ resets the clock cl. After reaching location $Operation$ $Process2$ waits maximum ub time units, i.e. till the location invariant $cl <= ub$ holds. Next edge ($Operation$, $Stop\_j$) can be fired earliest after lb time units (by clock cl) counted from arriving to the location $Operation$. This is specified in guard condition $cl >= lb$. A guard is an expression which evaluates to a boolean (and other conditions described in [80]). Firing edge ($Operation$, $Post2$) is synchronized again with edge ($Wait$, $Stop\_i$) in $Process1$. An edge is said to fire when it is executed which, in turn, leads the automaton to a new state. When firing edge ($Operation$, $Stop\_j$) the variable y is updated with the value of function f where variable x is an argument. Both automata terminate at the same time in locations $Stop\_i$ and $Stop\_j$ respectively.

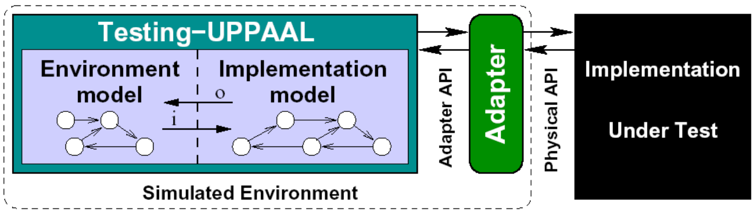


Figure 2: Uppaal test adapter. [117]

As detailed in Publication V, mapping the symbolic inputs and outputs [118] generated by test model to actual test inputs and outputs requires a test adapter that connects the Uppaal TA model to the robot (either real or simulated). Symbolic inputs and outputs in the model essentially denote sets of possible values that real test inputs and outputs can have. The test configuration is shown in Figure 2. The adapter connects to the SUT using Dtron which in turn uses Spread Toolkit [119] and Google Protocol Buffers. More details about the adapter and test configuration is available in Publication V.

2.8 Applying Uppaal TA for CPS conformance testing

Conformance testing aims to determine whether the implementation of the system conforms to its specification. The notion of conformance in the context of model-based testing of labeled transition systems was introduced by Tretmans [26] as the input output conformance (ioco) relation. Provided the input to SUT and its specification are same, the test results can be evaluated by comparing the outputs of the implementation with those of the specification. In case the implementation gives an output not expected by the specification the test verdict is $fail$. Upon outputting the expected values for the given input sequence, the test is declared $pass$.

Conformance testing is especially suitable to employ when designing software and hardware simultaneously since these processes can be worked on in parallel. The teams working on software and hardware can focus on developing the product that fulfills the requirements without spending resources on developing extraneous test interfaces. Conformance testing can start immediately when both teams have finished developing their respective parts of the product.

Uppaal TA has proved to be usable for CPS conformance testing [120]. Among other tools Uppaal tool suite contains Uppaal TRON [121] which offers the possibility of performing on-line conformance testing of real-time systems with respect to timed input-output automata. Uppaal TRON implements a sound and (theoretically) complete randomized testing algorithm, and uses a formally defined notion of correctness to assign verdicts. Uppaal TRON uses the Uppaal TA model checking engine to create symbolic timed traces of the model [122]. The reachable symbolic states are computed from each preceding state and then an enabled transition (and related to that input or output) is selected randomly. Upon reaching a final state the test ends. The test is terminated when the test duration expires or a violation of conformance between implementation and model is encountered.

In the following, the definition of ioco-relation and its extension rtioco (Eq. 4) for real-time systems are given formally using symbolic timed traces. Symbolic timed trace $TTrS$ of an Uppaal TA model is a possibly infinite sequence of symbolic states, each state defined as a tuple $(\bar{l}, D, \bar{v})$, where $\bar{l}$ is a locations vector, D is the set of clock constraints (zone) [123] and $\bar{v}$ a vector of non clock variable values. In operational semantics it is shown that the transition (Eq. 1) from a symbolic state to another can be either an action (a_i) or a delay (δ_i).

$$(\bar{l}_i, D_i, \bar{v}_i) \xrightarrow{a_i/\delta_i} (\bar{l}_j, D_j, \bar{v}_j) \tag{1}$$

Uppaal TA actions are either I/O events e or internal state updates (assignments to variables of V). After each state transition, either an event e or update of variables V or both occurs. The status of transitions (i.e., enabled or disabled) in a given state is evaluated during model execution by Uppaal Tron [117]. The state transition is enabled in case the guard condition of that transition evaluates to true.

The Uppaal TA test model executed by Uppal test execution tool Tron is assumed to have two parallel partitions $\mathcal{E}$ and $\mathcal{S}$ that represent the observable behavior between the tester and the SUT. The partition $\mathcal{E}$ models the environment and $\mathcal{S}$ models the SUT. The interaction between these partitions is achieved via input (A_I) and output (A_O) observable actions. Input actions, controllable by test, are used as stimuli to the SUT and the output actions are used for deciding on conformance. In addition, both $\mathcal{E}$ and $\mathcal{S}$ have internal actions ε which are confined to each individual partition which are used to evolve the partition to the next state where an observable action can be taken.

The observable actions are triggered based on a testing event e after an observable delay $\Delta \in \mathbb{R}^{\geq 0}$. This delay represents an internal delay for the events at an abstract level. Global variables in Uppaal TA (i.e., a vector of externally visible global variables) v are also observable. The events and variables are partitioned into three disjoint sets of input events/variables Ev_{in}/V_{in}, output events/variables Ev_{out}/V_{out} and internal events/variables Ev_{int}/V_{int} [124].

Consequently, after partitioning a symbolic trace can be rewritten as a timed I/O trace. A trace of a state s (Eq. 2) is a possibly infinite sequence of observations starting from a given state. Each observation is a tuple (e, D, v) consisting of an event $e \in Ev_{in}$ ($e \in Ev_{out}$), a clock zone D and a vector $v \in V_{in}$ ($v \in V_{out}$) containing the values of data variables that are externally visible as inputs/outputs at the time of event e.

$$ttr_{i/o}(s) = (e_0, D_0, v_0)(e_1, D_1, v_1)...(e_i, D_i, v_i)... \tag{2}$$

Uppaal Tron is using the observable events to interact with the SUT. The internal actions τ and internal delays d are represented as observable delays Δ. As a result, the test session is recorded as a finite sequence of events T_{seq} in the following form in [125]:

$$T_{seq} = (e_0, (\tau_0 + d_0), \bar{v}_0), (e_1, (\tau_1 + d_1), \bar{v}_1), ..., (e_n, (\tau_n + d_n), \bar{v}_n) \tag{3}$$

The trace in the form of Eq. 3 can be written in terms of observable delays and actions as:

$$T_{seq} = (e_0, \Delta_0, \bar{v}_0), (e_1, \Delta_1, \bar{v}_1), ..., (e_n, \Delta_n, \bar{v}_n) \tag{4}$$

where Eq. 4 allows checking of timed conformance of the SUT against the specification via the $rtioco$ relation by allowing the SUT to refine the timing behavior of the specification [125].

The relativized timed input/output conformance ($rtioco$) is defined in [80] as follows:

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

$$\mathcal{I}\ rtioco\ \mathcal{S}\ iff\ \forall \sigma \in TTr_i(\mathcal{E}) : TTr_o((\mathcal{I}, \mathcal{E}), \sigma) \sqsubseteq Tr_o((\mathcal{S}, \mathcal{E}), \sigma) \tag{5}$$

Such a sequence of test events is provided by Uppaal Tron. Each test event is specified in terms of clock constraints, variable valuation, list of next available states and a list of input/output actions. After each 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 states is updated [125].

2.9 Model-based test development process for autonomous multi-robot systems

2.9.1 Model-based test development

CPS development is somewhat different to traditional software development as it is more interdisciplinary. The standard software development processes still apply but as CPS usually contain electrical and mechanical components in addition to software, there is a need for increased interdisciplinary collaboration support. This collaboration can be supported by the model-based approach. Inter-team communication is challenging and can be made more clear by using unambiguous mediums such as models. These models can be created at various levels of abstraction depending on the purpose of the model.

The iterative and incremental development cycle [126] suggested for autonomous systems in [127] is a software development life cycle (SDLC) which is loosely based on the traditional waterfall [128]. The waterfall model divides development into well defined stages which are completed sequentially. The process consists of requirements specification, design, implementation, test, installation and maintenance phases. Such a development cycle supports clear understanding of the full project to all stakeholders. Unfortunately, that development cycle is very hard to implement in practice as requirements tend to change over the course of the project development life cycle. Another serious concern lies in the fact that the testing phase occurs at the end of the development cycle. If the software fails to satisfy the requirements at this late stage, invariably a substantial redesign of the software is required.

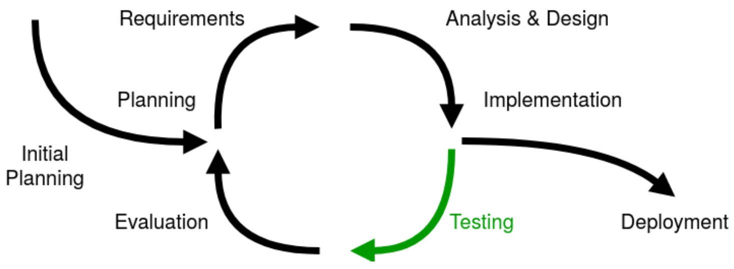


Figure 3: CPS iterative and incremental development.

The iterative and incremental development cycle aims to correct the aforementioned shortcomings. The general iterative development approach is depicted in Figure 3. The cycle is divided into discrete increments which are developed from the requirements to deployment phase before starting a new iteration. The first cycle begins with the initial planning phase during which a general overview of the whole project is developed. Once the general plan has been formed, the cyclical development process can start. Testing is performed at every iteration which reduces uncertainty about requirements satisfaction.

To overcome the limitations of the traditional waterfall, in Publication III, it is proposed to follow a novel combination of iterative and incremental SDLC [126] that is complemented with verifiable MBT steps. The novelty of this approach lies in the iterative development process that refines the abstraction incrementally by iterating model synthesis, model checking, simulation-based validation and testing.

Each cycle starts again with the planning phase. The iterative planning phase serves to concretize the requirements for the implementation. Next, after the planning phase, comes the implementation phase which takes up a large portion of the full cycle time as it also includes the analysis and design of the system increment that is being developed in that iteration. After the implementation is finished, the implementation must be tested. This phase is often given disproportionate resources compared to the other phases. As pointed out in Section 1.3, adequate testing is crucial for the overall success of the project. Inadequate testing can lead to increased technical debt as well as system defects that can be very costly to fix at a later state. Finally, after the testing phase is finished the team enters the evaluation phase. In this phase, the team assesses the development performed in the cycle and identifies problems and issues that arose during development.

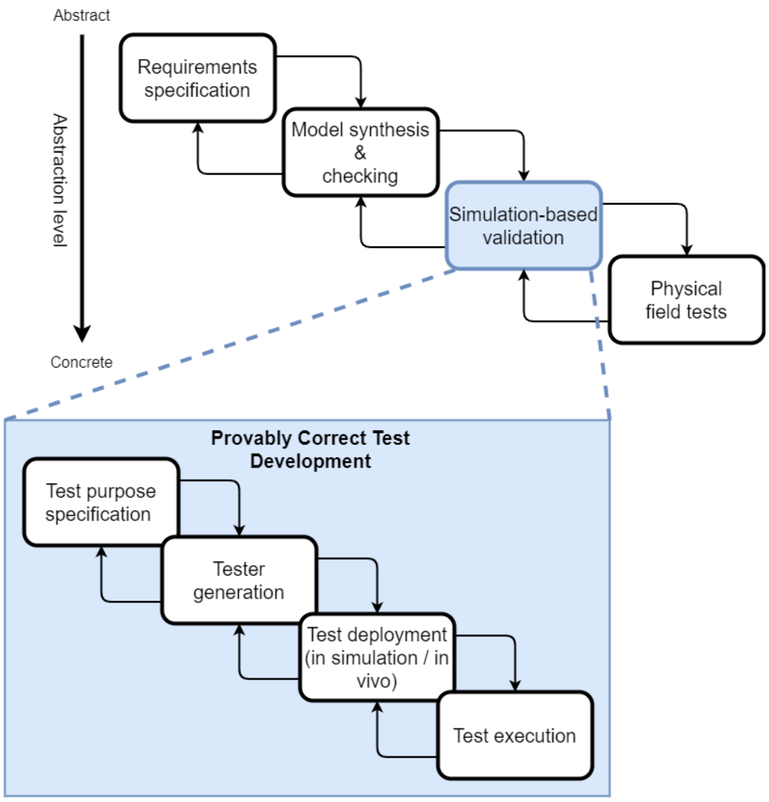

Abstract
Abstraction level
Concrete
Requirements specification
Model synthesis & checking
Simulation-based validation
Physical field tests
Provably Correct Test Development
Test purpose specification
Tester generation
Test deployment (in simulation / in vivo)
Test execution

Figure 4: CPS validation process.

The SDLC must also fit into the general system validation process. This process, presented in Publication III, as depicted in Figure 4, starts after new increment of requirements is specified and partially runs in parallel with other development cycle steps such as design and implementation. The test results are used also in the evaluation phase as shown in Figure 3. The final SDLC quality assurance step is field testing that follows the deployment phase.

The simulation-based validation process depicted in Figure 4 presumes formalization of the requirements. The requirements are used to create the models in the model synthesis and checking step. Once the models have been created and checked with the model checker the process continues with simulation-based validation step. The details of this step are depicted in Figure 4 in the blue rectangle. This step contains the provably correct test development process that is detailed in subsection 2.9.2. After test purpose has been specified, the tests can be generated. The tests are then deployed to either simulation environment or to the server connected to real hardware. Finally, the test is executed and the results logged.

2.9.2 Provably correct test development

As detailed in Publication VI, provably correct development (PCD) studied in [129], capitalizes on the development process paired with verification and design correctness assurance steps. Applying PCD processes to testing is motivated by the need to improve the trustability of testing results by showing their formal correctness through entire test development and execution process.

Model-based test generation algorithms presume that SUT models have properties which guarantee the feasibility of generated tests. Among these properties are connectedness, input completeness, output observability and strong responsiveness. These properties formulated originally for input-output transition system (IOTS) models [26] can be verified on well-formed Uppaal TA models as well by applying Uppaal model checker.

Publication IV presents verification conditions and the model checking method of proving the correctness properties for test development steps.

As explained in Publication VI, after the test model has been proved to be well-formed, the test purpose is specified in terms of coverage criteria. The SUT model structural coverage is expressed as a set of boolean trap variable updates that are added to the Uppaal TA edges. The trap updates may also be conditional boolean assignments presuming some contextual on model state variables to be satisfied. Such edges labeled with traps need to be executed in the course of the test run for satisfying the entire test purpose. The time bounded reachability of such specified coverage items can be proved by applying model checking.

The test deployment in the physical test execution architecture introduces additional processing and communication delays that are not natively expressed in the SUT requirements specification. For instance, these delays of test harness are caused by the physical distance between the SUT entities and/or by non-negligible processing time in test adapters. Thus, proper test feasibility verification presumes also the test model update with these delays and rechecking the correctness properties preservation thereafter. This verification step concerns timing correctness of the executable test in the first place. Additional details about such delays and modeling of timing aspects can be found in Publication VI.

After passing the verification process the tests are ready for execution while the test results can be provably trusted.

2.10 Chapter summary

In this Section, it is argued that testing CPS and autonomous robot systems, in particular, requires a different approach to testing than traditional software testing. Related work section provides an overview of other candidate frameworks and tools to be used for MBT of AS. But as shown by analysis in Section 2.6 none of them is supporting the combination of functionalities and features supported by the framework developed in this dissertation. The chapter also introduces theoretical foundations applied in our testing framework, the Uppaal TA as a suitable formalism for modeling AS and its use for verification and testing. Since this dissertation is focused on model-based conformance testing of autonomous systems with timing constraints, the rtioco relation is defined as relevant conformance relation verified by MBT of time-critical autonomous robot systems.

3 TestIt: an open-source scalable testing toolkit

3.1 Chapter overview

The chapter gives a detailed overview of the testing toolkit TestIt. First, the general design issues are outlined and an explanation of the design choices that have been made regarding the toolkit is presented. Secondly, the architecture of TestIt is described. Thirdly, the scalability aspect of the toolkit is discussed. Following that, long-term autonomy testing is presented. Finally, the TestIt test runner featuring the test optimization algorithm is proposed. This chapter investigates the research question RQ2.

3.2 Design considerations

The general design considerations of the TestIt toolkit are well aligned with the findings of review on CPS testbeds in [130]. The authors of the review list the key future research issues for CPS testbeds. They find that the desirable characteristics include accuracy, automation, controllability and observability, reliability and reproducibility, safe execution, high speed and capacity. The authors of [130] agree that CPS testing is time consuming and labor intensive which is why test automation is key. Because of that the future tools should improve the generation and execution of suitable test cases since CPS have complex interaction between software, hardware and networks. TestIt features test automation as its core concept. The design considerations of TestIt are discussed in detail in the following sections from the perspective of compensating the shortcomings of existing solutions discussed in Section 2.6 and addressing the needs stated in Section 2.5. The advantages of TestIt framework are outlined in the course of presenting its novel solutions.

3.2.1 General considerations

TestIt is a flexible testing toolkit that is designed to be modular and extensible. Autonomous robot software is highly heterogeneous as robotic platforms are diverse. The capabilities and the purpose of robots also differs from robot to robot. This fact makes designing a universal testing tool for all robot types very challenging. To alleviate this problem, TestIt was designed to be open-source and flexible so that new testing tools can be integrated into it without significant overhead. For example, it is possible to run ROS linters (e.g., roslint [131] and static code analysis tools (e.g., HAROS [132])) as part of the testing process but it might not be required for robots not using ROS. Such individual requirements can be accommodated in TestIt due to its container-oriented design.

The tools that need to be used in the testing process for a given project can be packaged into the TestIt test Docker [133] container and executed as part of the test package. Thanks to this, TestIt is testing methodology agnostic. However, to demonstrate TestIt feasibility and considering the benefits of model-based testing described in the previous sections, support for model-based testing tool family Uppaal has been developed. Uppaal tool suite includes a wide range of tools [134] making it a sensible first choice. Also, based on the reasoning in section 2.3.3, ROS has been selected as the default supported robot middleware.

3.2.2 Suitability for MBT

One of the most important benefits of using MBT is the complex emergent scenarios that can be discovered by simulating both the SUT and the environment (i.e., the static world and the dynamic actors in the world) together. It is very difficult to design test scenarios for autonomous systems which explore the full software stack thoroughly. This is due to the fact that usually software is developed by different teams and the knowledge of the full software stack concentrated in a single individual is very rare.

Accounting for all permutations of the conditions that can arise in complex dynamic environments is exceedingly difficult even with full knowledge of the software stack. Using MBT and tools (e.g., Uppaal TA[116], NModel[135]) helps in this regard by allowing modularity and separation of design concerns that is important when applying the "divide and conquer" principle [136] in complex system testing. The possibility to model different actors of the SUT and the environment separately reduces the modeling complexity which in turn reduces the cost of testing. Taking advantage of the functionality provided by Uppaal, it is possible to instantiate multiple copies of the single actor models. This feature simplifies the test setup and increases test maintainability.

TestIt uses MBT approach to generate test cases in test model exploration (i.e., model execution) mode to create SUT traces (i.e., SUT input sequences). These traces are used by the test optimization algorithm (described in detail in Section 3.4.4) to create probabilistic models which in turn are used by the TestIt test runner to optimize test scenario execution online.

3.2.3 Continuous integration orientation

ROSIN [137], a project funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 732287, aims to amplify its impact by making ROS-based industry oriented components better and even more business-friendly and accessible. The ROSIN project report[138] highlights the need for better QA practices to be adopted in ROS software development. One of the main issues is that the QA practices are not consistent across the various development streams (i.e., core, drivers and reusable packages). The report also indicates that the utilization of CI service in its current form is not sufficient because it is simply compiling and building the ROS projects. Analysis results show that the QA practice would be improved significantly by extending the CI service to run a collection of different kinds of code-scanning tools. To address these needs, TestIt has been designed to be easily integrated with the CI service. The CI integration is simplified by the command line interface (CLI). TestIt can be controlled directly via CLI commands which can easily be integrated into any CI service. It also provides a framework that can be augmented with aforementioned tools and bundled in a convenient package.

Another aspect the ROSIN report highlights is that although testing is regarded as critical in robotics, developers working on new components tend to focus more on creating the components rather than creating and setting up tests and gathering data based on simulation. As stated in the report, automated testing should compensate this practice by saving time and increasing software quality. TestIt aims to comply with these recommendations. It reduces time overhead by supporting automation of all testing phases, maximizing testing efficiency by using concurrent testing pipelines and minimizing testing time by learning from executed tests and optimizing test scenarios based on that.

Considering this, integrating testing into robotics software development CI processes is highly coveted. TestIt is well suited for integrating into CI services as the testing pipelines are designed to work with Docker containers. Using Docker containers makes it easier to integrate into CI processes because of the ephemeral on-demand nature of the container technology. The containers are always started from the same state and the state is not stored after finishing, which is the desired behavior in testing context. This feature ensures that testing is stable and there is no risk of influencing the initial state on subsequent test execution. Executing tests without sandboxing the software can run into mutability issues.

3.2.4 Long-term autonomy testing support

Reliability of long-term autonomy is another key concern for autonomous robots which operate in dynamic environments. Finding software defects that appear immediately or within a short time window is significantly easier than detecting erratic defects that emerge after a long time has elapsed (e.g., memory leaks and cumulative corner cases). Still, long-term testing with real robots is challenging and limited by real-time factor.

In some cases, it is possible to perform simulation faster than real-time to further increase the time efficiency of testing. For example, when using ARGoS, a multi-physics robot simulator [139], with simple wheeled robots it is possible to simulate about 10,000 robots 40% faster than real-time. Stage simulator has been demonstrated to simulate a single simple robot 1,000 times faster than real time [140]. Time compression is also possible in more advanced simulators such as Gazebo [141], CARLA [142] and AirSim [143]. But as pointed out in [144], simulation at merely real-time is already challenging. In order to achieve high simulation speed, richer collision response and advanced ground interaction models have been ignored in AirSim. However, the gravity of the aforementioned concerns diminish over time as advances in hardware performance carry over into simulation speed and level of detail improvements.

Considering these factors, it is important to utilize the available resources to the fullest. To that end, TestIt supports running tests over long time periods to find interesting scenarios that are exceedingly difficult to discover without model-based generated tests.

Using simulation for long-term autonomy testing with compressed timescale improves efficiency by reducing the amount of real-world time spent on simulation. Testing at uncompressed timescale (i.e., wall time) does not offer any time improvements over testing in real-world but time compression is not always possible due to computational complexity involved with simulating high-detail robot models and environments. Time compression requires additional computational resources depending on the compression level. This is especially true for high detail simulation as faster than real-time physics simulation can consume infeasible amounts of resources [144].

3.3 Architecture

Publication I presents a model-based testing toolkit named TestIt[1], an open-source ROS package containing the daemon, a CLI (Command-Line Interface) program to interact with the daemon, and a Docker container[133] with bundled testing tools. A high-level overview of a common TestIt configuration is depicted in Figure 5. As can be seen from the figure, TestIt is executed by the CI server (e.g., Jenkins [145]) after being triggered to build and test the SUT software stack. The CI server can be configured to send feedback to the developer on test failure.

The architecture of the TestIt toolkit is shown in Figure 6. The TestIt Docker container and the SUT (one or more containers over one or more servers) together form a testing pipeline.

The configuration for the SUT and TestIt Docker container is defined in the YAML [146] format configuration file which is passed to the daemon upon start up. The configuration consists of infrastructure configuration and test scenarios. The pipelines can be configured depending on the available hardware or budget constraints for the cloud testing.

[1]https://github.com/GertKanter/testit

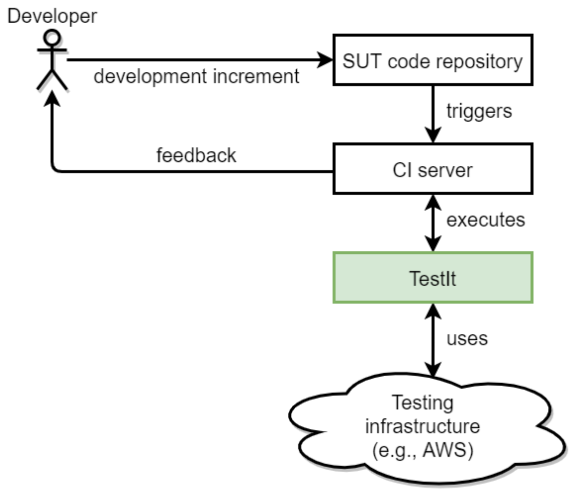


Figure 5: High-level overview of TestIt.

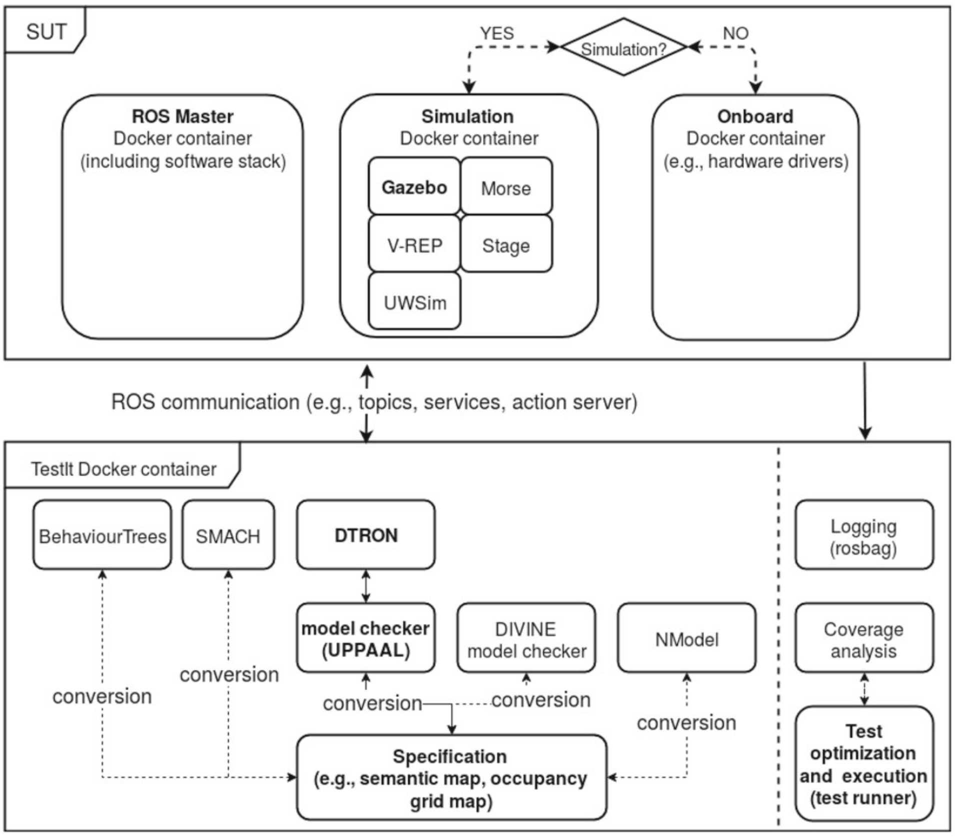


Figure 6: TestIt toolkit architecture.

3.3.1 Pipelines

TestIt daemon is the process which handles all TestIt commands (e.g., starting and stopping pipeline servers, initiating testing, examining results). The daemon can control multiple TestIt pipelines as can be seen in Figure 7. Test scalability is one of the benefits of simulation-based testing which using multiple pipelines provides. Each pipeline can run on a separate server, for example in the cloud (AWS [147], Google Cloud [148] or Azure [149]), ensuring scalability.

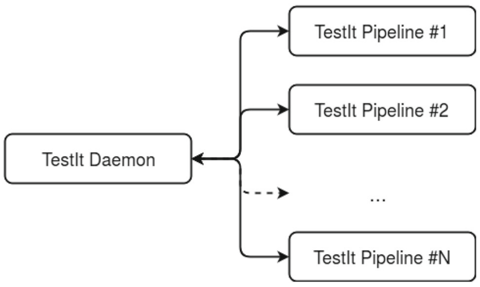


Figure 7: TestIt toolkit component communication diagram.

TestIt supports starting and stopping the servers hosting the testing pipelines as part of the testing workflow depicted in Figure 8. Only using the servers when needed helps reduce testing costs in case of using cloud servers for testing since cloud services are billed based on the time used. Therefore, only bringing them online when needed is cost efficient.

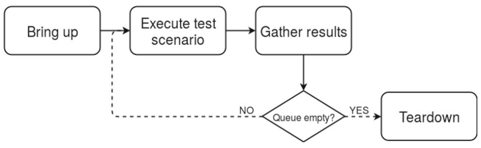


Figure 8: TestIt toolkit testing flow diagram.

3.3.2 Software packaging

The packaging requirements for the software running in the SUT are not strictly constrained. TestIt toolkit ROS integration relies on the SUT running the ROS master service (i.e., roscore) to which the tools in the TestIt container can connect. Other than that, the SUT can be considered as a black-box system and TestIt can be used as a black-box testing toolkit which requires no modification by the tester. If test code coverage measurement is required, the SUT must be configured in a way that supports it. The necessary preconditions are detailed in Section 3.3.3.

3.3.3 Test coverage measurement requirements

In model-based testing the test coverage can be measured either by using model coverage or coverage of the code units abstracted in the model or both combined. Code coverage measurement can target a number of different criteria such as function coverage, statement coverage, edge coverage, and branch coverage [25]. Statement code coverage measures the number of statements executed. Statement coverage and line coverage are often used interchangeably since conventionally each line contains a single statement.

Traditionally, line counters are never reset during the measurement process. This is not an issue in case of measuring the lines that are executed during a full test execution. This approach is refined with model-based testing with TestIt by measuring the lines of code that get executed when performing some action represented as a transition in the model. This detailed coverage information is later used to maximize the code coverage while keeping the test length reasonably bounded.

In order for the code coverage to be measurable, possibly in combination with model structural coverage, both the test model execution and the SUT software stack must support it. The main programming languages used in ROS software development are C++ and Python. For code coverage, the C++ stack must be compiled with code coverage support options (profile-arcs and test-coverage) [150]. Python programs must be run via a wrapper (e.g., Coverage.py library) which collects the code coverage information [151].

The SUT software stack has to handle the SIGUSR1 signal to support the state transition code coverage measurement. The C++ code needs to call *__gcov_flush()* function to flush the coverage data [152]. For Python, the coverage wrapper must call a save function. If *Coverage.py* is used, *coverage.save()* must be called [153] and internal class variables *lines* and *arcs* must be reset. If these variables are not reset, the subsequent calls to the save function will return the full list of lines executed since the start of the program.

3.3.4 Test scenario generation

The test scenarios specify the concrete cases that are executed to test the software. The scenarios can be executed in different ways. The simplest is to execute a program or script that gives inputs to the SUT software stack (e.g., goals for navigation) but the scenarios can be defined also using complex models. For instance, Uppaal timed automata [154] models used as test oracles are currently supported in TestIt and are executed using DTRON [155].

For other types of formal models and their analysis tools TestIt can be extended to accommodate other model checkers (e.g., DIVINE [156], NModel [135]), SMT constraint solvers (e.g., Z3 [157], CVC4 [158]) and other decision support tools in a way that does not inconvenience the toolkit user. This is one of the design goals of TestIt that the third-party components of the toolkit are packaged into the TestIt Docker container. It is possible to pre-install and configure everything inside the container so that the toolkit user or a CI service does not need to install and configure all of the tools separately.

One of the criticisms of model-based testing is the difficulty and labour intensive procedure of creating models. TestIt toolkit is addressing this issue by supporting generation of models from other specification formats as can also be seen in Figure 6. The generation of a model from topological map format developed for use in STRANDS project[159] has already been implemented in TestIt. The support for generating models from SMACH [160] state machines, ROS BehaviorTrees [161] or other formats can be achieved in a similar way. Creating models from other specification formats saves time and gives a good starting point to expand the models for automatic model-based testing. An algorithm of learning Uppaal TA models from SUT interface logs has been developed in Publication IV.

3.3.5 Test results logging

An important component for test results analysis supported by TestIt is the logger. The logs are used to diagnose the possible causes of test failure and to optimize the test scenarios for better coverage and efficiency. The log entries are stored as JSON notation strings with each string denoting one event. The entry is a dictionary with the test run identifier, timestamp, coverage information related to the transition, data transmitted to the SUT, transmission channel information and information whether the entry corresponds to before or after transmitting the information to the SUT. This discrimination allows the information to be further analyzed based on the result of executing a test model transition.

Specific feature to Uppaal TA logging is that TestIt gathers the information about the Uppaal TA channel that models interaction between the SUT and the environment contains the name, type and proxy name if required (for services and action library). The proxy is used to allow logging to occur without requiring modification of the software that is tested. This is caused by ROS design, namely, services can only be handled by a single server. To allow services to be monitored as state transitions, the logger needs to be able to provide a proxy service that forwards the actual service request to the SUT and gives the result to the requester. The proxies can be set up with ROS remapping without modification of the SUT.

An example of the log format (coverage data is omitted for brevity but more information about coverage is in Section 3.4.4) is shown in Figure 9.

```
{"run_id": "34e07c0c-316f-4044-8994-be95483e4af6", "timestamp": 49.2,
"coverage": {}, "test": "T1", "data": {"header": {"stamp":
{"secs": 0, "nsecs": 0}, "frame_id": "map", "seq": 1}, "pose":
{"position": {"y": 35.0, "x": 2.0, "z": 0.0}, "orientation":
{"y": 0.0, "x": 0.0, "z": 0.0, "w": 1.0}}}, "event": "POST",
"channel": {"identifier": "/robot_0/move_base_simple/goal",
"type": "geometry_msgs.msg.PoseStamped", "proxy": ""}}
{"run_id": "34e07c0c-316f-4044-8994-be95483e4af6", "timestamp": 54.6,
"coverage": {}, "test": "T1", "data": {"status": {"status": 3,
"text": "Goal reached.", "goal_id": {"stamp": {"secs": 49,
"nsecs": 200000000},
"id": "/robot_0/move_base_node-1-49.200000000"}}, "header":
{"stamp": {"secs": 54, "nsecs": 600000000}, "frame_id": "",
"seq": 0}, "result": null}, "event": "RESPONSE", "channel":
{"identifier": "/robot_0/move_base/result",
"type": "move_base_msgs.msg.MoveBaseActionResult", "proxy": ""}}
```

Figure 9: TestIt log format.

3.3.6 Test runner

The final component of TestIt is the online test runner. This component uses the optimization algorithm to dynamically guide the system into maximum gain states (e.g., gain function maximizes code coverage). As the online tester is executed at the same time as the SUT, it is possible to take actual gain information from the SUT into account while planning the next SUT input signal.

3.4 Features

3.4.1 Test configurability and observability

As robotic systems are heterogeneous, the toolkit needs to be configurable to accommodate the wide field of projects. TestIt toolkit has been designed to be flexible in terms of configuration. The general test configuration and execution cycle contains the following steps: infrastructure allocation, SUT launch (simulation-in-the-loop or hardware-in-the-loop), testing and infrastructure deallocation. TestIt allows the testing step to be configured in different ways. It is possible to configure the test to optionally include a test oracle. A test oracle is a software component that observes the SUT while testing takes place and makes a verdict whether the test was a $pass$ or $fail$ ($inconclusive$ is regarded as $fail$ in this dichotomy). In the case of not using model-based testing, it is possible to configure TestIt to receive the test verdict from the test script itself.

Observability is another key concern while configuring the test process for the project. Autonomous robotic system software is usually composed of multiple components and might include complex inter-process dependencies. Due to this, it can be difficult to configure the test process efficiently. TestIt features different verbosity levels for monitoring the test process so that misconfiguration detection is simplified by receiving more output from TestIt and the SUT.

3.4.2 Scalability

Testing of robotic systems often suffers from the problem of scalability due to complexity issues. Scalability issues can be mitigated by introducing simulation-based testing into the testing process. However, in order to take full advantage of the simulation-in-the-loop approach the simulation must be parallelized (multiple simulation threads running concurrently), the process must be controlled dynamically (infrastructure allocation, deallocation and reallocation) and the results of simultaneous threads must be aggregated for analysis and test optimization.

The support for parallelized testing is present in the proposed toolkit TestIt. TestIt allows testing to be scaled up using multiple testing pipelines. Using several pipelines increases testing throughput by running multiple simulations concurrently. In Publication I, linear scalability of TestIt using multiple pipelines is demonstrated (Figure 11). The details of the individual pipeline configuration used to acquire the shown results are presented in Section 4.7. This linear scalability is possible w.r.t. log data generation speed in both test exploration mode (i.e., execution of the SUT and environment state machine) and test optimization mode. The speed increase w.r.t. the code coverage and other criteria acquisition time might not be strictly linear since the test pipelines might have some overlap in terms of the state space traversed for each test scenario. For example, in case the robots (i.e., the SUT) always have to start at the same initial conditions and different scenarios split only after several identical steps in all scenarios. In such case, it is understandable that the increase is not linear w.r.t. the number of pipelines as there is some overlap between the scenarios.

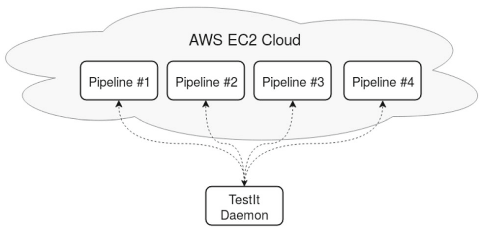


Figure 10: Test infrastructure configuration.

The scalability of TestIt was measured in AWS EC2 cloud. The infrastructure configuration is shown in Figure 10. During the testing procedure TestIt daemon managed the infrastructure with four pipelines dynamically. The servers were brought online just before the test execution and shut down upon test completion in order to optimize test resource consumption. During the test, TestIt loggers recorded log entries in each pipeline.

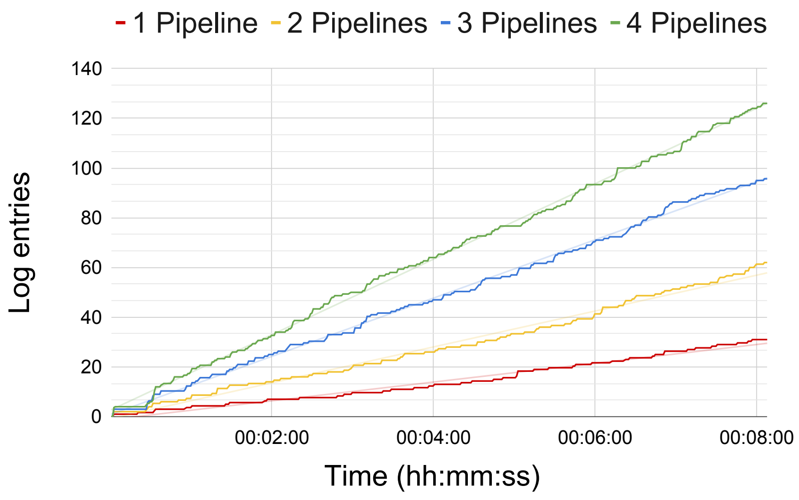


Figure 11: Linear scalability of TestIt.

The numerical results are shown in Figure 11 to demonstrate that entry generation scales roughly linearly in the number of pipelines configured in the system. As the plot is based on measuring simulation data there is a small margin of variance due to the fact that the robots generate state transition logs at different rates based on their actual navigation speed and waypoint reaching success rate. It must be stressed that linear scaling is expected as individual pipelines are operating separately and the overhead from communication with the TestIt daemon is negligible. The individual logs are retrieved from test pipelines after tests are finished which means there is no data transmission bottleneck even with very large pipeline configurations.

3.4.3 Long-term autonomy testing

Autonomous robotic systems are usually designed to operate for extended periods of time which adds another facet to their software testing. It is considerably less cumbersome to verify that the SUT conforms to its specification in the short-term scenarios compared to the long-term ones. Long-term autonomy can be regarded as the highest level of autonomy as it involves the robotic system operation for a long time without intervention. Testing physically whether the SUT performs without defects over a long time period is complex and expensive.

As pointed out in 2.5, the long-term software testing of autonomous robot systems can be simplified and its cost drastically reduced by utilizing simulation. Naturally, the quality of the simulation engine and the detail level of the robot and the environment plays a major role in the performance gain that can be achieved using simulation in testing. The SUT software algorithms can be tested at a higher level (e.g., task allocation and navigation planning) using simplified simulation models and physics simulation with limited realism. Testing lower level software requires more advanced simulation models and environments in order to be beneficial to the overall quality assurance which can require exponentially more resources to develop.

An aspect that simulation-based testing will provide the testing process regardless of the simulation model and environment quality or level of detail is verifying that the software does not have memory leaks or issues stemming from long-term operation. This is due to the fact that the SUT software will exhibit memory leaks in both the real-world hardware and the simulation environments. TestIt can be configured to run the simulation without concrete time limit while monitoring the memory consumption for each software component. This monitored value can also be integrated into the test run optimization as a weighted argument of possible test gain function. For example, there might be a memory leak in very specific conditions that can occur only when several circumstances coincide and its coverage by test gives substantial gain increase.

As a concrete example, let us suppose a situation where there is a defect in the testable software of a mobile robot that adds detected objects into a list in memory. The defect in the software would be that it adds objects without releasing the allocated memory once the object is no longer detected thus exhibiting a memory leak. This scenario can be detected with TestIt as it can be configured to prioritize a scenario in which the total memory consumption of the software stack grows without limit. The resulting scenario would possibly consist of a loop that involves guiding the robot to a location where it would add the detected object, then guide the robot away from the location so that it could add the object again later. This software defect and its detection do not rely on high detail simulation as this memory leak will exhibit itself regardless of the simulation quality level. Of course, this example is simplified and this concrete defect should have been detected at the unit testing level but this example serves only to give some intuition what kind of defects can be detected using this approach.

To increase simulation-based testing efficiency even further, it is feasible to use time compression in simulation. Accelerated simulation is supported in multiple simulation engines such as Gazebo simulator [141] and Stage simulator [162]. Increased speed increases the amount of potential defect-exposing encounters the robot will experience and also allows to maximize the resource utilization. The real-time factor can be fine-tuned to match the test infrastructure performance capacity to maximize the test efficiency.

As discussed in the previous section, TestIt can be scaled up to test multiple instances of the SUT by means of parallelization over the available test pipelines. This concurrent testing also allows gathering simulated defect-free operation hours and use in analogous study of the system to estimate mean time between failures like MTBF methodology is used for hardware testing.

3.4.4 Test optimization support

One of the most important features of TestIt is the concrete test model independent test scenario optimization functionality. As model-based testing approach allows generation of infinite test cases by executing the models, it is crucial to optimize the traversed state space to trim superfluous steps and only include the important steps. The test optimization algorithm is inspired by the reactive planning tester (RPT) gain tree construction method proposed in [163].

As detailed in Publication I and Publication II, the test optimization algorithm minimizes the input sequence to the SUT to provide the maximum gain in the least amount of steps. At its core, the algorithm recursively deepens the gain tree to the specified maximum depth level. The algorithm (pseudocode is presented in Algorithm 1) is initialized and started in $compute\_sequence()$ function. The initial state is the $(None)$ node of the optimization graph (Figures 13, 14 and 15).

The supporting functions $get\_state\_hash()$ and $get\_chan\_hash()$ provide a way to encode and decode the states and data channels so that they are unique and can be used as dictionary key values for the data structures used in the optimization algorithm. These

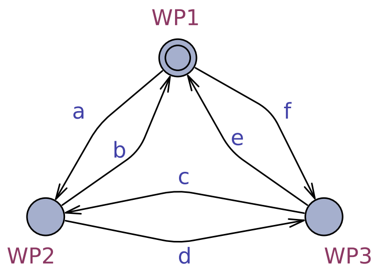


Figure 12: An example model.

data channels can be both internal and external w.r.t. ioco relation. For example, they can be the messages passed between different software components which are not observed at that particular model refinement level. As a concrete example, this can be the robot battery level data channel. This data channel might be used by the navigation component to decide whether to select a shorter but more battery draining path or longer but more easily traversable path.

The $expand()$ function expands each tree node (calculates the gain values of the children). Each gain tree level depends on the values of the previous level which means it has to be created level by level. After reaching the maximum depth level, the algorithm propagates the gain values (i.e., gained code coverage or other desired parameter gain) from the terminal nodes (the leaves at the maximum depth) to their ancestors up the tree to the root via the $update\_path\_gain()$ function. Once all the gain values have been backpropagated to the root node (creating the path of maximum gain) the algorithm can just return the best gain path first element of this maximum gain path as the next input to the SUT. The function $compute\_edge\_gains()$ computes the edge gains (i.e., possible gain values for the next step) for a state based on the current parameter state (e.g., it takes into account the code lines which have already been covered).

The novelty of the optimization algorithm is that it takes into account the actual values (code coverage and other measured parameters) from the online test runner. These measured values are compared to the probabilistic values based on the probabilistic model constructed by the optimization algorithm. The optimization algorithm gathers data from all parallelized pipelines to construct the probabilistic model. This probabilistic model is used to find the best sequence of steps to maximize the gain of code coverage or other criteria.

For example, the probabilistic model might contain the probabilities of the SUT software to execute specific lines of code after sending the specific input while being in a specific state. Since models are abstractions of the actual system the probability measures are required to handle the inherent non-determinism. In other words, the specific lines of codes might or might not be executed due to some hidden variable that is not accounted for in the SUT model.

An example state transition model of the SUT used for test optimization can be seen in Figure 12. The states (waypoints in this example) are denoted $WP1$, $WP2$ and $WP3$. The transitions between the states are labeled from a through f.

The optimization algorithm starts by constructing a graph from the logs. For clarity, only the relevant parts of the log entries are presented. The following example has three separate test execution paths denoted as L_1, L_2 and L_3.

$$\begin{aligned} L_1 : (a, cov_1 = \{...\}, t = 0) \rightarrow (b, cov_2 = \{...\}, t = 2) \rightarrow \\ (f, cov_3 = \{...\}, t = 3) \\ L_2 : (f, cov_4 = \{...\}, t = 0) \rightarrow (c, cov_5 = \{...\}, t = 1) \rightarrow \\ (b, cov_6 = \{...\}, t = 4) \\ L_3 : (a, cov_7 = \{...\}, t = 0) \rightarrow (d, cov_8 = \{...\}, t = 4) \rightarrow \\ (e, cov_9 = \{...\}, t = 5) \rightarrow (f, cov_{10} = \{...\}, t = 6) \end{aligned} \tag{6}$$

Each execution comprises several entries (state transitions) and is denoted as a tuple (T, cov, ts), where T is the transition (the source state is inferred from previous state), cov is the the code coverage set (i.e., the set of lines that have been executed since last query) recorded at the moment of logging and ts is the timestamp of the entry. An example of entries is shown in Eq 6. The coverage entries encode the dictionaries where the keys are tuples $(file, l)$, where $file$ is the file name and l is the code line (or non-overlapping code interval if pre-processed in this way) that was executed since the last log event. The values referred by the dictionary are the probabilities of the specific line (or interval) being executed. This probability is initially always 1.0 but the probability is modified as the graph is simplified. Each simplification step includes merging of edges and the coverage probability changes when similar edges are merged. The merge takes place if the transition occurs in the log multiple times. This results in several edges occurring between the same states. These transitions can be merged into a single edge with modified combined probabilities.

An example of such a merge is shown in Eq 7. As can be seen, if the line is executed in both coverage sets, the probability remains 1.0 but if some lines of code are not present in both coverage sets, after normalization the combined probability is reduced. In the example, line 1 of file a is executed in the coverage set cov_1. The cov_2 set specifies that lines 1 and 2 of file a are both executed. This implies that line 2 of a was not executed in cov_1. Therefore, when combining the coverages we deduce that line 2 of file a is executed with probability 0.5 and line 1 of file a is executed with probability 1.0.

$$\begin{gathered} cov_1 = \{(\text{"}a\text{"}, 1) : 1.0\} \\ cov_2 = \{(\text{"}a\text{"}, 1) : 1.0, (\text{"}a\text{"}, 2) : 1.0\} \\ n(cov_1 + cov_2) = \{(\text{"}a\text{"}, 1) : 1.0, (\text{"}a\text{"}, 2) : 0.5\} \end{gathered} \tag{7}$$

It is possible to create the optimization graph in three ways with each method providing different benefits. The first method and the one we use in the case study is the probabilistic graph optimization shown in Figure 13. The probabilistic graph is cyclic by construction as long as the logs it is constructed from have sequences in which the system has visited the same states in a loop. For example, if the logs show a transition from A to B in one log and from B to A in another, the resulting graph will have a loop between A and B states. Because of this the probabilistic graph is compact and allows long SUT input sequences to be generated due to normally having cyclic structure. Considering the previous example, we could generate an infinite input sequence $A \rightarrow B \rightarrow A \rightarrow B \rightarrow ...$ with each transition potentially providing some gain.

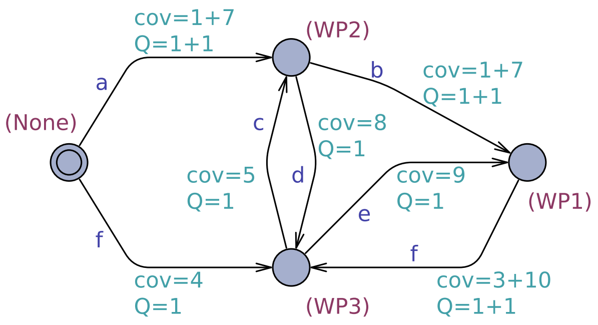


Figure 13: Graph for probabilistic optimization.

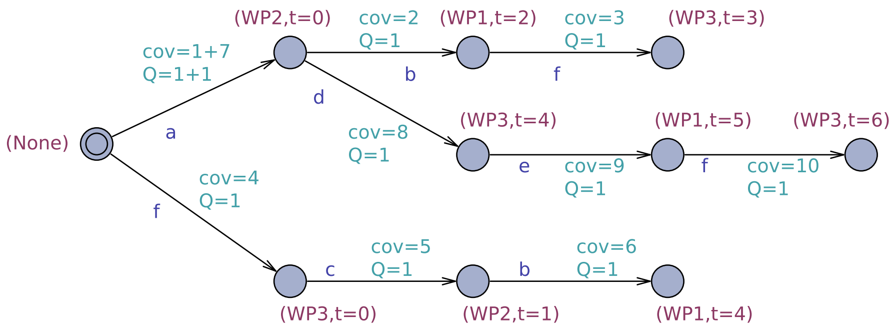


Figure 14: Graph for best trace optimization.

The second option is the best trace optimization for which the graph is constructed as a tree of state transition sequences that is based on the logs. As can be seen in Figure 14, the states are linked together as chains but there are no loops. This means that we can only generate a finite number of inputs to the SUT starting from the initial state because of bounded branching factor of the tree of finite depth. This method is the least computationally complex but the resulting graph can not be used to generate long sequences. This means that once the best trace has been traversed the test stops and no effort is made to attain more coverage based on recorded probabilities. This option can be used to simply pick and execute the best trace (chosen by static analysis) from all known traces. There are scenarios in which this can be useful. For example, if the logs are generated from real world data and tests are executed on real world systems with the requirement that the generated SUT input sequence has been seen before and is guaranteed to have worked previously.

The final option is the combination of the two aforementioned methods shown in Figure 15 and creates graphs that have both the logs with timestamps but also generalizes to probabilistic optimization after traversing through the logs. In order to benefit from the best trace optimization (i.e., initially follow a logged input sequence) but still be able to continue the test after traversing a known trace the final state of an actual log entry needs to be connected to a probabilistic fragment of the graph.

Following the optimization graph construction, the gain tree is created which is what the optimization algorithm uses to determine the best input to send the SUT to gain the most code coverage improvement (in the exemplified case). This can be combined with other optimization criteria such as localization covariance (localization uncertainty) [164]

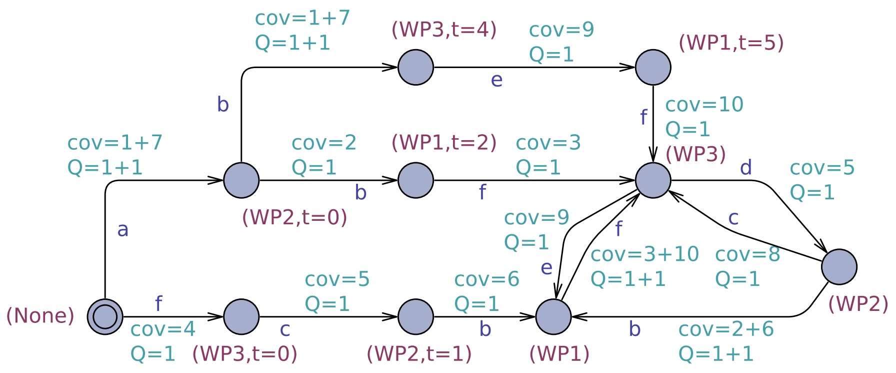


Figure 15: Graph for combined (probability and trace) optimization.

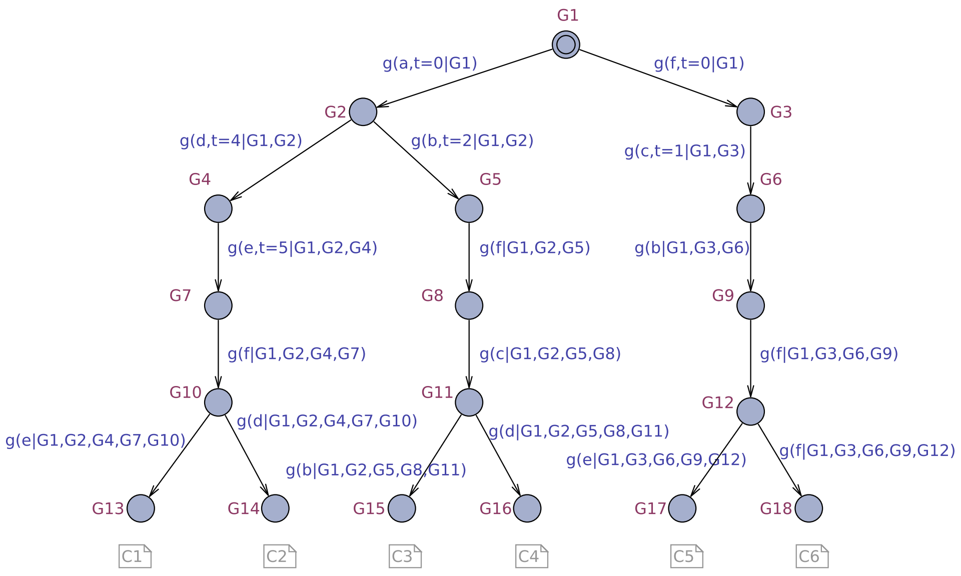


Figure 16: Optimization gain tree.

based on the simulation. It is possible to find sequences of inputs that maximize such uncertainty which in turn creates efficient test scenarios by forcing the SUT into a difficult situation. This difficult situation can possibly reveal a software defect that would not be revealed without the confluence of said localization uncertainty and some other action. The gain tree based on a probabilistic graph in Figure 15 is shown in Figure 16. The optimization algorithm generates the gain tree from top down with each level denoting its depth level. Each successive level contains more nodes and the number of which is directly dependent on the branching factor of the optimization graph. Large branching factor graphs will reduce the effective maximum depth of the tree since computation requirements will increase as more nodes are added to the tree. The optimization algorithm is greedy in the sense that it will choose the highest gain increase chain (chains are denoted as C1 through C6 in Figure 16) at each step.

The properties of Algorithm 1 are demonstrated in Section 4.

As detailed in Publication I, a sample TestIt incremental test development workflow is

Algorithm 1 Test scenario optimization algorithm

```
 1: $tree\_id \leftarrow 0$
 2: function compute_sequence(state,step_lim,max_d)
 3:     $seq \leftarrow []$
 4:     $next \leftarrow [state, \{\}, 0]$
 5:     for _ in $range(step\_lim)$ do
 6:         $next \leftarrow$ compute_step(max_d,next[0],next[1])
 7:         $data \leftarrow$ get_state_hash(next[0])
 8:         $chan \leftarrow$ get_chan_hash(next[0])
 9:         $seq \leftarrow seq + (chan, data)$
10:     end for
11:     return $seq$
12: end function
13: function compute_step(max_d,state,param)
14:     $tree\_id \leftarrow 0$
15:     $gain\_tree \leftarrow$ expand$(\{\}, [0, state, \{\}, param], max\_d)$
16:     update_path_gain($gain\_tree$)
17:     $best\_g \leftarrow 0$
18:     for $key$ in $gain\_tree$ do
19:         for $child$ in $gain\_tree[key]$ do
20:             $node \leftarrow gain\_tree[child[0]]$
21:             if $node = None$ then                                ▷ Terminal node
22:                 if $child[4] >= best\_g$ or $best\_g == 0$ then
23:                     $best\_g \leftarrow child[4]$
24:                     $best\_step \leftarrow child[5][1]$
25:                 end if
26:             end if
27:         end for
28:     end for
29:     $sel\_step \leftarrow None$
30:     for $edge$ in $gain\_tree[0]$ do
31:         if $edge[0] == best\_step$ then
32:             $sel\_step \leftarrow edge[1]$
33:             $best\_param \leftarrow edge[3]$
34:         end if
35:     end for
36:     return $(sel\_step, best\_param)$
37: end function
```

```
function expand(tree,elem,max_d)
    $el\_id \leftarrow elem[0]$
    $state \leftarrow elem[1]$
    $param \leftarrow elem[3]$
    $depth += 1$
    $gains \leftarrow$ compute_edge_gains$(state, param)$
    $tree[el\_id] \leftarrow []$
    for $g$ in $gains$ do
        $tree\_id \leftarrow tree\_id + 1$
        $new\_el \leftarrow [tree\_id, g, gains[g][0], gains[g][1]]$
        $tree[el\_id] \leftarrow tree[el\_id] + new\_el$
        if $depth <= max\_d$ then
            $tree \leftarrow$ expand$(tree, new\_el, max\_d, depth)$
        end if
    end for
    return $tree$
end function
```

depicted in Figure 17. The shown workflow is designed for offline test optimization with incremental feasibility verification using model checking. The input to the workflow is an executable SUT increment. Its execution in interaction with the simulated environment provides the SUT input-output logs. The logs include context information, timestamps and code coverage data, etc. necessary for test coverage optimization.

The model updates need to be verified for feasibility of test generation. The model is feasible if the goal specified in terms of the SUT model elements or other coverage items is reachable. The test goal can be, e.g., the coverage of a specific code segment in the SUT or reaching a specific state in the SUT model. The test goal reachability is verified using model checking [165]. In case the verification provides a negative result, it means either the log file does not include necessary traces to provide an executable model and further monitoring experiments are needed, or alternatively, the design itself is incorrect and excludes the behaviors that implement thetest goal. The decision on how to resolve the issue is put to the SUT developer and the test engineer involved in the process. Provided the verification proves the test model feasibility, i.e., the tests can be generated using the test model, further steps proceed with the test optimization.

3.5 Chapter summary

In this chapter, TestIt toolkit and its features are described in detail and the test optimization algorithm is presented. TestIt toolkit's core features such as scalability, simulation-based long-term autonomy testing and configuration flexibility were presented in the context of needs articulated in autonomous robotic testing state-of-the-art scientific literature and practice.

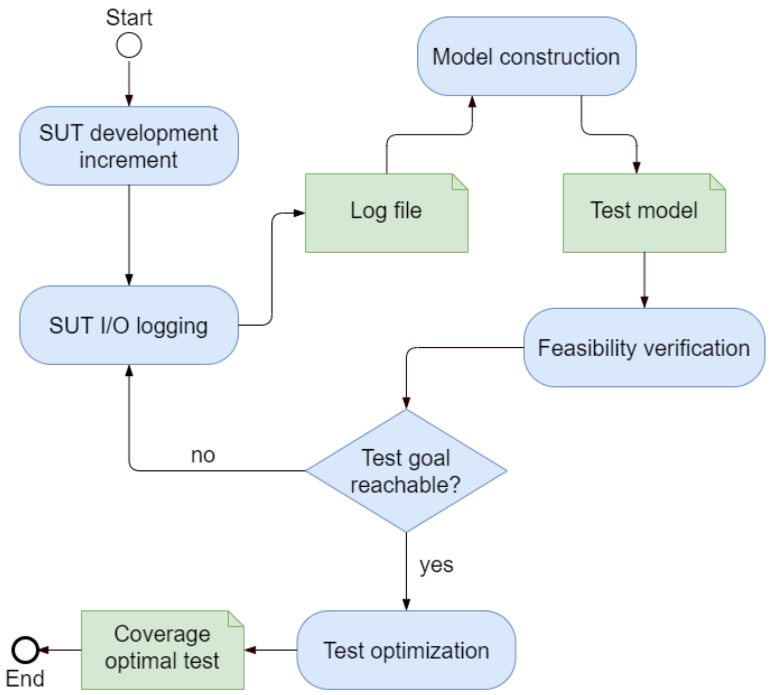


Figure 17: Offline testing workflow diagram.

4 Case study: Model-based testing of robotic intruder detection system

4.1 Chapter overview

This chapter describes the case study used in the demonstration of TestIt toolkit features and functionality. First, the system under test as well as the test scenario is described in detail. Secondly, the test model construction for model checking purposes before test generation phase is described. Thirdly, the test feasibility verification using model checking is covered. This is followed by the description of the TestIt configuration that was used to acquire the results. Finally, the results and a conclusion of the study are presented. This chapter investigates the research questions RQ4 and RQ5.

4.2 Rationale of the case study

The feasibility of the methods introduced in the dissertation and the TestIt toolkit is demonstrated on a robotic intruder detection system. Publication I presents a concise case study for the toolkit demonstration that presents the benefits of MBT-based approach clearly.

Case studies are commonly used to validate and demonstrate the benefits of the proposed theory or method. The most suitable use cases are emblematic of a larger population of use cases. The TestIt toolkit has been used in different industrial and academic projects and it has shown to be useful in widely variable contexts. Real-world projects are complex and have many superfluous details that impede clear understanding of the benefits the toolkit provides. In addition to that issue, the real-world projects might only use a subset of features that need to be demonstrated. To address these shortcomings, a synthetic but realistic use case that is a fragment of a larger building wide security system has been created.

One important design goal of TestIt is to increase software testing automation which is supported by the fact that the toolkit is based on black-box testing. Black-box testing approach enables the toolkit to test the SUT without requiring intensive manual effort for designing the test suite. We showcase this aspect with the case study.

The case study focuses on testing the software of security guard robots. The software under test is the software component dealing with the building floor exploration and cooperative intruder detection. For the intruder detection software to be tested it is necessary to have an intruder present in the environment partition of the model. We also know that there can be more than one guard robots patrolling the area to increase the patrol coverage. Therefore, the demonstration use case features several actors: two patrol robots and a single intruder. Each patrol robot moves around in the environment while using its lidar sensor for both navigation as well as for intruder detection.

4.3 System under test

The SUT in the case study is the software of the guard robot. The case study features an intruder detection algorithm together with the patrol planner component which is responsible for planning the patrol route. These algorithms together constitute the software components under test. The features outlined in this SUT are the concurrency of actions, non-determinism in decision making, coordinated navigation and timing constraints to be addressed in testing. The goal of the testing toolkit in this case study is to maximize the test code coverage.

The robot has several co-acting software components to achieve the task of security patrolling. The ROS computation graph is shown in Figure 18. Rectangles denote ROS

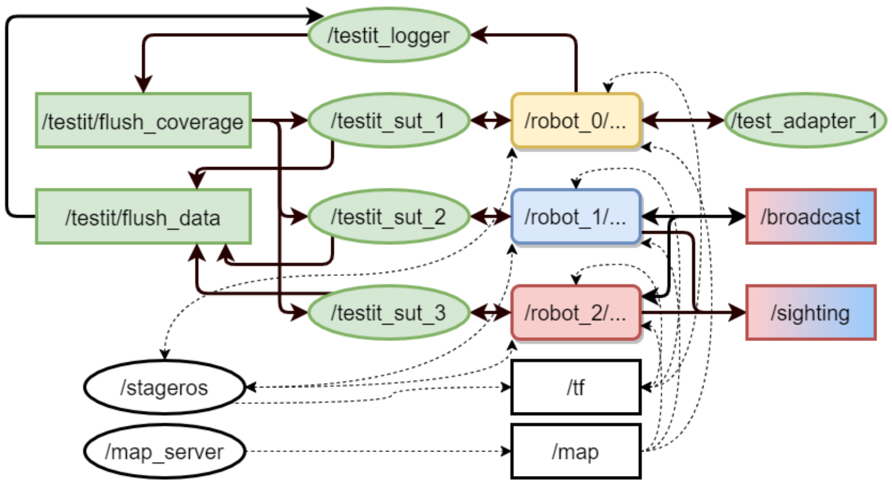


Figure 18: Simplified ROS computation graph.

topics and ellipses denote ROS nodes (software components). As can be seen from the graph, the SUT contains three robots (two guard robots and one intruder). Each robot has similar components which are needed for navigation. In addition to the navigation components, the system contains the TestIt SUT service[2]. This component is necessary for gathering the code coverage data. The code coverage can only be gathered at the SUT locally as it relies on sending signals (i.e., SIGUSR1 signal) to the software components which are then handled by each component by dumping the coverage information.

The SUT software includes also the standard ROS navigation software which is not shown in the figure (i.e., $move\_base$ [166] with $AMCL$ [167] localization component). The $move\_base$ component is responsible for planning the trajectory of the robot. The component essentially consists of the global and the local planner. The global planner finds the navigation trajectory based on the robot position on the map and the goal position. The local planner on the other hand uses the global plan to create a local plan which takes into account dynamic obstacles that are not present on the static map.

The $AMCL$ localization component is responsible for providing correction to the robot pose. This component uses the lidar data (i.e., laser scan) to detect whether the robot actually has the position and the orientation it believes it has. Position and orientation correction is often required as the initial prediction is based on odometry data. This data is seldom accurate as the wheels of the robot can slip which introduces defects into the predicted robot pose.

The planner component determines the least visited neighbor of the current waypoint and sends that waypoint as the goal to the $move\_base$ navigation component.

The SUT software publishes and subscribes to $broadcast$ topic which updates the known guard robot positions to the MRS. When a guard robot detects an intruder, it sends a message to $sighting$ topic.

The components denoted with green color are TestIt components. The system contains one test adapter ($test\_adapter\_1$) which acts as a bridge between the Uppaal TA model and ROS. The adapter transmits navigation goals to $robot\_0$ which acts as the intruder in this case study. Since Uppaal TA model is controlling only the intruder, only this robot is connected to the triggering of log events. Log events are triggered every time the robot

[2]https://github.com/GertKanter/testit_sut

is sent a navigation goal and upon receiving navigation feedback. Upon receiving the logging request, the logger component publishes a message to *flush_coverage* topic which triggers code coverage data transmission to the *flush_data* topic by the *sut_∗* nodes. The system contains multiple copies of the TestIt SUT component because each component runs on a different server. Section 4.7 contains more detailed information about the use case configuration.

Figure 18 also contains the simulator node (*stageros*) and map server (*map_server*) for completeness. The simulator node publishes sensor data to the robots and receives motor commands (i.e., *cmd_vel*) commands from the robots. The map server publishes the map data which in this use case is the floor plan of the building.

4.4 Test scenario description

The test scenario in this case study deals with the navigation and intruder detection in the ICT building of Tallinn University of Technology in Estonia. The floor plan of the fourth floor of the building has been implemented in the Stage simulator and can be seen in Figure 19. Stage simulator was chosen as the simulator for the case study primarily because of its simplicity. Demonstration of TestIt capabilities does not require an advanced simulator. TestIt itself is simulator agnostic allowing various simulators to be used depending on the SUT specifics.

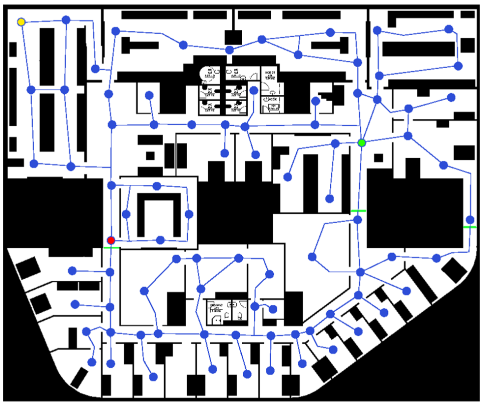


Figure 19: ICT building navigation graph.

The scenario includes three robots: two guard robots and one intruder.

For the sake of simplicity the intruder always starts at the same initial position (the top left node on the navigation map) in the navigation map and can choose the next waypoint to move to randomly. The patrol robots also start at the same starting location and they move along possible routes from waypoint to waypoint by preferring the least attended neighbor waypoint or if there are several options the decision is based on the waypoint identification number by preferring the larger identification number.

The patrol route before the intruder detection may be quite long and in general case it takes considerable time to complete. This means that in case a software fault occurs near the end of the route, it would take an excessive amount of time to recreate the scenario which would include numerous state sequences leading up to fault occurrence (i.e., software defect reproduction).

When using simulation for testing, it might be possible in some cases to drastically reduce the fault reproduction time by initializing the simulation at or near the fault occurrence state and/or optimizing the length of the test path and detect the fault again. This is possible only in cases of low degree of non-determinism in the model, i.e., when the model is sufficiently elaborate to correspond to the simulation or real-world environment completely. As this is rarely the case, it is not feasible to make this assumption and rely on this for improving the test efficiency.

Faults can also often exhibit after a specific sequence of events occurs which can not readily be identified. This sequence can not always be extracted as it is unclear what size fragment needs to be extracted in order for the fault to reoccur upon reinitialization.

Lastly, faults such as memory mismanagement can not be reproduced by reinitializing the simulation at the fault occurrence state as it might not occur without the correct history (prefix sequence). TestIt toolkit makes it possible to guide the SUT into the states along the test paths where the events of interest have occurred with higher probability. TestIt identifies the highest potential gain sequences based on occurrence probability and takes the achieved goals (e.g., code coverage) into account dynamically while directing the test path towards yet unachieved goals (i.e., the gain function changes over time). The gain function composition and test optimization is elaborated in 3.4.4.

4.5 SUT model construction

Typically, model-based conformance testing process starts with SUT model construction. This is based on the formalization of the requirements the SUT is implemented from. Test model construction from requirements is common in most of test driven development methodologies as discussed in Publication VI. Alternatively, the model can be learned from system monitoring logs by applying various machine learning techniques. In Publication IV, an algorithm has been developed for learning the Uppaal TA composition under synchronous communication assumptions. The feasibility of the algorithm has been demonstrated on the IEEE1394 leader election protocol example. Therefore, for ROS-based robot systems where asynchronous communication is prevailing as demonstrated in Publication I, we apply the combined approach where the model for test purpose feasibility verification is constructed from the SUT requirements description. For test optimization the model is augmented with additional coverage information extracted from the SUT interface logs. The latter is to demonstrate the TestIt usefulness in the development processes where the test model is evolving in lock step with the SUT development increments.

For verifying the feasibility of the test with respect to the test purpose specification the reachability of targeted test coverage needs to be proved, at first. This helps avoiding generating inconclusive test cases and waste of time when designing further testing steps. In the sequel, at first, the test model construction and test feasibility verification are demonstrated based on Uppaal TA modeling formalism and Uppaal model checker. The model for test feasibility analysis can be constructed even before the real development of SUT is started since the model and verification goals can be extracted directly from system requirements specification. Further coverage based test optimization steps can be made later incrementally in the course of SUT implementation, as demonstrated in the rest of this section.

For model construction, the state space of the model is constructed at first, to specify the conditions and effects of model actors. The actors are an intruder, and two robots patrolling on the office floor. The formal description of actors environment is based on a real office building floor topology that is abstracted in the form of a navigation graph (Figure 19). The data structure representing the navigation graph is a vector that consists of graph nodes names. Each node in the graph denotes a waypoint to be covered when the robots

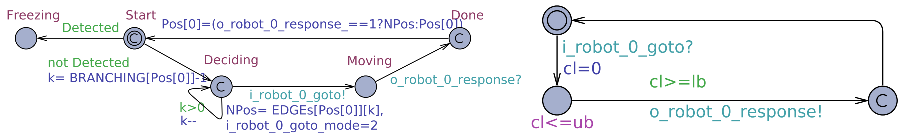


Figure 20: Intruder model and responder.

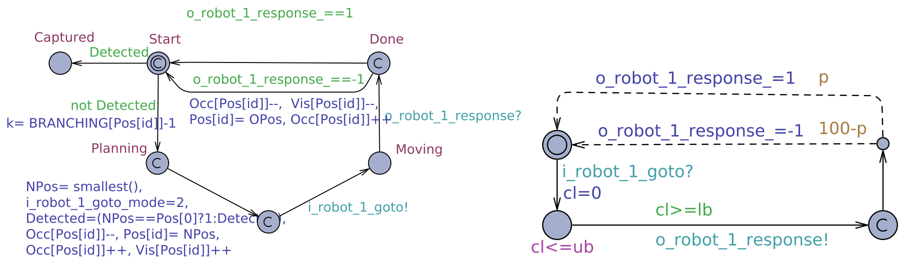


Figure 21: Robot 1 model and responder.

are patrolling. At the same time, the waypoints also denote the potential locations of the intruder to be detected. Possible moves of actors in the navigation graph are modelled using a two-dimensional array EDGEs with first dimension of length N (N is the number of waypoints and second dimension of length Br (Br is maximum branching factor of the graph). Each row in the array EDGEs corresponds to a waypoint and the elements of a row correspond to the neighbour waypoints of that node. The model is depicted graphically in Figures 20 and 21.

The data structures that describe the actors' state are vector Pos the elements of which encode the current positions of actors in the navigation graph. The vector Occ of size N encodes the occupancy of waypoints by robots and vector Vis the number of visits to each waypoint. Boolean variable $Detected$ is assigned the value $true$ when the intruder is located. Each actor's behavior is modelled using a pair of automata, one emulating the agent's decision making (named $Intruder\_decide$ or $Robot\_i\_decide$ respectively, where i in the name denotes the number of the patrolling robot), and the other automaton (named $Intruder\_act$ and $Robot\_i\_act$, respectively) emulating the action performed to actuate the decision. The automata decide the order of exploration of waypoints. Intruder picks the next waypoint randomly from the list of adjacent nodes of its current location. The patrolling robots pick the next node by preferring the least visited ones. The automata $Intruder\_act$ and $Robot\_i\_act$ emulate moving from one waypoint to another. Moving takes time specified by an interval $[lb, ub]$, lb and ub denoting lower and upper timebounds respectively. The navigation graph is designed so that the distance between waypoints is approximately the same. This keeps the duration interval $[lb, ub]$ and the move duration approximation close to real moving duration.

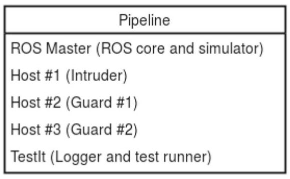
Since the robots' navigation is not perfect it can fail reaching the waypoint in some cases. This is represented in the model as a probabilistic transition. The probabilistic transitions of $Intruder\_act$ and $Robot\_i\_act$ automata are labelled with probability estimates. The probability of reaching the targeted waypoint is denoted by p. The probability estimates are normalized with value range $0-100$, so the failure of reaching target waypoint has estimate $100-p$. If the target waypoint is not reached, robot returns to its previous location and tries the same target again by changing its route. Intruder detection in the model is close to a real implemented detection mechanism. The intruder is visible to the patrol robot when the robot is oriented towards the next waypoint where the intruder is located, i.e., the distance between robots is not more than the distance between neighbour waypoints, and the intruder is visible in the robot's front view sector. This is modelled with conditional assignment $Detected = (NPos == Pos[0]?true : Detected)$, where $NPos$ is the variable modeling the next waypoint for a patrol to go and $Pos[0]$ models the current position of the intruder.

4.6 Test feasibility verification

Supposing that for maximum code coverage, both the robot navigation and intruder detection scenarios have to be represented in the model the verification property should express the reachability condition that it is always the case that at least one of the robots eventually detects the intruder. By referring to the model global variable $Detected$, this can be expressed by TCTL formula $A <> Detected$. The variable $Detected$ is updated to $true$ in the model whenever any of patrolling robots satisfies the detection condition. The verification experiments show that this test case is feasible for the navigation graph that includes 86 nodes and patrols implement described search strategy when there are at least three patrol robots and if the robots are at least three times faster than the intruder. The robot's navigation failure probability should be lower than 5 percent. Weaker verification condition $E <> Detected$ that is valid if there exists at least one of such behavior where the intruder is located has been proved under more relaxed conditions where only two robots with equal speeds are finding the intruder in the building. If further action is not needed to be covered in the test case, this proof is sufficient for continuing with the test case optimization. Keeping in mind, the test is feasible under given constraints, further test optimization steps are targeted to reduce the test length by allocating probabilistic gain functions to the test model that guide the test run towards the goal along the optimal test path.

4.7 TestIt configuration

TestIt configuration for testing the SUT is specified in a separate source code repository[3] as encapsulating the testing configuration is preferable to keep both repositories (i.e., the SUT repository and TestIt configuration repository) clean and concise. The TestIt configuration itself essentially consists of the testing infrastructure control configuration, SUT launch and test launch parameter specification. TestIt SUT launch is configured to start the full software stack and bring all robots (i.e., patrol robots and intruder robot) online and ready to receive navigation goals. After a short duration, the patrol planning as well as detection algorithm (the SUT component in this case study) is also started.


Pipeline

ROS Master (ROS core and simulator)
Host #1 (Intruder)
Host #2 (Guard #1)
Host #3 (Guard #2)
TestIt (Logger and test runner)

Figure 22: Test pipeline configuration.

To demonstrate our approaches scalability we have used the AWS cloud to create four pipelines which can run the tests and simulations in parallel. The configuration is shown in

[3]https://gitlab.com/GertKanter/testit-patrol-aws

Figure 10. Each pipeline consists of a SUT ROS master server (including the simulator), an intruder navigation stack server, two guard navigation stack and patrol algorithm servers and a TestIt logger and test runner server as seen in Figure 22.

4.8 Case study results

As discussed in Section 3.4.4, TestIt is designed to run in two modes. Exploratory mode is used to generate data for test scenario optimization. An example of exploratory phase is shown in Figure 23 visualized using RViz tool. In the figure, the trajectories of the robots are visualized as a collection of odometry arrows. In this particular example, the intruder (denoted as yellow) was operating in a limited area in the top left. The trajectories of the patrol robots are denoted in red and blue.

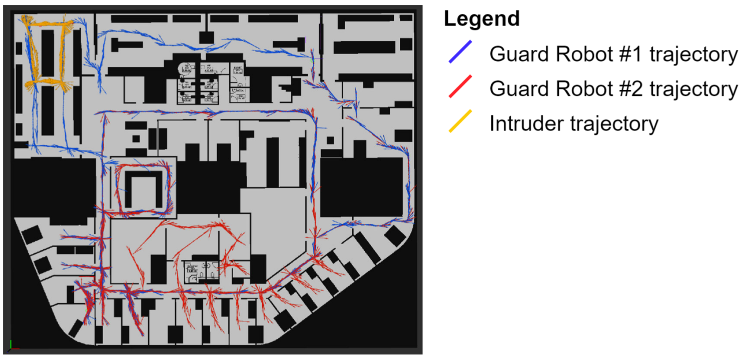


Figure 23: Test execution visualization.

After a sufficient amount of data has been logged the optimized scenario can be extracted from the log. The amount of log data required for effective optimization varies by use case. Optimization requires that the logs contain an event of interest (i.e., a situation where increased code coverage is achieved) in order to be useful. In our case, the logger captured a high coverage event near the starting point of one patrol robot (denoted as a red node in Figure 19). The optimization algorithm was used on the test runner and the algorithm guided the robot to achieve full relative coverage of the $patrol\_detector$ node. The term relative coverage refers to the coverage based on what have been captured in the logs. This means that the sections of code that have not been executed during logging remain unknown to the optimization algorithm.

The coverage plot is presented in Figure 24. The plot presents three cases: worst-case scenario, random scenario and optimized scenario coverage strategy. The test time was set to 500 seconds to allow the test runner sufficient time for guiding the intruder into different states. The number of states traversed ranged from 17 to 22.

In the worst-case, the test runner always chooses the worst possible next input that gains the least coverage. This is achieved by continually traversing the farthest left top corner edges on the navigation map in a loop. Based on the logs, there is no detection event occurring in that region and therefore the relative code coverage remains constant.

The random strategy test runner picks the next input randomly from the known alternatives. The result shown is the average case in multiple runs. As can be seen from the figure, the random strategy achieves marginally improved code coverage compared to the worst-case strategy.

In order to attain the best coverage as rapidly as possible we employ the optimization strategy in test generation. The beginning of the tests is similar as the main loop of the

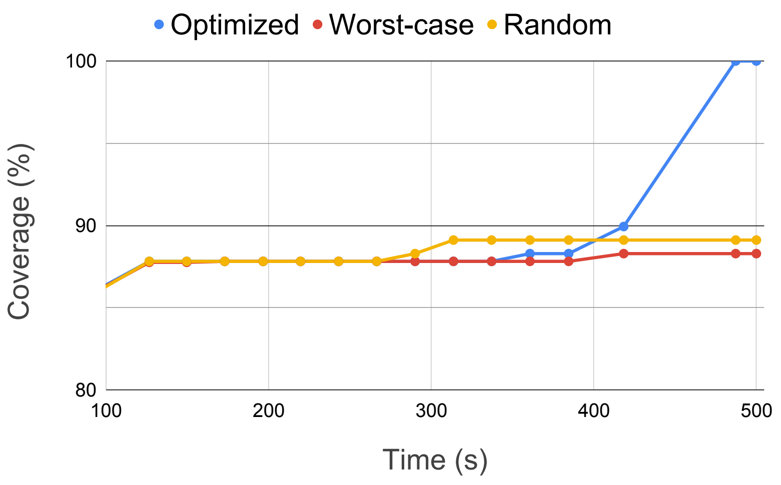


Figure 24: Relative code coverage of the SUT.

SUT software is running and covered even without optimal control. The real benefit of the optimization algorithm becomes evident after some time has been given for the control algorithm to navigate the robot into position for the high code coverage event. The high code coverage event for this particular use case is when the intruder is detected by the guard robot and it takes time for both the guard robot and intruder to move to the same location. After sufficient time has been given for navigation we can see the strategy succeeding in achieving full relative coverage within the time limit when the other strategies fail.

The test runner takes into account live code coverage when executing the test which makes it very responsive to actual results. This means that each test can be unique as the test runner tries to achieve full code coverage based on the actual online learning results and adjusts the optimal input sequence to the SUT accordingly. This makes the approach robust and enables to determine when to restart the simulation if the test optimization algorithm detects that the code coverage can not be improved sufficiently and in reasonable time from the current state.

The online test runner is especially useful in high branching state space with low probability high code coverage events as the test runner can identify the correct sequence in advance. This can be witnessed also in the presented use case with limited branching factor. The random strategy failed to find the high code coverage events whereas the optimization algorithm succeeded.

In the case study we did not actively control the patrol robots to demonstrate that it is possible to test autonomous systems using the black-box approach without explicitly modeling their internal behavior. If the patrol robots would have been deterministically controllable the results would have been different and full coverage could have been achieved faster with the cost of drastic model state space increase.

4.9 Chapter summary

This chapter gave a detailed overview of the demonstration case study. Firstly, the SUT for the case study was described. This was followed by a description of the test scenario. A detailed account of the SUT model construction and test feasibility verification came next. Finally, the TestIt configuration was presented and finished up with outlining the test results.

5 Conclusion

5.1 Chapter overview

This chapter concludes the dissertation and outlines the future work.

5.2 Results

The testing framework described in the dissertation provides support for MBT of AS in three categories, these are

- testing methodology
- formal test verification and optimization
- testing process tooling.

The MBT process that is integrated into incremental and iterative AS development addresses the need for provably correct test development where each test development step starting from well-formed test model construction to test deployment and execution presumes formal correctness verification. The issue is addressed in the dissertation because the test results cannot be fully trusted without the test correctness has been verified. Properly generated tests decrease the uncertainty of test conclusions and the need for repetitive testing. Also, late design defect discovery and need for expensive defect correction decreases.

The formal techniques proposed in the dissertation address the following three issues in MBT:

- the verification of test development increments;
- automated model learning that is commonly acknowledged bottleneck when applying MB techniques in the AS development and testing;
- test optimization.

New contributions have been elaborated in each of these aspects. For verification of development increments the templates of correctness properties have been expressed in temporal logic TCTL and their verification technique using Uppaal model checker has been demonstrated.

An algorithm for learning a subclass of Uppaal timed automata models from system and its environment interaction logs has been developed. A probabilistic online optimization algorithm is proposed that generates an optimal with respect to code coverage and other user-defined parameters gain test paths.

To provide automation support to the testing process TestIt toolkit has been developed. TestIt toolkit's primary focus is model-based testing of autonomous multi-robot systems to improve software quality assurance in various applications. The main novelty of the presented toolkit is the open and scalable multi-pipeline architecture that enables incorporation of test development and execution tools from various vendors. The second main contribution is adaptive test optimization technique that takes advantage of the proposed multi-pipeline architecture where testing threads can communicate the test runs based on their cooperatively collected test performance data. The usability of TestIt for test generation, its validation and optimization in autonomous navigation context is demonstrated using a robotic intruder detection system case study.

The framework has been validated also in industrial AS development projects in collaboration with Norcar AB and AS Milrem. The proposed framework helped improve the quality assurance of the autonomous robot software in these projects. The results show that the proposed framework and TestIt toolkit add value to the quality assurance workflow in autonomous multi-robot systems development.

5.3 Future work

TestIt toolkit can be improved in multiple aspects. One of the ongoing improvement works is adding an online state space exploration feature to TestIt. This functionality adds the possibility to learn the state transitions automatically without extra modeling overhead. This process uses the same logging facility as the rest of TestIt but these logs are generated by an exploration algorithm which aims to map the state space sequences that the SUT is able to complete. The resulting state space traversal automata can be used as input to test scenario optimization algorithm described in this dissertation. Combining automatic automata learning with TestIt will reduce the manual effort required to configure TestIt and lowers the modeling qualification requirements for using TestIt. The overall goal of TestIt is to minimize the manual work for testing in order to provide maximum test cost savings.

The second ongoing work is adding a universal Dtron test adapter to TestIt. Currently, new Uppaal TA synchronizations need to be added by the user manually but this new addition will make it possible for TestIt to support essentially any ROS-based robot project without requiring manual development effort. This can be achieved by shifting the specification of the channels and channel types from adapter source code to a configuration file. There is no need to change anything in Uppaal TA as the integer type is usable for encoding the configuration specification (i.e., configuration specification selection is encoded as integers).

Another research direction worth further consideration is reward-based scenario exploration. Reward-based exploration of the state space could reduce optimal test scenario discovery time by exploring the state space based on system feedback which would guide the search algorithm to explore in the most promising state space regions. TestIt would benefit from an improved method of test scenario exploration as would any test case generation tool. One of the possible implementations for this functionality would be to use Multi-Agent Reinforcement Learning (MARL) [168]. The MARL approach is resource-intensive but could result in a fully-automatic testing flow for the robotics software engineer.

References

[1] G. Kanter and J. Vain. Model-based testing of autonomous robots using TestIt. *Journal of Reliable Intelligent Environments*, 6(1):1–17, 2020.

[2] G. Kanter and J. Vain. Testit: an open-source scalable long-term autonomy testing toolkit for ros. In *Proceedings of the 10th International Conference Dependable Systems, Services and Technologies, DESSERT'2019*, pages 45–50, 2019.

[3] G. Kanter, J. Vain, S. Srinivasan, and S. Ramaswamy. Provably correct configuration management of precision feeding in agriculture4.0. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 1631–1637, 2019.

[4] J. Vain, G. Kanter, and A. Anier. Learning timed automata from interaction traces. In *14th IFAC Symposium on Analysis, Design, and Evaluation of Human Machine Systems, HMS 2019*, volume 52-19, pages 205–210, 2019.

[5] J. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ros packages: a mobile robot case study. In *2015 IEEE European Conference on Mobile Robots*, pages 1–7. IEEE, 2015.

[6] J. Vain, G. Kanter, and S. Srinivasan. Model based testing of distributed time critical systems. In *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 99–105. IEEE, 2017.

[7] What is a container?, 2020. `https://www.docker.com/resources/what-container`, Accessed on 2020-05-09.

[8] Rajeev Alur. *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[9] A. Gautam and S. Mohan. A review of research in multi-robot systems. In *2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS)*, pages 1–5, Aug 2012.

[10] Vladimir Lumelsky. *Sensing, Intelligence, Motion: How Robots and Humans Move in an Unstructured World*. 01 2006.

[11] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2016.

[12] R. Bangia. *Dictionary of Information Technology*. Laxmi Publications Pvt Limited, 2010.

[13] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[14] S. K. Khaitan and J. D. McCalley. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal*, 9(2):350–365, 2015.

[15] Iris F.A. Vis. Survey of research in the design and control of automated guided vehicle systems. *European Journal of Operational Research*, 170(3):677 – 709, 2006.

[16] Gregory Dudek, Michael Jenkin, and Evangelos Milios. A taxonomy of multirobot systems. *Robot teams: From diversity to polymorphism*, pages 3–22, 2002.

[17] List of self-driving car fatalities, 2020. `https://en.wikipedia.org/wiki/List_of_self-driving_car_fatalities`, Accessed on 2020-05-09.

[18] Top 10 strategic technology trends for 2019: Autonomous things, 2019. `https://www.gartner.com/en/documents/3904571/top-10-strategic-technology-trends-for-2019-autonomous-t`, Accessed on 2020-05-09.

[19] T. Linz. Testing autonomous systems. In S. Goericke, editor, *The Future of Software Quality Assurance*, pages 61–75, Cham, 2020. Springer International Publishing.

[20] J. Bosch. Speed, data, and ecosystems: The future of software engineering. *IEEE Software*, 33(1):82–88, Jan 2016.

[21] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.*, 52(5), September 2019.

[22] John Jeremiah. Survey: Is agile the new norm?, 2015. `https://techbeacon.com/app-dev-testing/survey-agile-new-norm`, Accessed on 2020-05-09.

[23] Estonian Research Information System - "Private sector (Estonia)" project LEP18082IT, 2018. `https://www.etis.ee/Portal/Projects/Display/3618849e-98af-4ca6-8942-ea713c6d0312?lang=ENG`, Accessed on 2020-05-09.

[24] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[25] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley Publishing, 3rd edition, 2011.

[26] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996.

[27] Albert Benveniste, Benoit Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim Guldstrand Larsen. Contracts for System Design. Research Report RR-8147, INRIA, November 2012.

[28] M. Kläs, T. Bauer, A. Dereani, T. Söderqvist, and P. Helle. A large-scale technology evaluation study: Effects of model-based analysis and testing. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 119–128, May 2015.

[29] M. Felderer and A. Beer. Estimating the return on investment of defect taxonomy supported system testing in industrial projects. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 426–430, Sep. 2012.

[30] T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand. Can robot navigation bugs be found in simulation? an exploratory study. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 150–159, 2017.

[31] A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley. A study on challenges of testing robotic systems. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020*. IEEE, In press.

[32] C. Robert. First insights into testing autonomous robot in virtual worlds. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 112–115, 2017.

[33] James Arnold and Rob Alexander. Testing autonomous robot control software using procedural content generation. In *Proceedings of the 32nd International Conference on Computer Safety, Reliability, and Security - Volume 8153*, SAFECOMP 2013, page 33–44, Berlin, Heidelberg, 2013. Springer-Verlag.

[34] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 318–328, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Franz Wotawa, Bernhard Peischl, Florian Klück, and Mihai Nica. Quality assurance methodologies for automated driving. *Elektrotechnik und Informationstechnik*, 135(4):322–327, 2018.

[36] Sandeep K.S. Gupta, Tridib Mukherjee, Georgios Varsamopoulos, and Ayan Banerjee. Research directions in energy-sustainable cyber–physical systems. *Sustainable Computing: Informatics and Systems*, 1(1):57 – 74, 2011.

[37] Dejanira Araiza-Illan, Anthony G. Pipe, and Kerstin Eder. Model-based test generation for robotic software: Automata versus belief-desire-intention agents, 2016.

[38] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: Tools and methodology. *Softw. Test. Verif. Reliab.*, 21(1):55–71, March 2011.

[39] Maurice Dawson, Darrell Burrell, Emad Rahim, and Stephen Brewster. Integrating software assurance into the software development life cycle (sdlc). *Journal of Information Systems Technology and Planning*, 3:49–53, 01 2010.

[40] Gabriela Nicolescu and Pieter J. Mosterman, editors. *Model-Based Design for Embedded Systems*. CRC Press. -1-4200-6784-2, Computational Analysis, Synthesis, and Design of Dynamic Systems. 1., 2010.

[41] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, Feb 2006.

[42] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139 – 155, 2015.

[43] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*, pages 731–736, June 2010.

[44] P. Helle, W. Schamai, and C. Strobel. Testing of autonomous systems - challenges and current state-of-the-art. *INCOSE International Symposium*, 26:571–584, 07 2016.

[45] D. Marijan, A. Gotlieb, and M. Kumar Ahuja. Challenges of testing machine learning based systems. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 101–102, 2019.

[46] F. Ingrand. Recent trends in formal validation and verification of autonomous robots software. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 321–328, 2019.

[47] P. Duan, Y. Zhou, X. Gong, and B. Li. A systematic mapping study on the verification of cyber-physical systems. *IEEE Access*, 6:59043–59064, 2018.

[48] Xi Zheng and Christine Julien. Verification and validation in cyber physical systems: Research challenges and a way forward. In *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*, SEsCPS '15, page 15–18. IEEE Press, 2015.

[49] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[50] Chapter 2 - the pain and the gain. In Mark Utting and Bruno Legeard, editors, *Practical Model-Based Testing*, pages 19 – 57. Morgan Kaufmann, San Francisco, 2007.

[51] Maja J. Matarić. Designing and understanding adaptive group behavior. *Adaptive Behavior*, 4(1):51–80, 1995.

[52] C. Ronald Kube and Hong Zhang. Collective robotics: From social insects to robots. *Adaptive Behavior*, 2(2):189–218, 1993.

[53] Y. Uny Cao, Alex S. Fukunaga, and Andrew Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, Mar 1997.

[54] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In Erol Şahin and William M. Spears, editors, *Swarm Robotics*, pages 10–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[55] Tamio Arai, Enrico Pagello, Lynne E Parker, et al. Advances in multi-robot systems. *IEEE Transactions on robotics and automation*, 18(5):655–661, 2002.

[56] Stephen Hendrick Kaisler. *Software paradigms*. Wiley Online Library.

[57] Romana Gnatyk. Microservices vs monolith: which architecture is the best choice for your business?, 2018. `https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/`, Accessed on 2020-05-09.

[58] Robin Flygare and Anthon Holmqvist. Performance characteristics between monolithic and microservice-based systems, 2017.

[59] Elena Dubrova. *Introduction*, pages 1–4. Springer New York, New York, NY, 2013.

[60] Mirko Ferrati, Alessandro Settimi, Luca Muratore, Alberto Cardellino, Alessio Rocchi, Enrico Mingo Hoffman, Corrado Pavan, Dimitrios Kanoulas, Nikos G. Tsagarakis, Lorenzo Natale, and Lucia Pallottino. The walk-man robot software architecture. *Frontiers in Robotics and AI*, 3:25, 2016.

[61] Guillaume Walck, Ugo Cupcic, Toni Oliver Duran, and Véronique Perdereau. A Case Study of ROS Software Re-usability for Dexterous In-Hand Manipulation. *Journal of Software Engineering for Robotics*, 5(1):36–47, 2014.

[62] Gergely Magyar, Peter Sincak, and Zoltán Krizsán. Comparison study of robotic middleware for robotic applications. *Advances in Intelligent Systems and Computing*, 316:121–128, 01 2015.

[63] What is ROS?, 2018. `http://wiki.ros.org/ROS/Introduction`, Accessed on 2020-05-09.

[64] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[65] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 05 2012.

[66] Core components. `https://www.ros.org/core-components/`, Accessed on 2020-05-09.

[67] Tully Foote. Community metrics report, 2019. `http://download.ros.org/downloads/metrics/metrics-report-2019-07.pdf`, Accessed on 2020-05-09.

[68] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*, page to appear, 2020.

[69] A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ros computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69, 2019.

[70] Chapter 1 - the challenge. In Mark Utting and Bruno Legeard, editors, *Practical Model-Based Testing*, pages 1 – 18. Morgan Kaufmann, San Francisco, 2007.

[71] Robert V Binder, Bruno Legeard, and Anne Kramer. Model-based testing: Where does it stand? *Queue*, 13(1):40–48, 2014.

[72] Matti Vuori. Model-based testing in modern agile software development - how to integrate it into the development process?, 2014. `http://www.cs.tut.fi/~swtest/atac/ATAC_report_MBT_in_modern_agile_development.pdf`, Accessed on 2020-05-09.

[73] Sanjit A. Seshia, Natasha Sharygina, and Stavros Tripakis. *Modeling for Verification*, pages 75–105. Springer International Publishing, Cham, 2018.

[74] Dennis Dams and Orna Grumberg. *Abstraction and Abstraction Refinement*, pages 385–419. Springer International Publishing, Cham, 2018.

[75] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. *Model Learning and Model-Based Testing*, pages 74–100. Springer International Publishing, Cham, 2018.

[76] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, pages 175–189, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[77] Axel van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 147–159, New York, NY, USA, 2000. Association for Computing Machinery.

[78] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

[79] Mohd Azizi Abdul Rahman, Katsuhiro Mayama, Takahiro Takasu, Akira Yasuda, and Makoto Mizukawa. Model-driven development of intelligent mobile robot using systems modeling language (sysml). In *Mobile Robots*, chapter 1. IntechOpen, Rijeka, 2011.

[80] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, *Lecture Notes in Computer Science vol. 3098*, pages 87–124. Springer-Verlag, 2004.

[81] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Křetínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 comparison of tools for the analysis of quantitative formal models. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 69–92, Cham, 2019. Springer International Publishing.

[82] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In Khaled El-Fakih, Gerassimos Barlas, and Nina Yevtushenko, editors, *Testing Software and Systems*, pages 194–207, Cham, 2015. Springer International Publishing.

[83] Aaron Kane, Thomas Fuhrman, and Philip Koopman. Monitor based oracles for cyber-physical system testing: Practical experience report. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, page 148–155, USA, 2014. IEEE Computer Society.

[84] Teck Ping Khoo. Model based testing of cyber-physical systems. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering*, pages 423–426, Cham, 2018. Springer International Publishing.

[85] Dejanira Araiza-Illan, Anthony G. Pipe, and Kerstin Eder. Intelligent agent-based stimulation for testing robotic software in human-robot interactions. In *Proceedings of the 3rd Workshop on Model-Driven Robot Software Engineering*, MORSE '16, page 9–16, New York, NY, USA, 2016. Association for Computing Machinery.

[86] Nidhi Kalra and Susan Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 12 2016.

[87] Ali Paikan, Silvio Traversaro, Francesco Nori, and Lorenzo Natale. A generic testing framework for test driven development of robotic systems. In Jan Hodicky, editor, *Modelling and Simulation for Autonomous Systems*, pages 216–225, Cham, 2015. Springer International Publishing.

[88] Jyotirmoy V. Deshmukh, Marko Horvat, Xiaoqing Jin, Rupak Majumdar, and Vinayak S. Prabhu. Testing cyber-physical systems through bayesian optimization. *ACM Trans. Embedded Comput. Syst.*, 16:170:1–170:18, 2017.

[89] Houssam Abbas, Matthew O'Kelly, Alena Rodionova, and Rahul Mangharam. Safe at any speed: A simulation-based test harness for autonomous vehicles. In Roger Chamberlain, Walid Taha, and Martin Törngren, editors, *Cyber Physical Systems. Design, Modeling, and Evaluation*, pages 94–106, Cham, 2019. Springer International Publishing.

[90] Matthew O' Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C Duchi. Scalable end-to-end autonomous vehicle testing via rare-event simulation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 9827–9838. Curran Associates, Inc., 2018.

[91] S. Liu, J. Tang, C. Wang, Q. Wang, and J. Gaudiot. A unified cloud platform for autonomous driving. *Computer*, 50(12):42–49, December 2017.

[92] C. S. Timperley, A. Afzal, D. S. Katz, J. M. Hernandez, and C. Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 331–342, April 2018.

[93] Smart continuous integration, 2020. `http://www.testributor.com/`, Accessed on 2020-05-09.

[94] The automated test framework (atf), 2020. `https://github.com/floweisshardt/atf/`, Accessed on 2020-05-09.

[95] Graphwalker, an open-source model-based testing tool, 2020. `http://graphwalker.github.io/`, Accessed on 2020-05-09.

[96] fmbt, 2020. `https://01.org/fmbt`, Accessed on 2020-05-09.

[97] Boost your test automation for devops!, 2019. `https://4test.io/`, Accessed on 2020-05-09.

[98] Jsxm, 2014. `http://www.jsxm.org/`, Accessed on 2020-05-09.

[99] Dimitris Dranidis, Konstantinos Bratanis, and Florentin Ipate. Jsxm: A tool for automated test generation. In George Eleftherakis, Mike Hinchey, and Mike Holcombe, editors, *Software Engineering and Formal Methods*, pages 352–366, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[100] Modbat - a model-based tester, 2019. `https://people.kth.se/~artho/modbat/`, Accessed on 2020-05-09.

[101] Momut, 2020. `https://momut.org/`, Accessed on 2020-05-09.

[102] Osmo mbt tool, 2020. `https://github.com/mukatee/osmo`, Accessed on 2020-05-09.

[103] Teemu Kanstrén and Olli-Pekka Puolitaival. Using built-in domain-specific modeling support to guide model-based test generation. *Electronic Proceedings in Theoretical Computer Science*, 80, 02 2012.

[104] Tcases: A model-based test case generator, 2020. `https://github.com/Cornutum/tcases`, Accessed on 2020-05-09.

[105] J. Tretmans. *On the Existence of Practical Testers*, pages 87–106. Springer International Publishing, Cham, 2017.

[106] Construction and analysis of distributed processes - software tools for designing reliable protocols and systems, 2020. `https://cadp.inria.fr/`, Accessed on 2020-05-09.

[107] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, Apr 2013.

[108] Distributed real-time embedded analysis method - dream, 2009. `http://dre.sourceforge.net/`, Accessed on 2020-05-09.

[109] Guillaume Gardey, Didier Lime, Morgan Magnin, et al. Romeo: A tool for analyzing time petri nets. In *International Conference on Computer Aided Verification*, pages 418–423. Springer, 2005.

[110] David Freedman. *Markov chains*. Springer-Verlag, New York-Berlin, 1983. Corrected reprint of the 1971 original.

[111] Ahmed Tamrawi, Kang Gui, and Suresh Kothari. Event-flow graphs for efficient path-sensitive analyses. *ArXiv*, abs/1404.1279, 2014.

[112] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. 01 2010.

[113] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[114] Unified modeling language, 2020. `https://www.omg.org/spec/UML/About-UML/`, Accessed on 2020-05-09.

[115] J. Srba. Comparing the expressiveness of timed automata and timed extensions of petri nets. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems*, pages 15–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[116] Gerd Behrmann, Alexandre David, and Kim Larsen. A tutorial on uppaal. volume 3185, pages 200–236, 01 2004.

[117] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal Methods and Testing*, 2008.

[118] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[119] The spread toolkit, 2019. `http://www.spread.org/`, Accessed on 2020-05-09.

[120] Kim Guldstrand Larsen. Dependable and optimal cyber-physical systems. In *SOFSEM 2017*, volume 10139 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 3–10, Germany, 2017. Springer.

[121] Kim Guldstrand Larsen. Validation, synthesis and optimization for cyber-physical systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017 held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Proceedings*, volume 10205 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 3–20, Germany, 2017. Springer. 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017 held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017 ; Conference date: 22-04-2017 Through 29-04-2017.

[122] Kim Larsen, Marius Mikučionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: An industrial case study. pages 299–306, 01 2005.

[123] J. Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, Dept. of Information Technology, 2002.

[124] Paula Herber. A framework for automated hw/sw co-verification of systemc designs using timed automata. *it - Information Technology*, 54, 11 2012.

[125] Junaid Iqbal, Dragos Truscan, Jüri Vain, and Ivan Porres. Reconstructing timed symbolic traces from rtioco-based timed test sequences using backward-induction. In *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*, ECBS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[126] Craig Larman and Victor Basili. Iterative and incremental development: A brief history. *Computer*, 36:47 – 56, 07 2003.

[127] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & systems engineering process and tools for the development of autonomous driving intelligence. *JACIC*, 4:1158–1174, 09 2007.

[128] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87*, 1970.

[129] Ralph Jeffords, Constance Heitmeyer, Myla Archer, and Elizabeth Leonard. A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 173–189, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[130] Xin Zhou, Xiaodong Gou, Tingting Huang, and Shunkun Yang. Review on testing of cyber physical systems: Methods and testbeds. *IEEE Access*, 6:52179–52194, 2018.

[131] roslint, 2017. `http://wiki.ros.org/roslint`, Accessed on 2020-05-09.

[132] A. Santos, A. Cunha, N. Macedo, and C. Lourenço. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, Oct 2016.

[133] Docker, 2017. `https://www.docker.com`, Accessed on 2020-05-09.

[134] Kim Guldstrand Larsen, Florian Lorber, and Brian Nielsen. 20 years of real real time model validation. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 22–36, Cham, 2018. Springer International Publishing.

[135] Juhan Ernits, Rivo Roo, Jonathan Jacky, and Margus Veanes. Model-based testing of web applications using nmodel. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems*, pages 211–216, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[136] T.Y. Chen and Man Lau. On the divide-and-conquer approach towards test suite reduction. *Information Sciences*, 152:89–119, 06 2003.

[137] Ros-industrial quality-assured robot software components, 2020. `https://rosin-project.eu/`, Accessed on 2020-05-09.

[138] Quality assurance process and community management in ros, 2017. `http://rosin-project.eu/wp-content/uploads/D3.1-Quality-Assurance-Process-and-Community-Management-in-ROS.pdf`, Accessed on 2020-05-09.

[139] C. Pinciroli, V. Trianni, R. OǴrady, and G. Pini et al. Argos: A modular, multi-engine simulator for heterogeneous swarm robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5027–5034, 2011.

[140] Richard Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2):189–208, Dec 2008.

[141] Gazebo - robot simulation made easy, 2020. `http://gazebosim.org/`, Accessed on 2020-05-09.

[142] Carla simulator, 2020. `https://github.com/carla-simulator/carla`, Accessed on 2020-05-09.

[143] Airsim simulator, 2020. `https://microsoft.github.io/AirSim/`, Accessed on 2020-05-09.

[144] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[145] Jenkins, 2020. `https://www.jenkins.io/`, Accessed on 2020-05-09.

[146] Yaml: Yaml ain't markup language, 2020. `https://yaml.org/`, Accessed on 2020-05-09.

[147] Amazon web services, 2020. `https://aws.amazon.com/`, Accessed on 2020-05-09.

[148] Solve more with google cloud, 2020. `https://cloud.google.com/`, Accessed on 2020-05-09.

[149] Microsoft azure: Cloud computing services, 2020. `https://azure.microsoft.com/en-us/`, Accessed on 2020-05-09.

[150] GCC - program instrumentation options, 2020. `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`, Accessed on 2020-05-09.

[151] Python - increase test coverage, 2020. `https://devguide.python.org/coverage/`, Accessed on 2020-05-09.

[152] Howto: Dumping gcov data at runtime - simple example, 2011. `http://www.osadl.org/Dumping-gcov-data-at-runtime-simple-ex.online-coverage-analysis.0.html`, Accessed on 2020-05-09.

[153] The coverage class, 2019. `https://coverage.readthedocs.io/en/v4.5.x/api_coverage.html`, Accessed on 2020-05-09.

[154] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[155] Aivo Anier and Jüri Vain. Model based continual planning and control for assistive robots. *HealthInf 2012*, (Proceedings of the International Conference on Health Informatics):382–385, 2012.

[156] Vladimír Štill, Petr Ročkai, and Jiří Barnat. Divine: Explicit-state ltl model checker. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 920–922, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[157] Z3 theorem prover, 2020. `https://github.com/z3prover/z3`, Accessed on 2020-05-09.

[158] H. Barbosa, A. Reynolds, D. El Ouraoui, C. Tinelli, and C. Barrett. Extending smt solvers to higher-order logic. In P Fontaine, editor, *Automated Deduction – CADE 27*. CADE 2019. Lecture Notes in Computer Science, vol 11716. Springer, Cham, 2019.

[159] N. Hawes, C. Burbridge, F. Jovan, L. Kunze, B. Lacerda, L. Mudrova, J. Young, J. Wyatt, D. Hebesberger, T. Kortner, R. Ambrus, N. Bore, J. Folkesson, P. Jensfelt, L. Beyer, A. Hermans, B. Leibe, A. Aldoma, T. Faulhammer, M. Zillich, M. Vincze, E. Chinellato, M. Al-Omari, P. Duckworth, Y. Gatsoulis, D. C. Hogg, A. G. Cohn, C. Dondrup, J. Pulido Fentanes, T. Krajnik, J. M. Santos, T. Duckett, and M. Hanheide. The strands project: Long-term autonomy in everyday environments. *IEEE Robotics Automation Magazine*, 24(3):146–156, Sep. 2017.

[160] Smach - state machine, 2018. `http://wiki.ros.org/smach`, Accessed on 2020-05-09.

[161] Behaviortree.cpp, 2020. `https://github.com/BehaviorTree/BehaviorTree.CPP`, Accessed on 2020-05-09.

[162] The stage simulator, 2019. `https://github.com/rtv/Stage`, Accessed on 2020-05-09.

[163] Juri Vain, Marko Kääramees, and Maili Markvardt. Online testing of nondeterministic systems with the reactive planning tester. *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, pages 113–150, 01 2011.

[164] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 2 edition, 2012.

[165] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[166] move_base, 2018. `http://wiki.ros.org/move_base`, Accessed on 2020-05-09.

[167] amcl - adaptive (or kld-sampling) monte carlo localization, 2019. `http://wiki.ros.org/amcl`, Accessed on 2020-05-09.

[168] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. In *Innovations in multi-agent systems and applications-1*, pages 183–221. Springer, 2010.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Jüri Vain for his guidance and support. I also thank my colleagues at Tallinn University of Technology for their support and advice.

Abstract
Model-Based Testing Framework for Autonomous Multi-Robot Systems

Alleviating the need for increased quality assurance in the face of rapidly growing number of autonomous robots, this dissertation presents a model-based testing framework including TestIt toolkit. The main goal of the framework is to provide methods and tools for automated model-based testing of autonomous multi-robot systems to verify correct behavior in various applications. The main contribution of the dissertation is a testing toolkit TestIt which features several novelties. Firstly, the open and scalable multi-pipeline architecture that enables incorporation of test development and execution tools from various vendors. The second main contribution is the adaptive test optimization technique that takes advantage of the proposed multi-pipeline architecture where testing threads can communicate and coordinate the test runs based on their cooperatively collected test performance data. The usability of TestIt for test generation, testing process performance improvement and optimization in multi-robot autonomous navigation context is demonstrated using a robotic security system case study.

Additionally, a provably correct test development process within an iterative and incremental software development cycle is detailed. The results show that the proposed incremental validation minimizes the validation time, design space exploration and cost during early stages of design, rather than during operations which could lead to significant cost increase.

The results of the framework have been validated in industry projects and have shown to add value to the quality assurance process.

Kokkuvõte
Mudelipõhine testimisraamistik autonoomsetele multirobotsüsteemidele

Seoses autonoomsete robotsüsteemide kasvava levikuga süveneb vajadus tagada nende funktsionaalsus, töökindlus ja ohutus valdkondades, kus robotid töötavad ja liiguvad vahetult inimkeskkonnas. Käesolev dissertatsioon esitleb mudelipõhise testimise raamistikku ning testi automatiseerimise töövahendeid autonoomsete multirobotsüsteemide mudelipõhiseks testimiseks. Töö põhitulemuseks on testimistööriist TestIt, mis pakub mitmeid uusi lahendusi. Esiteks, avatud ja skaleeritav multikonveier arhitektuur, mis võimaldab erinevate osapoolte testimistööriistade integreerimist arendusprotsessis. Teiseks, loodud on multikonveier arhitektuuril põhinev adaptiivne optimeerimisalgoritm, mis võimaldab lõimede omavahelise andmevahetuse tulemusi kasutada testistsenaariumite optimeerimiseks. TestIt tööriista võimalusi ja eeliseid testijuhtude genereerimisel, testimisprotsessi jõudluse tõstmisel ja optimeerimisel on demonstreeritud robotturvasüsteemi rakenduses multirobotsüsteemi autonoomse navigatsiooni testimise näitel. Lisaks eelnevale on dissertatsioonis esitatud testide tõestatavalt korrektne arendusprotsess ja selle integreerimine iteratiivsesse ja inkrementaalsesse tarkvaraarendustsüklisse. Töö tulemused näitavad, et pakutud inkrementaalne valideerimine võimaldab vähendada disaini valideerimisaega ja disaini lahendusteruumi analüüsi mahtu juba varajases arendusstaadiumis, vältimaks vigade kandumist arenduse lõpufaasi, kus vigade avastamine ja parandamine on oluliselt kulukam. Töö tulemused on valideeritud tööstuslike robotiprojektide raames ja tulemused on näidanud, et väljatöötatud raamistik annab robotsüstemide kvaliteedi tagamise protsessile olulist lisaväärtust.

Appendix 1

Publication I

G. Kanter and J. Vain. Model-based testing of autonomous robots using TestIt. *Journal of Reliable Intelligent Environments*, 6(1):1–17, 2020

Journal of Reliable Intelligent Environments
https://doi.org/10.1007/s40860-019-00095-w

ORIGINAL ARTICLE

Model-based testing of autonomous robots using TestIt

Gert Kanter[1] · **Jüri Vain**[1]

Received: 3 June 2019 / Accepted: 17 December 2019

Abstract
This paper presents a testing toolkit named TestIt. Its main goal is to provide tools for automated model-based testing of autonomous multi-robot systems to verify long-term autonomy in various applications including those of smart city environments and smart buildings. The main novelty of the presented toolkit is the open and scalable multi-pipeline architecture that enables incorporation of test development and execution tools from various vendors. The second main contribution is adaptive test optimization technique that takes advantage of the proposed multi-pipeline architecture where testing threads can communicate and coordinate the test runs based on their cooperatively collected test performance data. The usability of TestIt for test generation, testing process performance improvement and optimization in multi-robot autonomous navigation context is demonstrated using a smart building robotic security system case study.

Keywords Autonomous robotics · Robot operating system · Integration testing · Model-based testing · Smart building · Simulation

1 Introduction

New security technologies such as AI-supported data fusion, video analytics and application of mobile sensor platforms have provided substantial quality improvement of security-related services in smart city and smart building environments [2]. Despite recent advances in security technology, the weakest link in the security assurance loop (sensor/camera data capture, monitoring data fusion, malicious act threat recognition, decision making, prevention/counter action) is human involvement. As the experiment with autonomous robot security guard Knightscope's K5 shows, one promising way of raising the level of security systems' reliability and autonomy is replacing human personnel, e.g. patrols and remote operators with mobile guardian robots which instead of remote surveillance can guarantee required level of presence, situation awareness and timely reaction to events in the place.

Since security systems must be safe and secure themselves, it means the services provided by them must be validated and verified against the requirements set by the given application. The quality of service (QoS) characteristics studied in this paper include in the first place functional correctness, performance stability, reliability and scalability. For instance, the quality of the security system enforced with patrolling robots should ensure that the robots collaboration satisfies the criteria necessary for successful mission completion under given operation conditions (e.g. dimmed light, accessibility to rooms, presence of moving obstacles).

In this paper, we provide a novel testing framework TestIt for multi-robot surveillance system quality testing in the simulated operation environment. Since setting up a full-scale field test environment is expensive and running tests in there is time consuming, the goal of this work is to provide easier, faster and cheaper validation of system quality than traditional full-fledged field tests allow. he novel feature of this approach is the capability of runtime optimization of tests by using live feedback from the SUT. An edge labeled with a name and postfix symbol "!" synchronizes with an edge or edges of all other automata that are labelled with the same name and postfix symbol "?".

The autonomous patrolling robot systems such as autonomous robots in general are designed to operate in highly dynamic and unpredictable environments. Therefore, it is difficult to assure that these autonomous systems function safely in all possible situations. Moreover, even capturing such high-dimensional context data that characterize real opera-

Gert Kanter
gert.kanter@taltech.ee

Jüri Vain
juri.vain@taltech.ee

1 Department of Software Science, Tallinn University of Technology, Akadeemia tee 15a, Tallinn, Estonia

tion environments in all emerging border cases is exceedingly difficult. Therefore, to accelerate the validation of the algorithms used in autonomous robots, simulation is used to generate scenarios to ensure correct behaviour under large variety of constraints.

Simulation is beneficial not only because it is inexpensive as it does not require a physical robot, but also because it is scalable. Testing on multiple physical robots requires multiple test sites and evaluation systems which may constitute a substantial part of development costs. To conserve time and validate multiple simulated scenarios at the same time, these simulations can be run in parallel. For the simulation to be well integrated into the software production workflow, also the robot's software stack should support such integration. This support is present in robot operating system (ROS)-based robots' software thanks to the design principles of ROS [18]. ROS uses a publisher–subscriber communication model which allows easy component interoperability. This interoperability enables easily switching out physical hardware with simulation without the software being aware of the difference.

As an example of a combined testing solution where sensor data emitted by stationary security system are used for triggering and planning the mobile patrolling robots inspection missions, we introduce a case study where a team of collaborating robots locates and captures an intruder in the office building floor after the stationary monitoring system has released the alarm of unauthorized movement in the guarded zone.

The case study demonstrates the advantages of the proposed collaborative robot action testing method by providing high coverage of the simulated world state space as well as the coverage of the robots' software. Also, we show that the high variability of test scenarios needed for testing long-term autonomy of the multi-robot system in dynamic environment can be encoded in the test models, which in turn minimizes the need for human intervention in the course of testing process and, thus, allows improving the overall testing process performance.

The rest of the paper is structured as follows. In Sect. 2, we position our work with respect to testing toolkits from the point of view of simulation-based verification of multi-robot applications. Section 3 presents preliminaries that explain the methods, principles and underlying formal basis of our model-based testing approach. Section 4 outlines the TestIt architecture and workflows supported by it. In Sect. 5, the practical aspects of configuring TestIt for test execution are explained. In Sect. 6, the test development, optimization and execution process using TestIt is illustrated using the robotic intruder detection system case study. Finally, conclusions and future work are discussed.

2 Related work

TestIt toolkit is designed for testing systems that can be broadly classified as cyber physical systems. The authors of [3] state that CPS are typically so complex that solving their analysis and optimization problem analytically by examining the system dynamics is not feasible. An extensive survey on hybrid dynamic CPS monitoring techniques is presented in [4] with the focus on runtime verification. This survey is targeted to unification of tool-supported monitoring methodologies which is motivated by the need for an alternative to CPS exhaustive verification approaches. The main limitation of exhaustive verification techniques is their limited scalability for CPS. Our work has different focus, instead of passive monitoring we use active testing while the scalability issues of the method are addressed by parallelizing the testing tasks and applying runtime optimization of testing threads based on on-the-fly gathered tests execution data.

One of the first attempts to provide a unified, tool-supported methodology for CPS testing and optimization is presented in [3], where the authors consider a black-box approach, to perform optimization by testing the input–output behaviour of the CPS. They claim their tool is the first CPS testing tool that supports Bayesian optimization. It is also the first attempt to employ fully automated dimensionality reduction techniques for CPS testing. Our model-based testing technique is similar to that of [3] in the sense that the SUT is defined as a black box, where only its interface behaviour is observable. Similarly, TestIt supports Bayesian optimization, but the underlying computational model Uppaal TA [5] is different from that used in [3]. Another principal difference is that instead of a single testing thread, we apply multi-thread approach which provides a clear performance advantage. In TestIt, each of the test threads is runtime optimizing to form the posterior distribution over the test driving objective function. It uses the data about test runs' performance which is collectively gathered by all threads. The authors of [6] provide a thorough overview of traditional and advanced simulation-based modelling, testing, and verification techniques applied in the field of embedded systems (subset of CPS). Like the approaches referred in [6], TestIt enables running tests against SUT simulation instead of physical SUT. TestIt is specific in the sense that it is not restricted to any particular tool or method listed in [6], but rather provides flexible infrastructure for configuring the methods and tools that are most suited for a given testing task. The goal of TestIt is to improve the performance by parallelization, flexible test infrastructure and efficient coordination of methods working in parallel.

Robot system-specific testing tools have more narrow focus and most of them are adjusted to ROS-based software testing. At present, there is not any known significant effort made in the development of model-based testing toolkits designed specifically for autonomous multi-robot systems and their application.

There are several general purpose testing tools that allow test parallelization, but they do not employ the model-based testing of ROS-based autonomous robots. An example of such a general purpose testing tool is a continuous integration (CI) platform Testributor.[1]

Testributor is an open-source continuous integration platform that reduces building times by slicing up the test suite and runs the slices in parallel. This platform is not specifically designed for testing robot operating system ROS software and it does not support model-based testing natively. In contrast, TestIt toolkit is designed to automatize testing workflow by using model-based testing and to perform test scenario optimization by learning from earlier executed tests. This approach increases testing efficiency compared to Testributor's simple test suite slicing, because slicing does not improve the tests themselves. TestIt aims to provide tools to improve the tests themselves as well as provide a highly scalable test execution environment.

An alternative to Testributor is an ROS-specific automated test framework (ATF)[2] which has been developed specifically for ROS applications. The ATF framework supports executing integration and system tests and running benchmarks. Unfortunately, it is not readily scalable and is designed to run on a single machine. The ATF framework also only provides the execution of the test suite, but offers no tools to create or optimize the test suite itself. Additionally, the project is currently not in active development.

The ROSIN project report[3] highlights the need for better QA practices to be adopted in ROS software development. One of the main issues is that the QA practices are not consistent across the various development streams (i.e. core, drivers and reusable packages). The report also indicates that the current utilization of CI service is not sufficient, because it is simply compiling and building the ROS projects. Their results show that the QA practice would be improved significantly by extending the CI service to run a collection of different kinds of code-scanning tools. To address these needs, TestIt has been designed to be easily integrated with the CI service. It also provides a framework that can be augmented with the aforementioned tools and bundled in a convenient package.

Another aspect the ROSIN report highlights is that although testing is regarded as critical in robotics, developers working on new components tend to focus more on creating the components rather than creating and setting up tests and gathering data based on simulation. As stated in the report, automated testing should compensate this practice by saving time and increasing software quality. TestIt aims to comply with these recommendations. It reduces time overhead by supporting automation of all testing phases, maximizing testing efficiency by using concurrent testing pipelines and minimizing testing time by learning from executed tests and optimizing test scenarios based on that.

[1] http://www.testributor.com/.

[2] https://github.com/floweisshardt/atf.

[3] http://rosin-project.eu/wp-content/uploads/D3.1-Quality-Assurance-Process-and-Community-Management-in-ROS.pdf.

3 Preliminaries

3.1 Model-based testing

Model-based testing (MBT) presumes the usage of models for specifying the expected behaviour of system under test (SUT) and the test purpose. The behaviours and model elements to be covered by the test are subject to test purpose specification. Both, the SUT model and test purpose specification are prerequisites for automatic test generation.

The advantages of MBT are experienced most clearly in integration and system level testing, where the functionality, timing, safety, security and other aspects of SUT are exposed in their most intertwined form. MBT focuses most often 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 controllable and observable respectively by test. The internal behaviour of the system is abstracted away in the model. The aim of black-box conformance testing, according to [7], is to check if the behaviour observable on the system interfaces conforms to that given in the system requirements specification.

During MBT, a tester executes selected test cases by running SUT in the test harness and emits a test verdict (pass, fail, inconclusive). The verdict shows test result in terms of conformance relation between SUT and the requirements model. A conformance relation used most often in MBT is input–output conformance (IOCO) introduced in [8]. For the behaviour of an implementation to be IOCO-correct it should respect 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.

3.2 Model-based testing with TestIt

TestIt is designed to work with a wide variety of tools and models that are used in the test development process. Thanks to the open architecture of TestIt, the user can pick and use

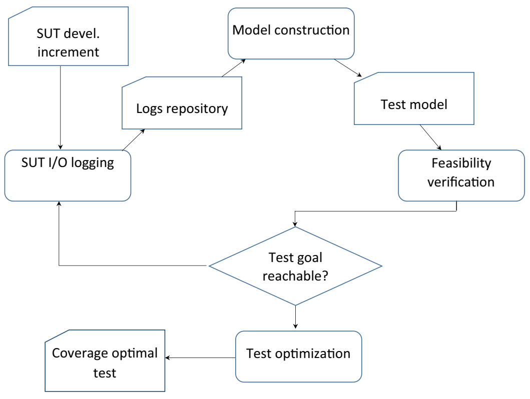


Fig. 1 A sample of test generation workflow in TestIt

the appropriate model-based testing tools that support various testing workflows. As a concrete example of tool integration, we consider Uppaal tool family being used in TestIt toolkit.

Typically, the model-based conformance testing process starts from SUT model construction. This is based on formalization of the requirements the SUT is implemented from. The test model construction from requirements is common in most of the test-driven development methodologies [9]. Alternatively, the model can be learned from system monitoring logs by applying various machine learning approaches. In this paper, we demonstrate the combined approach where the model for test purpose feasibility verification is constructed from the SUT requirements description and for test optimization the model is augmented with additional coverage information extracted from the SUT interface logs. The latter is to demonstrate the TestIt usefulness in the development processes where the test model evolves in lock step with the SUT development increments.

A sample of TestIt incremental test development workflow is depicted in Fig. 1. The input to the workflow is an executable SUT increment. Its execution in interaction with the simulated environment provides the SUT input–output logs. Based on the monitoring data, the logs are extended with context information, time stamps, semantic tags, etc. necessary for test coverage optimization (for more detailed description of logging ,we refer to Sect. 4.2).

The model updates needs to be verified for feasibility of test generation. The model is feasible if the test goal specified in terms of the SUT model elements or other coverage items is reachable. The test goal can be, e.g. the coverage of a specific code segment in SUT or reaching a specific state in the SUT model. The test goal reachability is verified using model checking [10]. In case the verification provides negative result, it means either the logs repository does not include necessary traces to provide an executable model and further monitoring experiments are needed, or alternatively, the design itself is incorrect and excludes the behaviours that implement the test goal. The decision on how to resolve the issue is up to the SUT developer and test engineer involved in the process. Provided the verification proves the test model feasibility, i.e. the tests can be generated using the test model, further steps (described in Sect. 6.1) proceed with the test optimization.

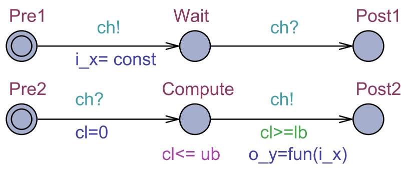


Fig. 2 Simple example of Uppaal TA model

3.3 Uppaal timed automata

TestIt uses Uppaal timed automata (TA) as core formalism. Uppaal TA are defined as a closed network of extended timed automata. These automata are combined into a single system by synchronous parallel composition. An automaton consists of locations (vertices in graphical notation) and edges (directed arcs in graphical notation) between the locations.

The set of variables associated with an automaton have valuations that are called states and the configuration of a model consists of its current control location and assignments to all model variables and clocks. The automata can be synchronized using synchronization links named channels between edges. The channels that by modelling naming convention have prefixes in and out in their names are used for sending commands to the SUT and receiving feedback from the SUT.

An example of Uppaal TA with two simple processes composed in parallel is depicted in Fig. 2. Both processes start from their initial location $Pre1$ and $Pre2$, respectively. At its first move from location $Pre1$ to $Wait$, the $Process1$ synchronizes with $Process2$ via channel ch that labels edge $(Pre2, Compute)$. At the same time, $Process1$ updates variable i_x with the value of constant $const$ and $Process2$ resets the clock cl. After reaching location $Compute$ $Process2$ waits maximum ub time units, i.e. till the location invariant $cl <= ub$ holds. Next, edge $(Compute, Post2)$ can be fired earliest after lb time units (by clock cl). This is specified in guard condition $cl >= lb$. Firing edge $(Compute, Post2)$ is synchronized again with edge $(Wait, Post1)$ in $Process1$. When executing transition $(Compute, Post2)$, the variable o_y is updated with the value of function fun where variable i_x is an argument. Both processes terminate at the same time in locations $Post1$ and $Post2$, respectively.

3.4 Model checking with Uppaal TA

The correctness verification of Uppaal TA models is performed using model checking technique [10]. Uppaal TA model checking problem can be stated as follows: given a correctness property φ stated in temporal logic TCTL and a model formalized in Uppaal TA, the validity of φ in model M can be verified by solving the satisfiability problem $M \models \varphi$. Model checker explores the model M state space trying to find an interpretation of formula φ. If it succeeds, the formula is declared to be valid and witness trace is issued. Otherwise, the counterexample is generated that provides diagnostic information why the formula is not valid. This allows to discover unintended behaviours in the model, such as states and transitions that the test case never reaches. The liveness properties are expressed as reachability constraints of legal model states and safety properties as non-reachability of illegal or unintended states. Typically, deadlocks and livelocks indicate violation of liveness properties. Such a model-based analysis can reveal the design errors of tests before their further development and execution.

In addition to standard safety and liveness properties verifiable in the test models, Uppaal model checker supports verifying timing properties such as time-bounded $leads\_to$, timed race conditions and other. Another practical outcome of model checking for testing is that verification witness traces can be used as symbolic test sequences applicable to verify that the correctness property is implemented properly also in SUT.

3.5 Symbolic test execution

Uppaal TA models are executed using DTRON [11] tool which extends Uppaal TRON [12], a testing tool based on Uppaal engine. TRON is suited for black-box conformance testing of timed systems. This tool enables simulation of the model in real time and allows interfacing with the SUT. For the Uppaal TA to send executable test inputs to the SUT, it needs an adapter to handle the synchronization signals from the model. These adapters have to be created depending on the use case. For the use case presented in this paper, we created an adapter for sending abstracted navigation goals to the SUT and converting the SUT outputs back to the symbolic form interpretable in the model.

3.6 Test coverage measurement

In model-based testing, the test coverage can be measured either by using model coverage or coverage of the code units abstracted in the model or both combined. The most common code coverage measurement is achieved by counting all the lines that have been executed. Traditionally, line counters are never reset during the measurement process. This is not an issue in case of measuring the lines that are executed during a full test execution. This approach can be improved with model-based testing by measuring the lines of code that get executed when performing some action represented as a transition in the model. This information can later be used to maximize the code coverage of the test.

For the code coverage to be measurable, possibly in combination with model structural coverage, both the test model execution and the SUT software stack must support it. The main programming languages used in ROS software development are C++ and Python. For code coverage, the C++ stack must be compiled with code coverage support options (profile-arcs and test-coverage). Python programs must be run via a wrapper (e.g. Coverage.py library) which collects the code coverage information.

The software stack has to handle the SIGUSR1 signal to support the state transition code coverage measurement. The C++code needs to call *__gcov_flush()* function to flush the coverage data. For Python, the coverage wrapper must call a save function. If *Coverage.py* is used, *coverage.save()* must be called and internal class variables *lines* and *arcs* must be reset. If these variables are not reset, the subsequent calls to the save function will return the full list of lines executed since the start of the program.

4 TestIt toolkit

4.1 General design considerations

TestIt is designed to be used with existing and new testing tools and supporting software. For example, it is possible to run linters (e.g. roslint[4] and static code analysis tools (e.g. HAROS [13])) as part of the testing process. The main use case for TestIt is however model-based testing.

One of the most important benefits of using MBT is the complex emergent scenarios that can be discovered by simulating both the SUT and the environment (i.e. the static world and dynamic actors in the world) together. It is very difficult to design test scenarios for autonomous systems which test the full software stack thoroughly. This is due to the fact that usually software is developed by different teams and the knowledge of the full software stack is very rare. Accounting for all permutations of conditions that can arise in complex dynamic environments is exceedingly difficult even with full knowledge of the software stack. Using MBT and tools (e.g. Uppaal TA, NModel) helps in this regard by allowing the environment and SUT (i.e. the autonomous system being developed) to be modelled separately. The possibility of modelling different actors separately reduces the modelling complexity which in turn reduces the cost of test-

[4] http://wiki.ros.org/roslint.

ing. After creating the single actor models, it is possible to instantiate these models in parallel and in multiple instances.

As discussed in Sect. 2, integrating testing into robotics software development CI processes is highly coveted. TestIt is well suited for integrating into CI services as the testing pipelines are designed to work with Docker containers. Using Docker containers makes it easier to integrate into CI processes, because of the ephemeral on-demand nature of the container technology. The containers are always started from the same state and the state is not stored after finishing, which is the desired behaviour in testing context. This feature ensures that testing is stable and there is no risk of influencing the initial state on subsequent test execution. Executing tests without sandboxing the software can run into mutability issues.

Long-term reliability is another key concern for autonomous robots which operate in dynamic environments. Finding software bugs that appear immediately or within a short time window is significantly easier than detecting errors that emerge after a long time has elapsed (e.g. memory leaks and difficult corner cases). Using simulation for long-term autonomy testing with compressed timescale improves efficiency. Still, long-term testing with real robots is challenging and limited by real-time factor. In many cases, it is possible to perform simulation faster than real time to further increase the time efficiency of testing. TestIt supports running tests over long time periods to find interesting scenarios that are exceedingly difficult to discover without model-based generated tests.

4.2 Architecture

TestIt[5] comprises an open-source ROS package containing the daemon and the Command-Line Interface (CLI) to send commands to the daemon and a Docker container[6] with bundled testing tools.

The daemon can control multiple TestIt pipelines as can be seen in Fig. 3. Due to the parallelization of test execution, using multiple pipelines in simulation based testing improves test scalability. Each pipeline can run on a separate server, for example in the cloud (AWS, Google Cloud or Azure), ensuring scalability.

TestIt supports starting and stopping pipeline servers as part of the testing workflow depicted in Fig. 4 which works well in case of using cloud servers for testing as cloud services are billed based on the time used. Therefore, only bringing them online when needed is cost efficient.

The configuration for the SUT and TestIt Docker container is defined in the YAML[7] format configuration file which is

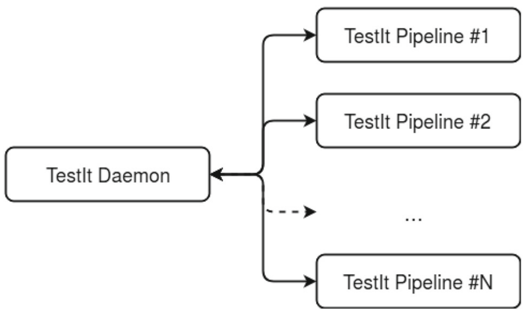


Fig. 3 TestIt architecture

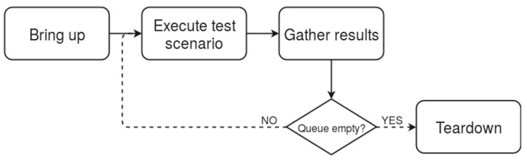


Fig. 4 TestIt pipeline workflow

passed to the daemon upon start up. The configuration consists of infrastructure configuration and test scenarios.

The overview of a single pipeline is shown in Fig. 5. The infrastructure configuration defines the pipelines that are used for running the test scenarios. A pipeline comprises the SUT with the software that is tested as well as the TestIt docker container configuration. The pipelines can be configured depending on the available hardware or budget constraints for the cloud testing. TestIt can even be used in a single pipeline testing configuration with a single computer.

The packaging requirements for the software running in the SUT are not strictly constrained. TestIt toolkit ROS integration relies on the SUT running the ROS master service (i.e. roscore) to which the tools in TestIt container can connect. Other than that, the SUT can be considered as a black-box system and TestIt can be used as a black-box testing toolkit which requires no modification by the tester. If test code coverage measurement is required, the SUT must be configured in a way that supports it. More information about this feature is provided in Sect. 3.6.

As can be seen in Fig. 5, the SUT can either run in simulation or in a hardware-in-the-loop configuration. TestIt is simulator agnostic. The concrete simulator that is used is not fixed thanks to ROS design principles. The simulation communicates directly with ROS along with the SUT software stack and TestIt interacts directly with ROS and not the simulator itself. This allows TestIt to be used in a broader field of robot software development.

The test scenarios specify the concrete cases that are executed to test the software. The scenarios can be executed in

[5] https://github.com/GertKanter/testit.

[6] https://www.docker.com.

[7] https://yaml.org/.

different ways. The simplest is to execute a program or script that gives inputs to the software stack (e.g. goals for navigation), but the scenarios can be defined also using complex models. Uppaal timed automata [14] models are currently supported for model-based testing using DTRON. TestIt can be extended to accommodate other model checkers (e.g. DIVINE [15], NModel [16]) in a way that does not inconvenience the toolkit user. This is one of the design goals of TestIt as it is packaged into a Docker container. It is possible to pre-install and configure everything inside the container so that the toolkit user or a CI service does not need to install and configure all of the tools separately.

As depicted in Fig. 5, TestIt Docker container can be bundled with multiple supporting tools. One of the criticisms of model-based testing is the difficulty and labour-intensive procedure of creating models. TestIt toolkit addresses this issue by supporting generation of models from other specification formats as can also be seen in Fig. 5. The generation of a model from topological map format developed for use in STRANDS project [17] has already been implemented in TestIt [18]. The support for generating models from SMACH state machines, ROS BehaviorTrees or other formats can be achieved in a similar way. Creating models from other specification formats saves time and gives a good starting point to expand the models for automatic model-based testing.

An important component for TestIt is the logger. The logs are used to optimize the test scenarios. The log entries are stored as JSON notation strings with each string denoting one event. The entry is a dictionary with the test run identifier, timestamp, coverage information related to the transition, data transmitted to the SUT, transmission channel information and information whether the entry corresponds to before or after transmitting the information to the SUT. This discrimination allows the information to be further analysed based on the result of executing a test model transition. The information about the Uppaal TA channel that models interaction between the SUT and the environment contains the name, type and proxy name if required (for services and action library). The proxy is used to allow logging to occur without requiring modification of the software that is tested. This is caused by ROS design, namely, services can only be handled by a single server. To allow services to be monitored as state transitions, the logger needs to be able to provide a proxy service that forwards the actual service request to the SUT and gives the result to the requester. The proxies can be set up with ROS remapping without modification of the SUT.

The final component of TestIt is the online test runner. This component uses the optimization algorithm to dynamically guide the system into maximum gain states (e.g. gain function maximizes code coverage). As the online tester is executed at the same time as the SUT, it is possible to take actual gain information from the SUT into account while planning the next SUT input signal.

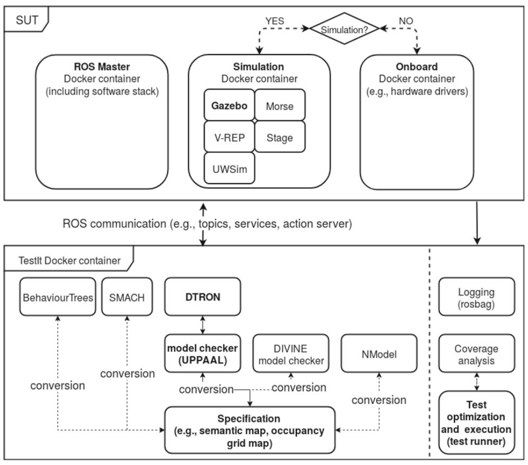


Fig. 5 TestIt pipeline

5 Configuring TestIt for test execution

To start testing with TestIt, the following steps have to be taken. First, the SUT must be defined. For ROS-based robots, a convenient way to test the robot software is by packaging the SUT in a Docker container. This allows TestIt to easily start and stop the SUT without mutability problems. By using Docker containers, we can be sure that the initial state of the system is always the same which ensures repeatability.

TestIt can be configured to test systems that are not packaged into containers as well. The ROS-based use case requires that the TestIt container has access to the ROS core node. This means that one can also use TestIt to test hardware-in-the-loop systems if the ROS core node actually runs not in simulation but on real hardware.

In general, TestIt can be used to test non-ROS systems as well, as there is no strict requirement on software configuration. It is possible to configure TestIt to launch tests that use non-ROS connection to the SUT and it is still possible to assess whether tests pass or fail.

In the rest of the paper, we focus on ROS-based system as it is the main use case of TestIt. Once the SUT interaction (i.e. starting and stopping the SUT) is specified in the configuration file, the TestIt test runner docker container must be configured. The base TestIt configuration includes an ROS installation and TestIt ROS-logger which logs the interaction between the tester and the SUT.

Next, the test scenarios have to be configured. A test scenario can include both a test and an oracle to monitor the outcome of the test. In case a scenario is configured without an oracle, the outcome of the test is based on the return code of the test itself. This means that the test script should terminate with a return code zero upon success and non-zero upon failure. If the test is executed using an MBT tool (e.g. Uppaal and DTRON), the test does not terminate itself (i.e. the model is executed without time limit) and must be terminated after an oracle determines the test as a success or failure. An alternative termination method is via timeout that can be specified for the scenario. The outcome of a timeout occurrence (i.e. pass or fail) can also be defined in the test scenario specification.

It is also possible to run an individual test scenario in parallel to take advantage of concurrency if there is enough computing power available for multi-threaded testing. To support this, TestIt features a credit-based system. The credits can be added or removed via the TestIt command-line interface. Each test execution decrements the credit value by one and testing will continue until the credit value reaches zero.

Finally, the test pipelines must be configured. Each pipeline consists of the SUT and the TestIt Docker container. In the main use case, both are Docker containers which simplifies pre-test setup and post-test teardown. The simplest configuration uses the same host machine for testing as the TestIt daemon host (i.e. running the tests on local host). But, as discussed, TestIt can be scaled as resources allow by creating several pipelines that are run on different servers. This allows for more time-efficient testing by running the scenarios in parallel.

TestIt is highly flexible, as it can provide value to the testing process even when the SUT is considered a black box and only its environment behaviour is modelled. In that case, the SUT model just needs to be responsive to the environment inputs. That is necessary to avoid blocking of model-simulated interactions. TestIt can still be used to log the interaction and results of changing the environment. In case the SUT is deterministic, it is possible to create a model of an environment process that is executed in the simulation (e.g. opening or closing of doors in a mobile robot use case).

6 Test optimization

To increase testing efficiency, it is possible to optimize the tests. TestIt is designed to be run in two modes: exploratory testing (learning) mode and focused testing mode (aiming to maximize gain via either code coverage, data coverage or model coverage criteria).

In the exploratory testing mode, we send stimuli to the SUT (both the software under test and the environment) according to some specification. In the main use case, this specification consists of models for all the aspects that are necessary to model the system at the required level of detail. For example, this can be a navigation graph model for an autonomous vehicle augmented with a behaviour model of a pedestrian and traffic light state machine in an autonomous driving scenario. In the exploratory testing mode, TestIt will explore the SUT according to the model and will log all the inputs sent to the SUT. In addition to the details of the input, the result of the sent input as well as code coverage information is logged. It is also possible to log additional variable values that might be useful for improving testing quality and efficiency.

After the exploratory phase when enough log data has been gathered, it is possible to perform optimized testing based on the data. Optimized testing is performed using the online test runner. This approach ensures time-efficient testing. Combining TestIt and MBT will help the software team to automatically discover edge and corner cases in the complex interaction of multiple components of an autonomous system. To achieve test optimization, we propose an optimization algorithm based on weighted gain function. To compute the gain function, the optimization criteria need to be specified. For example, if the SUT consists of two software packages it is possible to define different weights for each package and each source code file which optimizes the scenario based on these weights. This allows teams that work on different components to create test scenarios that are the most efficient for their developed component by maximizing the code coverage and other criteria that are most interesting for them (Fig. 6).

The optimization algorithm has three modes. Each mode has a different way of creating the optimization graph, but is algorithmically identical. The optimization modes are probabilistic, best trace and combined. The different optimization graphs that are generated are depicted in Figs. 7, 8, 9 respectively. These figures use different notation semantics compared to other figures presented. The states are marked in parentheses (e.g. $(WP1)$). The *cov* and Q signify the coverage set and the edge quantity, respectively. The coverage sets can be merged with a plus operator (denoted as $+$ in the figures). The merge operation is demonstrated in Eq (1). For example, the notation $\mathrm{cov} = 1 + 7$ in the Fig. 7 at the edge from $(WP2)$ to $(WP1)$ denotes the merge operation of coverage sets cov_1 and cov_7. It is important to note that one state is considered an initial state (marked with a double circle). The initial state is undefined due to the fact that the optimization graphs are constructed from logs and it is not possible to determine the initial state from the log because only the monitored events are logged (i.e. the source of the first transition is not explicit). The transitions in the figures are encoded with lowercase characters.

6.1 Optimization algorithm

The algorithm for optimization algorithm is shown in Algorithm 1.

The optimization algorithm optimizes the input sequence to the SUT to provide the maximum gain in the least amount of steps. At its core, the algorithm recursively deepens the gain tree to the specified maximum depth level. The algorithm is initialized and started in $compute\_sequence()$ function. The initial state is the ($None$) node of the optimization graph. The supporting functions $get\_state\_hash()$ and $get\_chan\_hash()$ provide a way to encode and decode the states and data channels so that they are unique and can be used as dictionary key values for the other data structures. The recursive computation happens in the $expand()$ function which expands each tree node (calculates the gain values of the children). Each gain tree level depends on the values of the previous level which means it has to be created level by level. After reaching the maximum depth level, the algorithm propagates the gain values (i.e. gained code coverage or other desired parameter gain) from the terminal nodes (the leaves at the maximum depth) to their ancestors up the tree to the root via the $update\_path\_gain()$ function. Once all the gain values have been propagated to the root node (creating the path of maximum gain), the algorithm can just return the best gain path first element of this maximum gain path as the next input to the SUT. The function $compute\_edge\_gains()$ computes the edge gains (i.e. possible gain values for the next step) for a state based on the current parameter state (i.e. it takes into account the code lines which have already been covered).

The novelty of the optimization algorithm is that it takes into account the actual values from the online test runner (i.e. what actually happened) and uses these values on the probabilistic values based on the constructed probabilistic model from all parallelized pipelines to find the best sequence of steps to maximize the gain of code coverage or other criteria.

Algorithm 1 Test scenario optimization algorithm

```
tree_id ← 0
function COMPUTE_SEQUENCE(state,step_lim,max_d)
    seq ← []
    next ← [state, {}, 0]
    for _ in range(step_lim) do
        next ← COMPUTE_STEP(max_d,next[0],next[1])
        data ← GET_STATE_HASH(next[0])
        chan ← GET_CHAN_HASH(next[0])
        seq ← seq + (chan, data)
    end for
    return seq
end function
function COMPUTE_STEP(max_d,state,param)
    tree_id ← 0
    gain_tree ← EXPAND({}, [0, state, {}, param], max_d)
    UPDATE_PATH_GAIN(gain_tree)
    best_g ← 0
    for key in gain_tree do
        for child in gain_tree[key] do
            node ← gain_tree[child[0]]
            if node = None then                        ▷ Terminal node
                if child[4] >= best_g or best_g == 0 then
                    best_g ← child[4]
                    best_step ← child[5][1]
                end if
            end if
        end for
    end for
    sel_step ← None
    for edge in gain_tree[0] do
        if edge[0] == best_step then
            sel_step ← edge[1]
            best_param ← edge[3]
        end if
    end for
    return (sel_step, best_param)
end function
function EXPAND(tree,elem,max_d)
    el_id ← elem[0]
    state ← elem[1]
    param ← elem[3]
    depth+ = 1
    gains ← COMPUTE_EDGE_GAINS(state, param)
    tree[el_id] ← []
    for g in gains do
        tree_id ← tree_id + 1
        new_el ← [tree_id, g, gains[g][0], gains[g][1]]
        tree[el_id] ← tree[el_id] + new_el
        if depth <= max_d then
            tree ← EXPAND(tree, new_el, max_d, depth)
        end if
    end for
    return tree
end function
```

6.2 Example

To illustrate how the optimization algorithm works, we present an untimed model for simplicity. The example model which generates the inputs to the SUT is shown in Fig. 6. The model consists of three states, $WP1$, $WP2$ and $WP3$, and transitions, a, b, c, d, e, f, with the arrows showing the direction of the transition.

The optimization algorithm starts by constructing a graph from the logs. Only the relevant parts of the log entries are presented. The following example has three separate test executions denoted as L_1, L_2 and L_3.

$$L_1 : (a, \text{cov}_1 = \{...\}, t = 0) \rightarrow (b, \text{cov}_2 = \{...\}, t = 2) \rightarrow (f, \text{cov}_3 = \{...\}, t = 3),$$

$$L_2 : (f, \text{cov}_4 = \{...\}, t = 0) \rightarrow (c, \text{cov}_5 = \{...\}, t = 1) \rightarrow (b, \text{cov}_6 = \{...\}, t = 4),$$

$$L_3 : (a, \text{cov}_7 = \{...\}, t = 0) \rightarrow (d, \text{cov}_8 = \{...\}, t = 4) \rightarrow$$

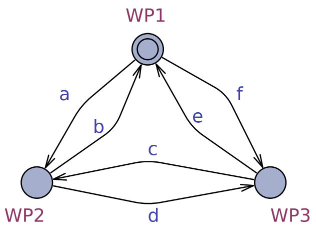


Fig. 6 An example model

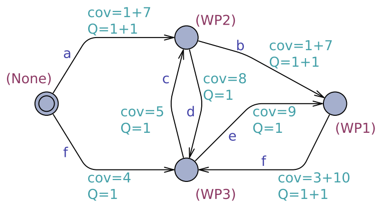


Fig. 7 Graph for probabilistic optimization

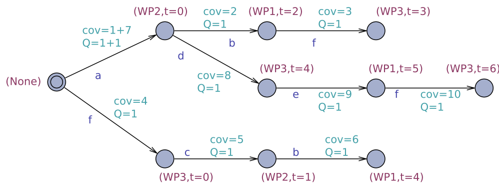


Fig. 8 Graph for best trace optimization

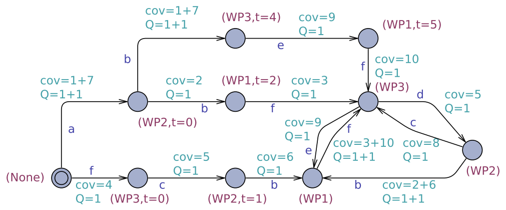


Fig. 9 Graph for combined (probability and trace) optimization

$(e, \text{cov}_9 = \{...\}, t = 5) \rightarrow (f, \text{cov}_{10} = \{...\}, t = 6).$

Each execution comprises several entries (state transitions) and are denoted as a tuple $(T, \text{cov}, \text{ts})$, where T is the transition (the source state is inferred from previous state), cov is the the code coverage recorded at the moment of logging and ts is the timestamp of the entry. The coverage entries are dictionaries where the keys are tuples (file, l), where file is the file name and l is the line (or non-overlapping interval if pre-processed in this way) that was executed since the last log event. The values of the dictionary are the probabilities of the specific line (or interval) being executed. This probability is initially always 1.0, but is modified as the graph is simplified and the similar edges are merged. An example of such merge is shown in Eq (1). As can be seen, if the line is executed in both coverage sets, the probability remains 1.0, but if some lines of code are not present in both coverage sets, after normalization the combined probability is reduced.

$$\begin{aligned}
\text{cov}_1 &= \{("a", 1) : 1.0\}, \\
\text{cov}_2 &= \{("a", 1) : 1.0, ("a", 2) : 1.0\}, \\
n(\text{cov}_1 + \text{cov}_2) &= \{("a", 1) : 1.0, ("a", 2) : 0.5\}.
\end{aligned} \tag{1}$$

As discussed in Sect. 6.1, it is possible to create the optimization graph in three ways with each method providing different benefits.

The first method and the one we use in the case study is the probabilistic graph optimization shown in Sect. 7. The probabilistic graph is cyclic by construction as long as the logs it is constructed from have sequences in which the system has visited the same states in a loop. For example, if the logs show a transition from A to B in one log and from B to A in another, the resulting graph will have a loop between A and B states. Because of this the probabilistic graph is compact and allows long SUT input sequences to be generated due to normally having cyclic structure. Considering the previous example, we could generate an infinite input sequence $A \rightarrow B \rightarrow A \rightarrow B \rightarrow \ldots$ with each transition potentially providing some gain.

The second option is the best trace optimization for which the graph is constructed as chains of states based on the logs. As can be seen in Fig. 8, the states are linked together as chains but there are no loops. This means that we can only generate a finite number of inputs to the SUT starting from the initial state since there are no loops. This method is the least computationally complex, but the resulting graph cannot be used to generate long sequences. This means that once the best trace has been traversed, the test stops and no effort is made to attain more coverage based on recorded probabilities. This option can be used to simply pick and execute the best trace from all known traces. There are scenarios in which this can be useful, for example, if the logs are generated from real-world data and tests are executed on real-world systems with the requirement that the generated SUT input sequence has been seen before and is guaranteed to have worked previously.

The final option is the combination of the two aforementioned methods shown in Fig. 9 and creates graphs that have both the logs with timestamps but also generalizes to probabilistic optimization after traversing through the logs.To benefit from the best trace optimization (i.e. initially follow a logged input sequence) but still be able to continue the test after traversing a known trace, the final state of an actual log entry needs to be connected to a probabilistic state.

Following the optimization graph construction, the gain tree is created which is what the optimization algorithm uses to determine the best input to send the SUT to gain the most of code coverage (in the exemplified case). This can be combined with other optimization criteria such as localization covariance (localization uncertainty) based on the simulation. It is possible to find sequences of inputs that maximize such uncertainty which in turn creates efficient test scenarios by forcing the SUT into a difficult situation (possibly to an edge or corner case). The gain tree based on probabilistic graph in Fig. 9 is shown in Fig. 10. The optimization algorithm generates the gain tree from top down with each level denoting its depth level. Each successive level contains more nodes, the number of which is directly dependent on the branching factor of the optimization graph. Large branching factor graphs will reduce the effective maximum depth of the tree, since computation requirements will increase as more nodes are added to the tree. The optimization algorithm will choose the highest gain increase chain (chains are denoted as C1 through C6 in Fig. 10) at each step.

7 Case study: model-based testing of robotic intruder detection system

7.1 System under test

Robotic systems comprise numerous subsystems and rely on both software and hardware which means that there are enormous challenges to overcome before achieving a reliable autonomous system. For this case study, we only focus on the software component as TestIt toolkit is designed to improve testing predominantly software. With this is mind and to emulate a more realistic use case of autonomous system software development scenario, we limit the testing scope to one critical component for the smart house system—intruder detection algorithm. Therefore, the SUT in this case study is the robot software responsible for detecting the intruder. This means that all other components are not under test, but are still used as prerequisites for testing the SUT. As TestIt is used for testing complex integrations of the full software stack, it is usually advantageous to create placeholder components for subsystems that have not yet been developed. This allows the SUT component to be tested before all the components that the SUT component depends on are finished. This allows for faster time to market for the developed product.

The robot model used for this use case is based on TurtleBot.[8] This robot features a Kinect sensor which is converted to lidar data, as this data source is easier to integrate with available software components. For navigation, we utilize the standard ROS navigation stack. Relying on standard components is preferred because it allows easier integration if the need for replacing some component arises in the later stage of product development. The robot model is also irrelevant from the point of software development, as the software team only relies on the input data and not how this data is generated. If a need for a different robot arises at a later stage, the model can easily be replaced and the tests can be executed again.

The SUT component for this use case is available at a source code repository.[9] At this repository, there are two ROS components: $patrol\_detector$ and $patrol\_planner$. The $patrol\_detector$ is responsible for analysing the lidar data to detect a target. The $patrol\_planner$ plans the navigation goals to cover the whole area of the floor as specified by the navigation graph. It is important to note that we intentionally designed the planner to be deterministic to make the demonstration use case more easy to follow.

7.2 Test scenario description

To demonstrate the feasibility of TestIt toolkit, we present the following scenario. To improve the security of smart buildings, autonomous patrol robots are deployed to investigate unidentified visitors. For the use case, we have created the floor plan based on the Information and Communication Technology (ICT) building 4th floor in the TalTech campus. The plan depicted in Fig. 11 is annotated with the navigation graph for the robots with the nodes denoted as blue dots and the edges are indicated as lines between them. For this case study, we use two autonomous patrol robots working as a team to discover unidentified moving targets. The patrol robots start positions are marked with red and green dots. There is also one actor that is designated as the intruder whose starting location is denoted by the yellow dot. The intruder uses the same simulation model as the patrol robots, but it does not have an active autonomous navigation planner.

7.3 SUT model construction

For verifying the feasibility of the test with respect to the test purpose specification, the reachability of the targeted test coverage needs to be proved, at first. This helps avoiding generting inconclusive test cases and waste of time when designing further testing steps. In this paper, the test model construction and test feasibility verification are demonstrated based on Uppaal TA modelling formalism and Uppaal model checker. As discussed in Sect. 3.2, the model for test feasibility analysis can be constructed even before the real development of SUT is started, since the model and verification goals can be extracted directly from system requirement specification. Further coverage-based test optimization steps

[8] https://www.turtlebot.com/.

[9] https://gitlab.com/GertKanter/testit-patrol-sut.

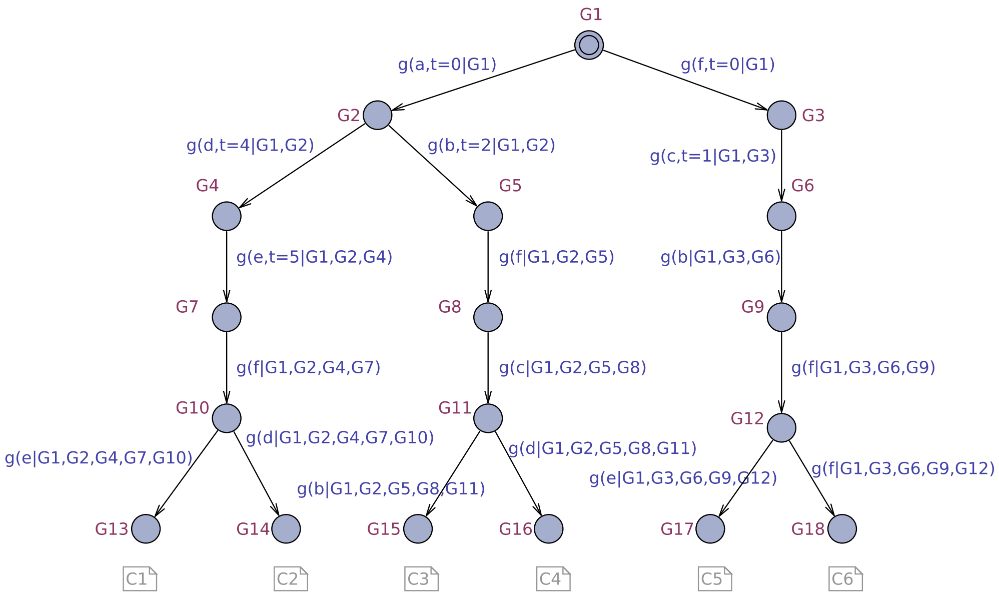


Fig. 10 Gain tree

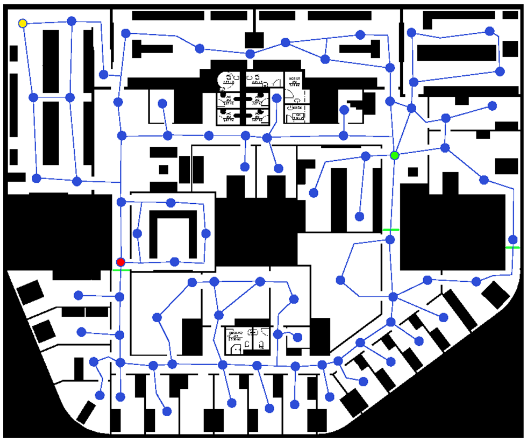


Fig. 11 ICT building 4th floor navigation graph

can be made later incrementally in the course of SUT implementation, as demonstrated in the rest of this section.

For model construction, the state space of the model is constructed at first, to specify the conditions and effects of model actors. The actors are intruder, and two robots patrolling on the office floor. The formal description of actors environment is based on a real office building floor topology that is asbtracted in the form of a navigation graph (Fig. 11). The data structure representing the navigation graph is a vector that consists of graph nodes names. Each node in the graph denotes a waypoint to be covered when the robots are patrolling. At the same time, the waypoints also denote the potential locations of the intruder to be detected. Possible moves of actors in the navigation graph are modelled using a two-dimensional array EDGEs with first dimension of length N (N is the number of waypoints) and second dimension of length Br (Br is maximum branching factor of the graph). Each row in the array EDGEs corresponds to a waypoint and the elements of a row correspond to the neighbour waypoints of that node. The models of the intruder and the guard robot behaviour are depicted in Figs. 12, 13, respectively.

The data structures that describe the actors' state are vector Pos, the elements of which encode the current positions of actors in the navigation graph. The vector Occ of size N encodes the occupancy of waypoints by robots and vector Vis the number of visits to each waypoint. Boolean variable $Detected$ is assigned the value $true$ when the intruder is located. Each actor's behaviour is modelled using a pair of automata, one emulating the agent's decision making (named $Intruder\_decide$ or $Robot\_i\_decide$, respectively, where i in the name denotes the number of the patrolling robot), and the other automaton (named $Intruder\_act$ and $Robot\_i\_act$, respectively) emulating the action performed to accomplish the decision. The automata decide the order of exploration of waypoints. Intruder picks the next waypoint randomly from the list of adjacent nodes of its current location. The patrolling robots pick the next node by preferring the least visited ones. The

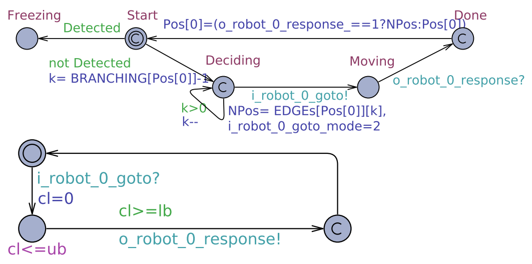


Fig. 12 Intruder model and responder

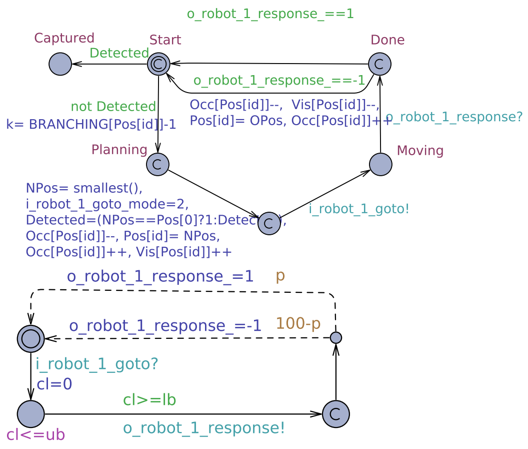


Fig. 13 Robot 1 model and responder

automata $Intruder\_act$ and $Robot\_i\_act$ emulate moving from one waypoint to another. Moving takes time specified by an interval $[lb, ub]$, lb and ub, denoting lower and upper timebounds, respectively. The navigation graph is designed so that the distance between waypoints is approximately the same. This keeps the duration interval $[lb, ub]$ and the move duration approximation close to real moving duration. Since the robots' navigation is not perfect, it can fail to reach the waypoint in some cases. This is represented in the model as probabilistic transition. The probabilistic transitions of $Intruder\_act$ and $Robot\_i\_act$ automata are labelled with probability estimates. The probability of reaching the targeted waypoint is denoted by p. The probability estimates are normalized with value range $0 - 100$, so the failure of reaching target waypoint has estimate $100 - p$. If the target waypoint is not reached, the robot returns to its previous location and tries the same target again by changing its route. Intruder detection in the model is close to real implemented detection mechanism. The intruder is visible to the patrol robot when the robot is moving towards the next waypoint where the intruder is located, i.e. the distance between robots is not more than the distance between neighbour waypoints, and the intruder is visible in robot's front view sector. This is modelled with conditional assignment $Detected = (NPos == Pos[0]?true : Detected)$, where $NPos$ is the variable modelling the next waypoint for a patrol to go and $Pos[0]$ models the current position of the intruder.

7.4 Test feasibility verification

Supposing that for maximum code coverage, both the robot navigation and intruder detection scenarios have to be represented in the model the verification property should express the reachability condition that it is always the case that at least one of the robots eventually detects the intruder. By referring to the model global variable $Detected$, this can be expressed by the TCTL formula $A <> Detected$. The variable $Detected$ is updated to $true$ in the model whenever any of the patrolling robots detects the intruder. The verification experiments show that this test case is feasible for the navigation graph that includes 86 nodes and patrols implement described search strategy when there are at least three patrol robots, and if the robots are at least three times faster than the intruder. The robots' navigation failure probability should be lower than 5%. Weaker verification condition $E <> Detected$ that is valid if there exists at least one of such behaviour where the intruder is located has been proved under more relaxed conditions where only two robots with moving speeds find the intruder in the building. If further action is not needed to be covered in the test case, this proof is sufficient for continuing with the test case optimization. Keeping in mind the test is feasible under given constraints, further test optimization steps are targeted to reduce the test length by allocating probabilistic gain functions to the test model that guide the test run towards the goal along the optimal test path.

7.5 TestIt configuration

TestIt configuration for testing the SUT is specified in a separate source code repository[10], as encapsulating the testing configuration is preferable to keep both repositories (i.e. the SUT repository and TestIt configuration repository) clean and concise. The TestIt configuration itself essentially consists of the testing infrastructure control configuration, SUT launch and test launch parameter specification. TestIt SUT launch is configured to start the full software stack and bring all robots (i.e. patrol robots and intruder robot) online and ready to receive navigation goals. After a short duration, the patrol planning as well as detection algorithm (the SUT component in this case study) is also started. To demonstrate our

[10] https://gitlab.com/GertKanter/testit-patrol-aws.

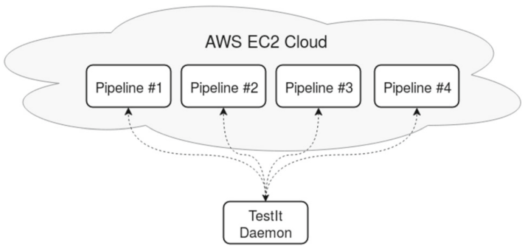


Fig. 14 TestIt AWS configuration

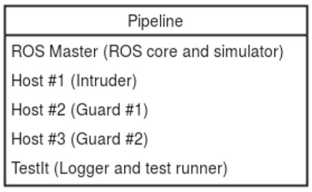

Pipeline
ROS Master (ROS core and simulator)
Host #1 (Intruder)
Host #2 (Guard #1)
Host #3 (Guard #2)
TestIt (Logger and test runner)

Fig. 15 Pipeline configuration for multi-robot system

approaches scalability we have used the AWS cloud to create four pipelines which can run the tests and simulations in parallel. The configuration is shown in Fig. 14. Each pipeline consists of a SUT ROS master server (including the simulator), an intruder navigation stack server, two guard navigation stack and patrol algorithm servers and a TestIt logger and test runner server as seen in Fig. 15.

The scalability of TestIt was measured and is shown in Fig. 16 to demonstrate that it scales roughly linearly with the number of pipelines configured in the system. As the plot was based on measuring real simulation data, there is a small margin of variance due to the fact that the robots generate state transition logs at different rates based on their navigation speed and waypoint reaching success rate. Linear scaling is expected as individual pipelines operate separately and the overhead from communication with the TestIt daemon is negligible. The logs are retrieved after tests are finished, which means there is no data transmission bottleneck even with very large pipeline configurations.

We have configured two separate tests for demonstration purposes. The first test is used to generate data for the optimization algorithm (log generation). The test launch parameter for this test specifies the process and how to generate inputs to the process. For this case study, we use Uppaal model and DTRON with a test adapter. The intruder goals are sent using this navigation model, but the patrol robot navigation goals are determined by the *patrol_planner*. The second test starts the online test runner which uses the optimization algorithm to efficiently navigate the intruder to achieve maximum coverage. The logs generated while running the optimization algorithm can also be used to modify

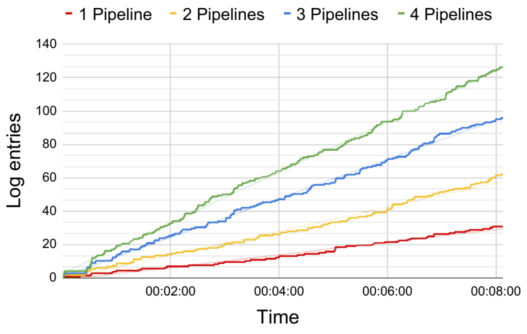


Fig. 16 Linear scalability of TestIt

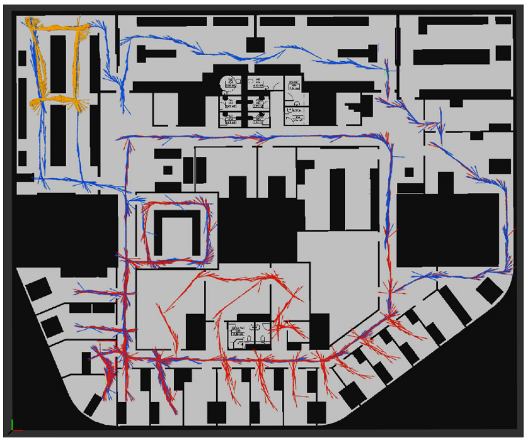


Fig. 17 Stage simulation visualization

the model probabilities which make the future tests more accurate with continuous feedback from the system.

7.6 Results

As discussed in Sect. 6.1, TestIt is designed to run in two modes. Exploratory mode is used to generate data for test scenario optimization to create optimal test scenarios. An example of exploratory phase is shown in Fig. 17 which is simulated with Stage simulator [19] and visualized using RViz tool. In the figure, the trajectories of the robots are visualized as a collection of odometry arrows. In this particular example, the intruder (denoted as yellow) was operating in a limited area in the top left. The trajectories of the patrol robots are denoted in red and blue.

After sufficient amount of data has been logged, it is possible to extract the optimized scenario from the log. Optimization requires that the logs contain an event of interest (i.e. a situation where increased code coverage is achieved) to be useful. In our case, the logger captured a high cover-

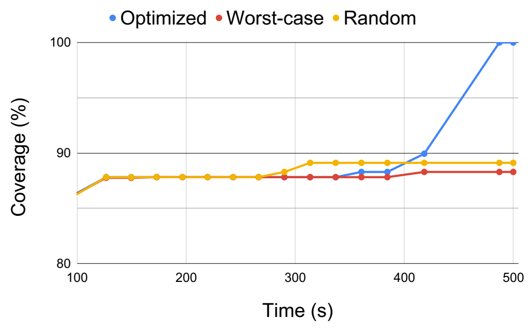


Fig. 18 Relative code coverage of the SUT

age event near the starting point of one patrol robot (denoted as a red circle in Fig. 11). The optimization algorithm was used on the test runner and the algorithm guided the robot to achieve full relative coverage of the *patrol_detector* node. The term relative coverage refers to the coverage based on what has been captured in the logs. This means that the sections of code that have not been executed during logging remain unknown to the optimization algorithm.

The coverage plot is presented in Fig. 18. The plot presents three cases: worst-case scenario, random scenario and optimized scenario coverage strategy. The test time was set to 500 s to allow the test runner sufficient time for guiding the intruder into different states.

In the worst case, the test runner always chooses the worst possible next input that gains the least coverage. This is achieved by continually traversing the farthest left top corner edges on the navigation map in a loop. Based on the logs, there is no detection event occurring in that region and therefore the relative code coverage remains constant.

The random strategy test runner picks the next input randomly from the known alternatives. The result shown is the average case in multiple runs. As can be seen from the figure, the random strategy achieves marginally improved code coverage compared to the worst-case strategy.

To attain the best coverage as rapidly as possible, we employ the optimization strategy. The beginning of the tests are similar to the main loop of the SUT software that is running and covered even without optimal control. The real benefit of the optimization algorithm becomes evident after some time has been given for the control algorithm to navigate the robot into position for the high code coverage event. The high code coverage event for this particular use case is when the intruder is detected by the guard robot and it takes time for both the guard robot and intruder to move to the same location. After sufficient time has been given for navigation, we can see the strategy succeeding in achieving full relative coverage within the time limit when the other strategies fail.

The test runner takes into account live code coverage when executing the test which makes it very responsive to actual results. This means that each test can be unique as the test runner tries to achieve full code coverage based on the actual results and adjusts the optimal input sequence to the SUT accordingly. This makes our approach robust and enables us to determine when to restart the simulation if the test optimization algorithm detects that the code coverage cannot be improved sufficiently and in reasonable time from the current state.

The online test runner is especially useful in high branching state space with low probability high code coverage events, as the test runner can identify the correct sequence in advance. This can be witnessed also in our use case with limited state branching. The random strategy failed to find the high code coverage events, whereas the optimization algorithm succeeded.

In the case study, we did not actively control the patrol robots to demonstrate that it is possible to test autonomous systems using the black-box approach without explicitly modelling their internal behaviour. If the patrol robots would have been controllable, the results would have been different and full coverage could have been achieved faster.

8 Conclusion

This paper addresses the quality assurance problem of smart building security systems where autonomous surveillance robots-based solution is studied. The quality of autonomous robotic systems integration needs significant testing effort to be supported by the test automation toolset. To address this need, the paper presents a testing toolkit named TestIt. Its primary focus is model-based testing of autonomous systems to improve long-term autonomy in the context of smart environment applications. The testing methodology, architecture and tools incorporated in TestIt are introduced. The main novelty of the presented solution is the scalable multi-pipeline testing architecture that enables incorporation of multi-purpose testing tools including those used in state-of-the-art model-based testing. As the second main contribution, test verification and optimization techniques have been presented. The usability of TestIt for test generation, its validation and optimization in autonomous navigation context is demonstrated using a realistic smart building intruder detection case study.

Acknowledgements This research was partially supported by the Estonian Ministry of Education and Research institutional research Grant no. IUT33-13.

References

1. Quigley M et al (2009) ROS: an open-source robot operating system. In: ICRA workshop on open source software, 2009(online). https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf
2. Kumar S, Vealey T, Srivastava H (2016) Security in internet of things: challenges, solutions and future directions. 49th Hawaii international conference on system sciences (HICSS). Koloa, HI, pp 5772–5781
3. Deshmukh J, Horvat M, Jin X, Majumdar R, Prabhu V (2017) Testing cyber-physical systems through Bayesian optimization. ACM Trans Embed Comput Syst 16(5s):1–18. https://doi.org/10.1145/3126521 (article 170, Sep 2017)
4. Bartocci E et al (2018) Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci E, Falcone Y (eds) Lectures on runtime verification. Lecture notes in computer science, vol 10457. Springer, Cham
5. Larsen KG, Pettersson P, Yi W (1997) Uppaal in a nutshell. Int J Softw Tools Technol Transf 1(1–2):134–152. https://doi.org/10.1007/s100090050010
6. Kapinski J, Deshmukh JV, Jin X, Ito H, Butts K (2016) Simulation-based approaches for verification of embedded control systems: an overview of traditional and advanced modeling, testing, and verification techniques. IEEE Control Syst Mag 36(6):45–64
7. Utting M, Pretschner A, Legeard B (2012) A taxonomy of model based testing approaches. Softw Test Verif Reliab 22(5):297–312. https://doi.org/10.1002/stvr.456. Accessed 18 May 2015
8. Tretmans J (1996) Test generation with inputs, outputs, and quiescence. In: TACAS, vol 1055 of LNCS, p 127–146. Springer, New York
9. Vain J, Kanter G, Srinivasan S (2017) Model based testing of distributed time critical systems. In: 2017 6th international conference on reliability, Infocom technologies and optimization (ICRITO) (trends and future directions), 20–22 Sep 2017, Noida
10. Baier C, Katoen J-P (2008) Principles of model checking (representation and mind series). MIT Press, New York
11. Anier A, Vain J, Tsiopolous L (2017) DTRON: a tool for distributed model-based testing of time critical applications. Est Acad Sci 66:75–88
12. Larsen KG, Mikucionis M, Nielsen B, Skou A (2005) Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: EMSOFT
13. Santos A, Cunha A, Macedo N, Lourenço C (2016) A framework for quality assessment of ROS repositories. In: 2016 IEEE/RSJ international conference on intelligent robots and systems (IROS), Daejeon, pp 4491–4496
14. Behrmann G, David A, Larsen KG (2004) A tutorial on UPPAAL. In: Bernardo M, Corradini F (eds) SFM-RT 2004. LNCS, vol 3185, pp 200–237. Springer, New York
15. Štill V, Ročkai P, Barnat J (2016) DIVINE: explicit-state LTL model checker. Tools and algorithms for the construction and analysis of systems. Springer, Berlin
16. Ernits J, Roo R, Jacky J, Veanes M (2009) Model-based testing of web applications using NModel. TESTCOM/FATES 2009. Springer, New York
17. Hawes N et al (2017) The STRANDS project: long-term autonomy in everyday environments. IEEE Robot Autom Mag 24(3):146–156
18. Gummel A (2018) Modelbased testing with TestIt: the robot operating system case-study. MSc Thesis, Tallinn University of Technology, Tallinn, Estonia. https://digi.lib.ttu.ee/i/?10616
19. Vaughan R (2008) Massively multiple robot simulations in stage. Swarm Intell 2(2–4):189–208 (Springer)

Appendix 2

Publication II

G. Kanter and J. Vain. Testit: an open-source scalable long-term autonomy testing toolkit for ros. In *Proceedings of the 10th International Conference Dependable Systems, Services and Technologies, DESSERT'2019*, pages 45–50, 2019

The 10th IEEE International Conference on Dependable Systems, Services and Technologies, DESSERT'2019
5-7 June, 2019, Leeds, United Kingdom

TestIt: an Open-Source Scalable Long-Term Autonomy Testing Toolkit for ROS

Gert Kanter[1], Jüri Vain[1]
[1] Tallinn University of Technology, Tallinn, Estonia, {gert.kanter, juri.vain}@taltech.ee

Abstract—**This paper presents an open-source testing toolkit TestIt that is primarily developed for model-based testing of autonomous systems to improve long-term autonomy. The architecture and tools within this architecture are introduced. The main novelty of presented solution is the scalable multi-pipeline testing architecture that enables incorporation of multi-purpose testing tools including those used in state-of-the-art model-based testing. The usability of TestIt for software testing in autonomous navigation context is demonstrated using Uppaal timed automata model based testing and Uppaal-family tools such as model checker and test execution environments Uppaal TRON and DTRON.**

Keywords—autonomous robotics; robot operating system; integration testing; model-based testing; timed automata; simulation.

I. Introduction

Autonomous robotics is growing at a rapid pace as can be witnessed by the fast development of self-driving cars as well as an increase in the number of mobile robots in other sectors. According to Technavio's global autonomous mobile robots market research report[1], the market is expected to grow at a compound annual growth rate of 24% during 2018 to 2022. Double digit compound annual growth rate is expected for all service robot sectors according to International Federation of Robotics report[2]. Also, the sales value of service robots for professional use has increased by 39 percent[3] and the prospect remains positive, as it is currently the primary field for startups.

Autonomous vehicles are designed to operate in a highly dynamic and unpredictable environment, it is difficult to ensure that these autonomous systems function safely. Due to the high complexity and dimensionality of such environments it is difficult to validate the correctness of robot behaviour based solely on real world data. It is exceedingly difficult to gather enough real world data to sufficiently cover the vast number of different scenarios emerging in such complex state space. To help accelerate the validation of the algorithms used in autonomous robots, simulation is used to generate scenarios to ensure correct behaviour.

Simulation is beneficial not only because it is inexpensive as it does not require a physical robot but also because it is scalable. Testing on multiple physical robots requires multiple test sites and evaluation systems which are very expensive. To validate different scenarios at the same time many simulations can be run in parallel in order to conserve time. In order for the simulation to be well integrated into the software production workflow the robot's software stack should support such integration.

Robot Operating System (ROS) [1] has become the de facto standard for developing autonomous robots which also supports simulation integration. According to an International Federation of Robotics report[4], over 66% of all service robot suppliers use ROS.

At present, there is a lack of toolkits that provide model-based testing support for long-term autonomy verification. There has been some efforts made in this area but to the best of the authors' knowledge there are no tools that utilize model-based testing as an integral part of the solution. This strongly motivates the creation of testing toolkits for automated testing of autonomous systems.

In addition, due to continuous integration (CI) servers becoming very widely used in the software development industry [2], there is significant need for such a toolkit to support CI server integration.

In this paper, we present an open-source testing toolkit named TestIt which allows testing ROS based robot software. TestIt has integrated support for model-based testing using Uppaal TA [3]. The proposed toolkit is designed to be scalable to maximize testing efficiency and minimize testing time.

II. Related Work

Currently, there has not been significant effort made in development of model-based testing toolkits designed specifically for autonomous robots. There are several general purpose quality assurance (QA) tools but there is no toolkit that is specifically designed to employ model-based testing of ROS based autonomous robots.

An example of a general purpose QA tool is a CI platform Testributor[5]. Testributor is an open-source continuous integration platform that reduces building times

[1] https://www.businesswire.com/news/home/20181211005561/en/Global -Autonomous-Mobile-Robots-Market-2018-2022-Growth
[2] https://ifr.org/downloads/press2018/WR_Presentation_Industry_and_Service_Robots_rev_5_12_18.pdf
[3] https://ifr.org/downloads/press2018/2018-10-18_Press_Relase_IFR_WR_2018_Service_Robots_ENG.pdf
[4] https://ifr.org/downloads/press/Presentation_PC_11_Oct_2017_1.pdf
[5] http://www.testributor.com

by slicing up the test suite and runs the slices in parallel. This platform is not specifically designed for testing ROS software and it does not support model-based testing natively. In contrast, TestIt toolkit is designed to improve testing workflow and test scenario optimization by using model-based testing. This approach increases testing efficiency compared to Testributor's simple test suite slicing because slicing does not improve the tests themselves. TestIt aims to provide tools to improve the tests themselves as well as provide a highly scalable test execution.

There exists a ROS specific automated test framework (ATF)[6] which has been developed specifically for ROS applications. ATF framework supports executing integration and system tests and running benchmarks. Unfortunately, it is not readily scalable and is designed to run on a single machine. ATF framework also only provides the execution of the test suite but offers no tools to create or optimize the test suite itself. Additionally, the project is currently not in active development.

The ROSIN project report[7] highlights the need for better QA practices to be adopted in ROS software development. One of the main issues is that the QA practices are not consistent across the various development streams (i.e., core, drivers and reusable packages). The report also indicates that the current utilization of CI service is not sufficient because it is simply compiling and building the ROS projects. Their results show that the QA practice would be improved significantly by extending the CI service to run a collection of different kinds of code-scanning tools. The proposed TestIt toolkit addresses some of these concerns by introducing a framework that can be augmented with aforementioned tools and bundled in a convenient package. This toolkit has been designed to be easily integrated with the CI service.

The ROSIN report also highlights that although testing is regarded as critical in robotics, developers working on new components tend to focus more on working on the component rather than creating and setting up tests and gathering data based on simulation. As stated in the report, automated testing is a way forward and will save time and increase software quality. TestIt aims to comply with these recommendations. It reduces time overhead by supporting automation as much as possible, maximizing testing efficiency by using concurrent testing pipelines and minimizing testing time by optimizing test scenarios using model-based testing approach.

III. Architecture

TestIt[8] is a testing toolkit which comprises an open-source ROS package containing the daemon and the CLI (Command-Line Interface) to send commands to the daemon and a Docker[9] container with bundled testing tools.

The daemon can control multiple TestIt pipelines as can be seen in Fig. 1. Test scalability is one of the benefits of simulation-based testing which using multiple pipelines provides. Each pipeline can run on a separate server, for example in the cloud (e.g., AWS, Google Cloud or Azure), ensuring scalability.

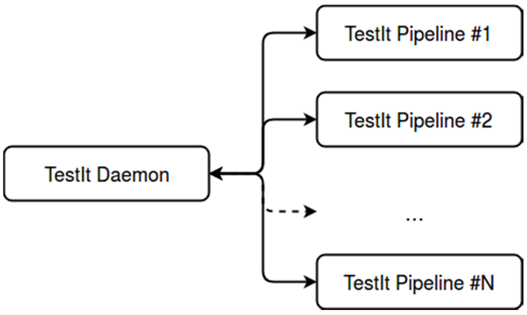


Figure 1. TestIt architecture.

TestIt supports starting and stopping pipeline servers as part of the testing workflow depicted in Fig. 2 which works well in case of using cloud servers for testing as cloud services are billed based on the time used. Therefore, only bringing them online when needed is cost efficient.

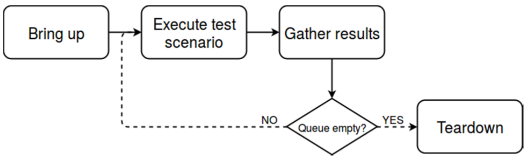


Figure 2. TestIt pipeline testing workflow.

The configuration for the system under test (SUT) and TestIt Docker container is defined in the YAML[10] format configuration file which is passed to the daemon upon start up. The configuration consists of infrastructure configuration and test scenarios.

The overview of a single pipeline is shown in Fig. 3. The infrastructure configuration defines the pipelines that are used for running the test scenarios. As mentioned before, a pipeline comprises the SUT with the software that is going to be tested as well as the TestIt Docker container configuration. The pipelines can be configured as needed depending on the available hardware or budget constraints for the cloud testing. TestIt can even be used in a single pipeline testing configuration with a single computer.

The packaging requirements for the software running in the SUT is not strictly constrained. TestIt toolkit ROS integration relies on the SUT running the ROS master service (i.e., *roscore*) to which the tools in TestIt container can connect. Other than that, the SUT can be considered as a black-box system and TestIt can be used as a black-box testing toolkit which requires no modification by the tester. If test code coverage measurement is required, the SUT

[6] https://github.com/floweisshardt/atf
[7] http://rosin-project.eu/wp-content/uploads/D3.1-Quality-Assurance-Process-and-Community-Management-in-ROS.pdf
[8] https://github.com/GertKanter/testit
[9] https://www.docker.com
[10] https://yaml.org

must be configured in a way that supports it. More information about this feature is provided in Section IV.

As can be seen in Fig. 3, the SUT can either run in simulation or in a hardware-in-the-loop configuration. TestIt is simulator agnostic. The concrete simulator that is used is not fixed thanks to ROS design principles. The simulation communicates directly with ROS along with the SUT software stack and TestIt interacts directly with ROS and not the simulator itself. This allows TestIt to be used in a broader field of robot software development.

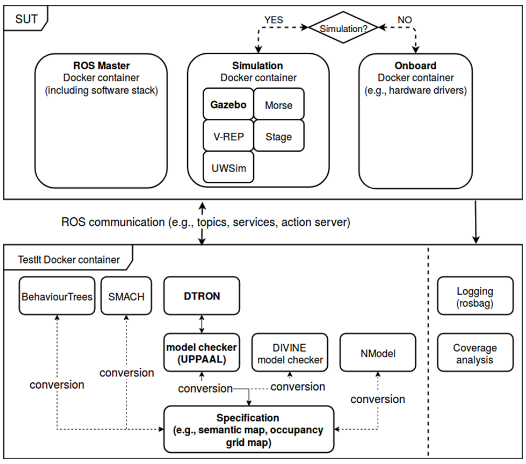


Figure 3. TestIt pipeline.

The test scenarios specify the concrete tests that are executed to test the software. The scenarios can be executed in different ways. The simplest way is to execute a program or script that gives inputs to the software stack (e.g., goals for navigation) but the scenarios can be defined using models. Uppaal timed automata (Uppaal TA) [3] is currently supported for model-based testing using DTRON.

TestIt can be extended to accommodate other model checkers (e.g., DIVINE [4], NModel [5]) in a way that does not inconvenience the toolkit user. This is one of the design goals of TestIt as it is packaged into a Docker container, it is possible to pre-install and configure everything inside the container so that the toolkit user or a CI service does not need to install and configure all of the tools separately.

As depicted in Fig. 3, TestIt Docker container can be bundled with multiple tools that support testing. One of the criticisms of model-based testing is the difficulty and labour intensive procedure of creating models. TestIt toolkit is addressing this issue by supporting generation of models from other specification formats as can also be seen in Fig. 3. The generation of a model from topological map format developed for use in STRANDS project [6] has already been implemented in TestIt [7]. The support for generating models from SMACH state machines, ROS BehaviorTrees or other formats can be achieved in a similar way. Creating models from other specification formats saves time and gives a good starting point to expand the models as necessary. These models can then be used for automatic model-based testing.

IV. TESTIT TOOLKIT

TestIt is designed to be used with existing and new testing tools and supporting software. For example, it is possible to run linters (e.g., roslint[11] and static code analysis tools (e.g., HAROS [8])) as part of the testing process. The main use case for TestIt is however model-based testing.

One of the most important benefits of using MBT is the complex emergent scenarios that can be discovered by simulating both the SUT and the environment (i.e., the static world and dynamic actors in the world) together. It is very difficult to design test scenarios for autonomous systems which test the full software stack thoroughly. This is due to the fact that usually software is developed by different teams and the knowledge of the full software stack is very rare. Accounting for all permutations of conditions that can arise in complex dynamic environments is exceedingly difficult even with full knowledge of the software stack. Using MBT and tools (e.g., Uppaal TA, NModel) helps in this regard by allowing the environment and SUT (i.e., the autonomous system being developed) to be modelled separately. The possibility to model different actors separately reduces the modelling complexity which in turn reduces the cost of testing. After creating the single actor models it is possible to instantiate these models in parallel and in multiple instances. It is possible to create complex behaviour from simple actors.

As discussed in Section II, integrating testing into robotics software development CI processes is highly coveted. TestIt is well suited for integrating into CI services as the testing pipelines are designed to work with Docker containers. Using Docker containers makes it easier to integrate into CI processes because of the ephemeral on-demand nature of the container technology. The containers are always started from the same state and the state is not stored after finishing, which is the desired behaviour in testing context. This feature ensures that testing is stable and there is no risk of influencing the initial state on subsequent test execution. Executing tests without sandboxing the software can run into mutability issues.

Long-term reliability is a key concern for autonomous robots which operate in dynamic environments. Finding software bugs that appear immediately or within a short time window is significantly easier than detecting errors that emerge after a long time has elapsed (e.g., memory leaks and difficult corner cases). Using simulation for long-term autonomy testing with compressed timescale improves efficiency. Long-term testing with real robots is very challenging and is still limited by real-time factor. In many cases, it is possible to perform simulation faster than real-time to further increase the time efficiency of testing. TestIt toolkit supports running tests over long time periods using model-based approach to find interesting scenarios

[11] http://wiki.ros.org/roslint

that are exceedingly difficult to discover with manual test scenario design.

A. *Model-based testing*

TestIt is designed to work with wide variety of tools that are used in the development process. Thanks to the open architecture of TestIt, the user can pick and use the appropriate model-based testing tools. As a concrete example of tool integration, we consider Uppaal tool family into TestIt toolkit.

The modelling formalism of Uppaal tools is Uppaal TA. Uppaal TA are defined as a closed network of extended timed automata. These automata are combined into a single system by synchronous parallel composition. The automata are composed of *locations* (vertices in graphical notation) and *edges* (directed arcs in graphical notation) between the locations. The set of variables associated with the automaton has valuations that are called *state* and the configuration of a model consists of its current control location and assignments to all variables and clocks. The automata can be synchronized using synchronisation links named *channels* between edges. The channels that by modelling naming convention have prefixes *in_* and *out_* in their names are used for sending commands to the SUT as well as receiving feedback from the SUT.

Uppaal TA models are executed using DTRON [9] which extends Uppaal TRON [10], a testing tool based on Uppaal engine. TRON is suited for black-box conformance testing of timed systems. This tool enables simulation of the model in real-time and allows interfacing with the SUT.

In order for the Uppaal TA to be able to send executable test inputs to the SUT, it needs an adapter to handle the synchronization signals and test data from the model. These adapters have to be created depending on the use case. For the use case presented in this paper, we have created a DTRON adapter[12] for sending navigation goals to the SUT and converting the SUT outputs back to the symbolic form interpretable in the model.

B. *Code coverage measurement*

Code coverage is an important metric for software testing. Code coverage measurement is achieved by counting all lines that have been executed. Traditionally, these line counters are never reset during the measurement process which is not an issue in case of measuring the lines that are executed during a full test execution. This approach can be improved with model-based testing by measuring the lines of code that get executed when performing a certain action (represented as a transition in the model). This information can later be used to optimize test scenarios.

In order for the code coverage to be measurable, the SUT software stack must support it. The main programming languages used in ROS software development are C++ and Python. For code coverage, the C++ stack must be compiled with code coverage support options (profile-arcs and test-coverage). Python programs must be run via a wrapper (e.g., Coverage.py library) which collects the code coverage information.

The software stack has to handle the SIGUSR1 signal to support the state transition code coverage measurement. The C++ code needs to call *__gcov_flush()* function to flush the coverage data. For Python, the coverage wrapper must call a save function. If *Coverage.py* is used, *coverage.save()* must be called and internal coverage data must be erased. If the coverage data is not reset, the subsequent calls to the save function will return the full list of lines executed since the start of the program.

C. *Test scenario optimization*

The test actions and SUT reactions logged during test run are automatically annotated with test coverage items showing which part of the implementation code is executed when the given action is triggered. This allows decorating the Uppaal TA edges with weights used for calculating the achieved coverage metrics. Composing such a test model with test trajectory optimizer, generated by Uppaal model checker, allows choosing test sequences that either maximize the code coverage provided a fixed bound of test sequence length is given, or minimize the test sequences provided required minimum coverage is defined.

V. USE CASE

The advantage of model-based testing is demonstrated with a sample use case. In this use case, a Turtlebot\footnote{https://www.turtlebot.com} robot is simulated in Stage simulator [11]. The robot software stack is packaged into a Docker container. The relevant software is depicted in Fig. 4. The SUT container contains ROS navigation stack along with an object detector program and Stage simulator.

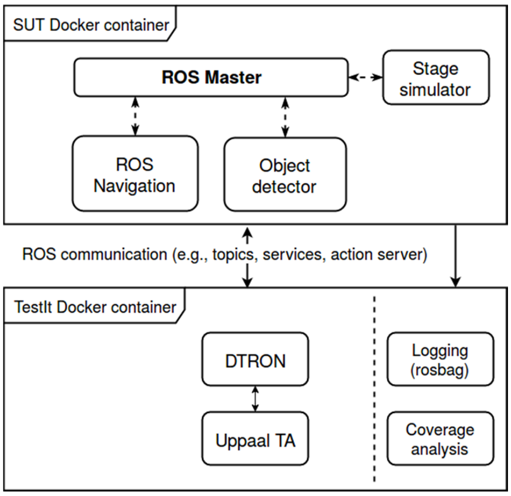


Figure 4. Turtlebot use case software stack.

[12] https://github.com/GertKanter/testit_dtron

The simulated robot environment is shown in Fig. 5. The robot is in a square-shaped room with 20 meter sides (the black dotted lines represent 5 by 5 meter squares). Robot's starting position is in the bottom left corner depicted as a blue circle. The small black circle represents an object to be detected.

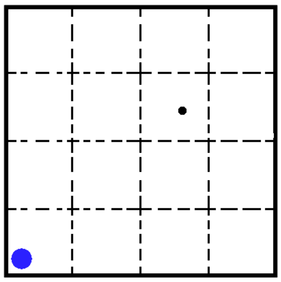


Figure 5. Simple environment with an object.

The software under test in this use case is the object detection program. The object detection program specification states that it should detect an object from the laser scan.

In case black-box testing methodology is used, the actual implementation of the software is assumed to be unknown to the tester. Provided the navigation space is given by the requirements model, the robot should navigate to the object according to this model to find the object. The test is designed to validate that the detection node succeeds in this task. Due to efficiency reasons, we target to optimize the scenario to reach the object as quickly as possible.

The optimized scenario will be generated from an Uppaal trace which includes a situation where the object is in view of the lidar.

Testing is divided into two phases: learning phase and optimized execution phase. In the first phase, the state space is explored randomly while executing the robot software and measuring the code coverage. DTRON generates the sequence of actions to navigate the robot according to the Uppaal model. For this use case the model is depicted in Fig. 6.

The executed lines of code during the exploration are recorded in the model. The number of lines covered are then assigned to the variable *V* as shown in Fig. 7. After annotating the model with these assignments, it is possible to use the model checker with TCTL query *E<>V==maxcov*. This finds the optimal trace which leads to the location with the largest code coverage.

In the second phase, after finding the optimal trace, the robot is guided to that location. This optimized scenario implemented by the test model in Fig. 8 can then be added to the regression testing suite.

The workflow described above can then be reused in regression testing where changes are introduced incrementally in the model and the need for manual or semi-automatic test scripting can be discarded.

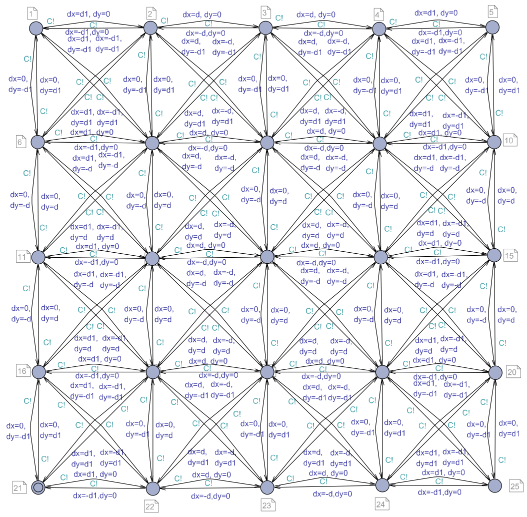


Figure 6. Uppaal TA model.

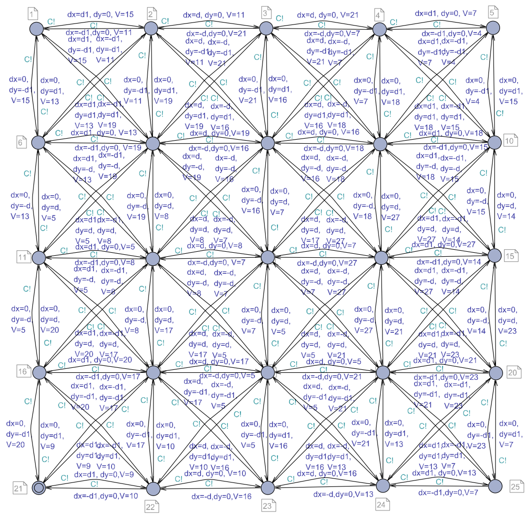


Figure 7. Uppaal TA model annotated with weights.

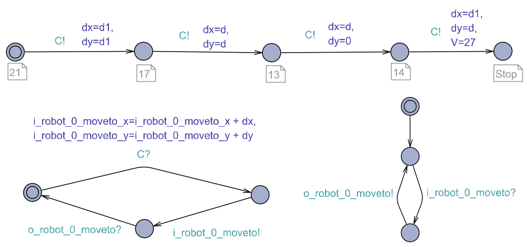


Figure 8. Optimized test model.

VI. Conclusion

This paper presents a testing toolkit named TestIt. TestIt's primary focus is model-based testing of autonomous systems to improve long-term autonomy. The architecture and tools within this architecture are introduced. The main novelty of presented solution is the scalable multi-pipeline testing architecture that enables incorporation of multi-purpose testing tools including those used in state-of-the-art model-based testing.

The usability of TestIt for software testing in autonomous navigation context is demonstrated using Uppaal TA model based testing and Uppaal-family tools such as model checker and test execution environments Uppaal TRON and DTRON.

VII. Future work

In the future, we plan to improve the scenario learning capability by using active machine learning methods in simulation. Currently, the learning phase uses stochastic exploration. Finding interesting operational situations (edge and corner cases) for testing in this way is not optimal. Among other machine learning methods, we would like to explore the efficiency of using reactive planning algorithms [12] for price function-guided state space exploration.

VIII. Acknowledgements

This research was partially supported by the Estonian Ministry of Education and Research institutional research grant no IUT33-13.

References

[1] M. Quigley et al., "ROS: An open-source Robot Operating System," ICRA Workshop on Open Source Software, 2009. [Online].Available: https://www.willowgarage.com/sites/default/files/icraoss09-ROS.pdf

[2] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. "Usage, costs, and benefits of continuous integration in open-source projects," Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 426-437, 2016.

[3] G. Behrmann, A. David, and K.G. Larsen, "A tutorial on UPPAAL, " in M. Bernardo and F. Corradini (eds.), SFM–RT 2004, volume 3185 of LNCS, 200–237. Springer Verlag, 2004.

[4] V. Štill, P. Ročkai, J. Barnat, "DIVINE: Explicit-State LTL Model Checker," Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, 2016.

[5] J. Ernits, R. Roo, J. Jacky, M. Veanes, "Model-Based Testing of Web Applications using NModel," TESTCOM/FATES 2009, Springer Verlag, 2009.

[6] N. Hawes et al., "The STRANDS project: long-term autonomy in everyday environments," in IEEE Robotics & Automation Magazine, vol. 24, no. 3, pp. 146-156, Sept. 2017.

[7] A. Gummel, "Model-Based Testing with TestIt: the Robot Operating System case-study," M.S. thesis, Tallinn Univ. of Technology, 2018, Accessed on: Feb.13, 2019. [Online]. Available: https://digi.lib.ttu.ee/i/?10616

[8] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Daejeon, pp. 4491-4496, 2016.

[9] A. Anier, J. Vain, and L. Tsiopolous, "DTRON: a tool for distributed model-based testing of time critical applications," Est. Acad. Sci. 66, 75–88, 2017.

[10] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. "Testing real-time embedded software using UPPAAL-TRON: an industrial case study," EMSOFT, 2005.

[11] R. Vaughan. "Massively Multiple Robot Simulations in Stage", Swarm Intelligence 2(2-4):189-208, Springer, 2008.

[12] 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, pp. 113-150. IGI Global, 2012.

Appendix 3

Publication III

G. Kanter, J. Vain, S. Srinivasan, and S. Ramaswamy. Provably correct configuration management of precision feeding in agriculture4.0. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 1631–1637, 2019

Provably Correct Configuration Management of Precision Feeding in Agriculture4.0

G. Kanter, J. Vain, Seshadhri Srinivasan and Srini Ramaswamy

Abstract— **Agriculture4.0 aims to use recent advances in technology to enhance productivity and reduce operating costs. In precision feeding systems with robots farm configuration should support robot maneuvers and precision feeding operations. Such configuration needs to be checked during design and operations to avoid costly modifications and damage to robots. This investigation presents an incremental model-based approach for robot farm runtime configuration management. First, an abstract model of the configuration is generated and its feasibility is verified using model checking. Then, simulation based verification by introducing low-level operational details (robot speed, capacity, etc.) is performed which is used to refine formal verification. Finally, operational correctness in real exploitation conditions is tested against the simulation. Our results show that the proposed incremental validation minimizes the validation time, design space exploration and cost during early stages of design, rather than during operations which could lead to significant cost.**

I. Introduction

Agriculture's contribution to global gross domestic product has fallen below 3% and more than 800 million people suffer from hunger worldwide [1]. This has necessitated adoption of novel techniques in farming for not only modernizing the operations but also for optimizing the resources [2]. This has led to the development of the concept of Agriculture4.0 which aims to use recent advances in technology for enhancing productivity and profits from agriculture. While technologies such as the Internet of Things [3]-[5], precision farming [6]-[8], aerial images [9]-[10], data-analytics [11]-[13] and others are becoming a major enabler for Agirculture4.0, human labor is emerging as a scarce resource. Furthermore, hostile environments and scalability of manual operations have necessitated automation of human tasks. Robots have been deployed for farming operations for automating tasks [14], weed recognition [15], plant diseases [16] and smart farming. Within the livestock farming, robots have been deployed for milking [17], precision livestock management [18] and other operations. Robots are becoming important due to their ability to scale, operations over large spaces and hostile environments. Furthermore, robots could save farmers against odour, high ambient noise, night shifts and unfavorable working conditions.

Robotic feeding is targeted to taking care of animals around the clock with precise ration and schedule adjusted to the groups or even individual animals. Health monitoring can be done indirectly by measuring the animals' appetite based on the fodder consumption rate, or more directly, using computer vision algorithms to assess their physical conditions. Farms are deployed on large territories and they may accommodate tens of thousands of animals. Ideally, in the fully automated farm the feeding program can be flexible to be adjusted to animal needs. Automated farming system is also life-critical. Though small deviations in feeding regime do not have direct effect on animals' health, longer breaks in feeding due to mistakes in planning or robots' task coordination may cause fatal consequences to animals' health. Farm animals are sensitive to feeding regularity especially during the gestation period and extreme outdoor temperatures. Thus, the robotic farm should minimize the downtime and keep operating even in degraded mode till the normal operation is restored.

While robotic applications have been exacerbated in existing investigations, often the deployment constraints are ignored. Securing the farm operation with an ample backup fleet of robots is not economically feasible either. To achieve a resilient but economically still feasible farm operation in changing conditions the re-planning procedures need to include also feasibility analysis of possible configurations changes and evaluate the decisions under the operational constraints throughout the farm life cycle. For robots to operate in farms, configuration management is important as they need to operate with farming constraints. Verifying these constraints after deployment could lead to costly farm modifications and damaging of robots. Furthermore, they could also endanger livestock health and productivity. However, such management should also consider low-level information such as robot kinematics, operating speed etc. To our best knowledge a configuration management tool for precision feeding considering robot information is not available in the literature. Existing smart feeding solutions for pig farms, e.g. Pellon [19] and dairy farms, e.g. [20] presume infrastructure in the farm that inherently supports the robot navigation and operation solution. For instance, such farm infrastructure enhancements include magnetic or colour stripes on the floor/wall for navigation, RFID tags on cages are needed for robot positioning, special light conditions for visual inspection, etc. Investments to such augmenting infrastructures bring along also higher maintenance costs due to the need for regular inspection and repair of those instrumentation assets. Another drawback of existing robotic farm solutions is their inflexibility to changing operation

*This research was supported by the Estonian Ministry of Education and Research institutional research grant no. IUT33-13, and in part by industry contract LEP15048.

G. Kanter and J. Vain are with Tallinn University of Technology, Estonia 19086, gert.kanter@ttu.ee, juri.vain@ttu.ee

Seshadhri Srinivasan is with Berkeley Education Alliance for Research in Singapore, Singapore 138602, seshadhri.srinivasan@bears-berkeley.sg
Srini Ramaswamy is with ABB Inc. USA srinitn@gmail.com

conditions. Any farm system upgrade, e.g. extending the farm territory, constructing new buildings, introducing new mechanism, or downgrade due to robots maintenance, needs redesign of the infrastructure and the investments might be far from optimal. Chrisiansen et al. [30] studied the used of model-based control design for single robots, the problem of multiple robot deployment has not been studied in the literature. To overcome shortcomings referred in literature, this investigation proposes a dynamic configuration management for farms using model checking. We start with an abstract model whose feasibility is verified using model-checking and simulation-based verification is done by embedding low-level information (e.g. robot speed). Finally, we test the operational constraints against environmental conditions. The methodology is illustrated on a mink farm in Estonia.

The paper is organized into five sections including the introduction. Section II presents the preliminaries and mink farm case-study is presented in Section III. Farm reconfiguration and its validation is presented in Section IV. Conclusions and discussions are presented in Section V.

II. Preliminaries

A. Modelling with Uppaal Timed Automata

Uppaal Timed Automata (UTA) [21] address the behavioral and timing aspects of systems providing efficient data structures and algorithms for their representation and analysis through model checking. UTA is a network of extended timed automata where each automaton is defined as the tuple $(L, E, V, CL, Init, Inv, TL)$, where L is a finite set of locations, E is the set of edges defined by $E \subseteq L \times G(CL, V) \times Sync \times Act \times L$, $G(CL, V)$ is the set of transitions' guard conditions, $Sync$ is a set of synchronization actions over channels and 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 of initial values to variables and clocks. $Inv : L \rightarrow I(CL, V)$ is a map that assigns an invariant over clocks CL and variables V to each location in L. $TL : L \rightarrow \{ordinary, urgent, committed\}$ is the function that assigns the type to each location of the automaton.

The graphical representation of UTA (see, e.g. Fig. 4) is a directed graph, where locations are represented by vertices and are connected via edges. Locations are labelled with invariants. The invariants are Boolean expressions where the literals are predicates on clock variables and integer constants, e.g. *Clock* <= *const*. The edges are annotated with guards, synchronisation conditions and updates. An edge is enabled if the guard evaluates to *true*. The parameterized instances of automata templates, called processes, synchronize their transitions via channels. The edges of two parallel automata to be synchronous must be labelled with a common channel name. The channel names of synchronous edges are suffixed with ! and ?, respectively where symbols ! and ? denote emitting and receiving end of the channel. Updates that also label edges express the change of the system state when the edge is executed, e.g., *Clock* = 0 resets the value of model clock named *Clock*.

B. Verification with model checking

Model checking is an algorithmic verification technique for checking satisfiability relation $\models$ stated as follows: "Given a model M of a system and its specification φ, check exhaustively and automatically whether this model meets a given specification". Satisfiability is expressed symbolically as $M \models \varphi$. For UTA models the specification is defined in terms of Timed Computation Tree Logic TCTL [22]. We use UPPAAL [23] model checker which is designed to check a subset of TCTL formulas for networks of timed automata. The formulas contain no nested temporal operators and should be one of the following forms [24]:

- A[] ϕ - invariantly ϕ;
- E<> ϕ - possibly ϕ;
- A<> ϕ - always Eventually ϕ;
- E[] ϕ - potentially always ϕ;
- ϕ --> ψ - ϕ always leads to ψ. This is a shorthand for TCTL formula $\forall[](\phi \Rightarrow \forall<>\psi)$, where ϕ, ψ are properties that can be checked locally on a state, i.e. *Boolean* expressions over predicates on locations, integer variables, and clock constraints of model *M*. Note that local property formulas ϕ and ψ above may involve also 1st order quantifiers over integer variable domains.

C. Design validation with model-based testing

Conformance testing with UTA means interpreting symbolic timed traces generated by UTA test model and comparing these interpretations with the i/o traces of System Under Test (SUT). A symbolic timed trace *TTrS* of a UTA model is a sequence of symbolic states, each state being defined as a tuple (*l*, *D*, *v*), where *l* is a location, *D* is the clock constraint, and *v* is a set of non-negative variables' values. A transition from a symbolic state to another state is possible either by an action *a* or by delay *d* denoted as $(l, D, v) \rightarrow_{a/d} (l', D', v')$. The test models expressed in UTA have two partitions, one representing the SUT and the other representing its environment. These partitions are synchronized by input/output actions. The interaction between the system and its environment are identified as observable test actions. During test execution, the internal actions are abstracted as delays. Therefore, in conformance testing the abstract test sequences comprise delays and observable i/o actions only.

Definition: Relativized Timed Input/Output Conformance (rtioco) relation

For input enabled timed input/output labeled transition systems $\mathcal{I}$ ($\mathcal{I}$ denotes SUT in testing context), $\mathcal{S}$ (denotes the model of SUT) and $\mathcal{E}$, (the environment of SUT) relativized timed input/output conformance is defined as:

$\mathcal{I}$ *rtioco*$_{\mathcal{E}}$ $\mathcal{S}$ iff $\forall\sigma \in TTr(\mathcal{E})$:
$Out(\langle \mathcal{I}, \mathcal{E} \rangle \; after \; \sigma) \subseteq Out(\langle \mathcal{S}, \mathcal{E} \rangle \; after \; \sigma)$,

where $TTr(\mathcal{E})$ is a set of timed i/o traces of the environment $\mathcal{E}$, $\langle \mathcal{I}, \mathcal{E} \rangle$ and $\langle \mathcal{S}, \mathcal{E} \rangle$ are observable synchronized i/o actions respectively of I and E and S and E. *Out*($\langle I, E\rangle$ *after* σ) and *Out*($\langle S, E\rangle$ *after* σ) return the sets of output actions of system implementation and specification respectively. The practical interpretation of the given definition is that the implementation conforms to the specification within a shared environment if and only if the observable i/o behavior of the system implementation is always the same as the behavior defined by its specification. The result of conformance testing will be one of the three cases: *passed*, *failed*, or *inconclusive*. If the implementation output is not in the set of inputs specified for the environment or no input/output is provided within the defined time (test timeouts), then the test result is interpreted as *inconclusive*.

III. CASE STUDY: MINK FARM AUTOMATION

The model of mink fur farm consists of entities such as farm houses, feeding robots, and a loading station where food is loaded to robots. The number of mink farm houses can differ from farm to farm, and they tend to be aligned in parallel (see Figure 1a). The standard length of row of mink farm houses varies from 30 meters to several hundred meters. The widths of the entrance and exit is 1.2 to 1.55 meters and they are the narrowest parts the robot must drive through.

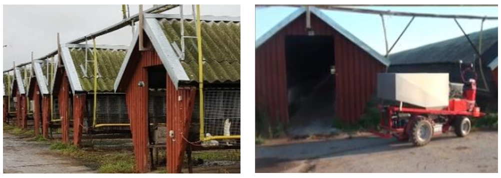


Fig. 1. a) Outlook of the farm; b) Feeding robot.

The loading station (Figure 2) includes a food silo lifted on metal bars and a remotely controlled loading door. For loading the robot has to position itself under the loading door so that the opening above is centred to the robot's food container. When the right position is achieved the robot starts loading by remotely operating the loading door. The loading is stopped by robot when the food level in the robot's container reaches a pre-calculated margin.

The feeding robot (Figure 1b) is a four-wheeled autonomous vehicle with front wheels steering and the rear wheels differential driving. The robot receives sensory inputs from a laser-range scanner, GPS, IMU, and rotary encoders on the back wheel and front wheel kingpins. Actuators control the vehicle steering, driving, and feeding system. Two feeder arms are mounted on the robot to dispense the food on the top of cages at the predetermined locations.

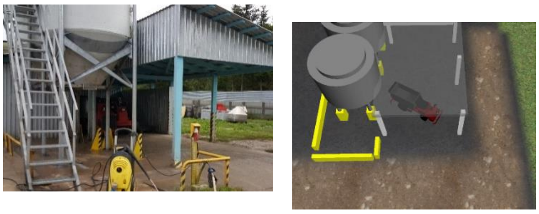


Fig. 2. a) Loading station; b) Simulation view

Fused sensory data are utilized to determine the current location and enable the robot to navigate and perform feeding operations. Aruco tags are placed along the animal cage rows at every 20 m, for localization. The characteristic of the feeding robot are following:

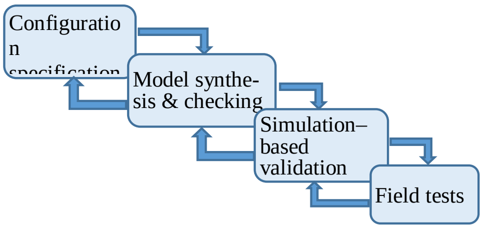
- Maximum indoors speed during feeding is 0.5 m/s
- Maximum vehicle speed outdoors is 1.5 m/s
- Feeding precision is 10 cm inside the placement areas.
- Collision free navigation that means that neither the vehicle nor feeding arm collide with the surroundings.

The causes of farm reconfiguration are following: populating empty farm houses with animals or emptying farm houses, adding/removing robots in operation due to technical maintenance, changing the topology of connecting roads between houses and loading station, changing the speed of robot loading, cruising with and without load and changing food delivery speed indoors. As a result of reconfiguration the following characteristics can change: the number of robots, consumption of food per house, time of loading, point-to-point transport delay of robots, duration of food delivery per house. For automatic generation of farm configuration models all these characteristics are defined as parameters of the model components' templates.

IV. FARM RECONFIGURATION AND ITS VALIDATION

A. *Reconfiguration validation process*

The reconfiguration validation process is divided into incremental stages (Figure 3). The goal of incremental validation is to minimize the validation time and cost. The design and implementation faults detected at an earlier stage are considerably less costly to rectify than the ones detected after system is fully implemented. In the first stage, a formal model is generated from farm configuration specification. The model allows verification of configuration feasibility by abstracting from implementation details. The model is generated using Uppaal TA templates of farm entities. Feasibility verification of farm configuration means proving that configuration satisfies constraints, e.g. "animals get always fed on time and with required amount of food".


Configuration specification
Model synthesis & checking
Simulation–based validation
Field tests
Fig. 3. Farm reconfiguration validation process

In the next validation stage, the reconfiguration feasibility is validated using a simulation model which incorporates implementation aspects that refine the initial verification model. Lastly, when all model-based abstract verification steps are successfully passed the robots are deployed and long-term field test scenarios are executed to validate the simulation results in the real farm environment. In case some of the validation stages fails the process backtracks to the configuration specification stage for finding an alternative solution. After the configuration correction, the validation process is repeated. Note that the configuration correction can be assisted by the diagnostics provided by the earlier model checking stage to localize the cause of feasibility violation.

B. *Generating high-level farm model*

The configuration model synthesis is based on composing model templates each corresponding to some farm functional entity. For verification the model templates are instantiated with configuration specific parameter values such as the number of farm houses and robots, house traversal time, travel time between the robot navigation waypoints in the territory and indoors.

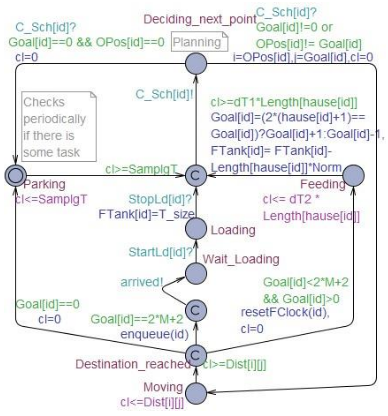
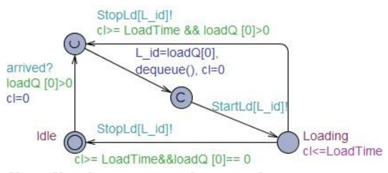


Fig. 4. Model template *Robot*

The active entities of the farm model are feeding robot, loading station, global planner (task broker), and robot's on-board planner. The rest of attributes of farm entities are represented as variables in the model. Composed farm model specifies an observable behaviour and interactions of entities.

Feeding robot (template *Robot*, Fig. 4). The template is parameterized by robot ID. The states of a robot are: *Parking*, *Planning* next action, *Moving* to the action location, *Loading* food, and *Feeding* animals. Loading is divided into two sub-states: *Waiting* in the queue and *Loading*.

Fig. 5. Model template *Loader*



Initially, all robots are in their parking position and sampling if there are tasks in the tasks queue. For a task to be allocated, the robot communicates with the Broker which decides on the next task (defines the house to be fed). The tasks to be executed are kept in the *Broker's* task list. The task acceptance decision is made by robot in the state *Planning* (on-board planning actions are refined in the template *Planner*). To accomplish the feeding task the robot needs enough food in its tank. If the tank is full for a task the robot navigates to the destination building and starts feeding there. If the tank is not filled the robot moves to the loading station and registers in the waiting queue (function *enqueue*). After loading, the robot returns to planning state to pick a new task.

Loading station (template *Loader*, Fig. 5) operates the loading queue of robots that have registered for loading. When the queue is empty the *Loader* returns to state *Idle*. The *Loader* activates in the arrival of any robot. Each time some robot finishes loading its ID is removed from the queue (function *dequeue*) and the waiting robots' IDs are shifted one position further in the queue.

Global planner (template *Broker*, Fig. 6) periodically samples the status of the food in the feeding houses to decide on the next feeding task. When the time comes to feed a house the corresponding task (defined by house number, the cage row and deadline) are generated and added to the tasks list. Whenever a task is taken by some robot the task status changes to *assigned*, and when it is completed the task is removed from the tasks list.

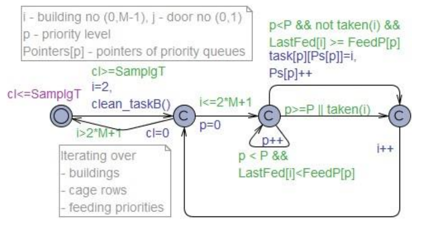


Fig. 6. Model template *Broker*

Robot's *on-board planner* (template *Planner*, Fig. 7) models the robot decision process of choosing next actions. After the task is chosen from the *Broker*'s task list the *Planner* decides if it can be completed with the food amount it has in the tank. If not the Planner chooses task *loading*. Otherwise, the robot's individual goal is updated with the task assigned the highest priority by *Broker*.

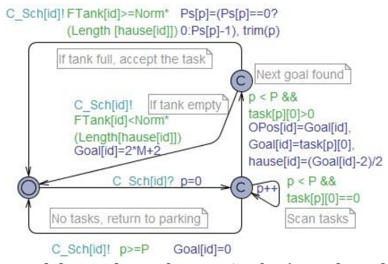


Fig. 7. Model template *Planner* (*Robot*'s on-board planning)

C. *Formal verification of the farming system*

After model generation the validation process starts with model checking to prove that the farm operation under given configuration constraints does not violate the farm feasibility criteria. The primary criterion is that animals get fed with required amount of food and the feeding intervals meet the prescribed constraints, that is, each feeding house should be fed with right intervals where the timing deviations can vary only within some prescribed error margin. The verification goals are formalized as TCTL formulas in which following farm model terms can be referred to: *Constants*: L - number of loading stations; N - number of robots; M - number of buildings; K - number of grids in one row of the building; P - the number of feeding priority groups; $dT1/dT2$ – lower/upper bound of the food delivery time per grid; *Tick* - model time step; *SamplgT* - period of sampling the need for feeding; *Horizon* – verification time horizon; *FeedP*[*P*] – vector of criticality thresholds of feeding delays (*FeedP*[0] corresponds to the highest priority); *Norm* – size of the portion per animal; *T_size* - the size of on-board food tank; *LoadTime* - maximum duration of loading the tank; *Length*[*m*] - number of grids in one row of *m*-th building; *Dist*[*i*][*j*] - durations of moving between *i*-th and *j*-th outdoor objects of the farm.

Variable arrays: *LastFed*[*m*] – clock variables counting time since the last feeding of the *m*-th house; *task*[*p*][*m*] – two dimensional array of feeding tasks, where $p \in [1, P]$ is index showing the priority queue and $m \in [1, M]$ index of the building to be fed in the task. *Goal*[*n*] - the destination of *n*-th robot to navigate to; *OPos*[*n*] – location the *n*-th robot departed from when navigating; *FTank*[*n*] – amount of the food left in the *n*-th robot tank; *house*[*n*] - the building *n*-th robot is heading to or is feeding; *loadQ*[.] – the queue of robots waiting in the loading station.

Functions: *enqueue*(*n*) – adding *n*-th robot in the loading queue; *dequeue*() – removing a robot from the loading queue after loading; *taken*(*i*) – returns *true* if the *i*-th task in the task queue is assigned to some robot, *false* otherwise.

The weakest feasibility constraint (1) requires that for all buildings the feeding can be delayed at most *delta* after the building feeding has reached the highest priority, i.e. the delay from last feeding exceeds an upper bound *FeedP*[0], and, on the other hand, it is not more frequent than the feeding interval lower bound *FeedP*[2].

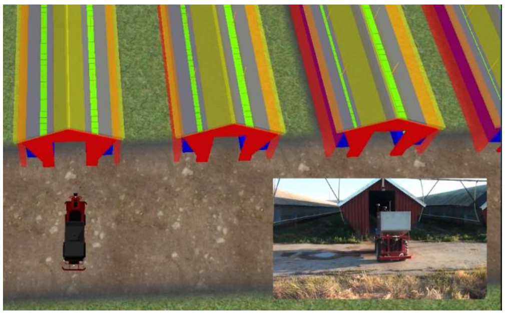
$$\text{A[] forall } (i\text{: } [1,M])\ LastFed[i] < FeedP[0] + delta \qquad (1)$$
$$\&\&\ LastFed[i] >= FeedP[2]$$

Normally, it is expected that the feeding period is within interval [*FeedP*[2], *FeedP*[1]], i.e. not letting any feeding house to wait longer than specified by second highest priority value *FeedP*[1]. It means if there is no emergency the condition (1) reduces to (2).

$$\text{A[] forall } (i\text{: } [1,M])\ LastFed[i] < FeedP[1] \qquad (2)$$
$$\&\&\ LastFed[i] >= FeedP[2]$$

D. *Simulation based validation*

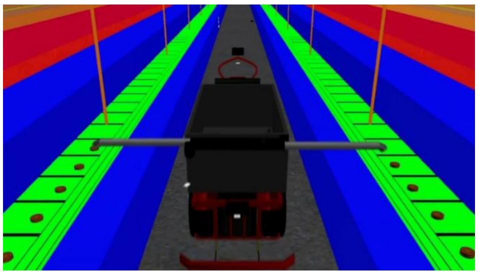
Simulation based validation means running test scenarios in the simulation world. The simulation world is dynamically generated using parameterized generator scripts. The parameters are specified based on the Uppaal model checking results. The properties verified in Uppaal guide selecting the specific configuration of the simulation, e.g., the number of houses and robots. Simulation experiments enable validation of the configuration in deeper level of detail which in turn can be tested in a full real world scenario. A screenshot of the simulated scenario on a real robot is shown in Fig. 8. Any significant deviation from simulation are detected by runtime monitor and analyzed in order to localize the source of the problem. As the simulation is never exactly the same as real



world, improvements can be made to the simulation models and parameters to better match the real world conditions. Fig. 8. Validation of farm scenarios against simulation

Fig. 9 depicts an excerpt of simulated feeding scenario. The feeding system is designed to maximize efficiency by simultaneously dispensing fodder on both rows of cages. The amount of fodder dispensed per cage is reconfigurable and can be tailored to suit the specific requirements of the animals (based on age or other characteristics).

Fig. 9. Simulated feeding scenario



E. *Testing workbench TestIt for robotic systems*

The Gazebo simulator [28] based validation of the system was performed using the testing toolkit TestIt [27]. At its core, the TestIt toolkit is a test runner and test result aggregator but it also enables the tests to be more efficient and useful when combined with model-based testing techniques and tools. TestIt toolkit enables tests to be run in parallel in flexible configuration which scales well with hardware. The tests were executed on Amazon Web Services [25] cloud servers in parallel to minimize testing time. TestIt toolkit is designed to work with Docker containers [29]. This is not a strict requirement and the TestIt tool can be used in a variety of ways as a test runner and result aggregator. In the robotic farming use case, we have the SUT software stack running in one container and the tests are running in another container (TestIt container). This follows the traditional separation of concerns principle and enables black-box testing. Black-box testing allows the testing process to be unbiased and is more flexible as the tests are not closely coupled to the source code implementation. In this use case we utilize an extension of TestIt which incorporates model-based testing tool Dtron [27]. Dtron enables executing Uppaal models as abstract tests. For this use case the tests were divided into two categories: navigation and feeding. The navigation tests focused on testing whether the navigation algorithms were able to maneuver the robot successfully and the feeding tests focused on the correctness of feeding process. The tests were asserted as successes if the robot completed the scenario in the allotted time. The completion of tasks was evaluated by an oracle which was running as a separate stand-alone process.

V. CONCLUSIONS AND DISCUSSION

This paper presented an approach for incremental configuration management and configuration validation for robotic feeding in Agriculture 4.0. The method proposed had several modelling and validation steps that gradually refine the farm description and verified feasible operation. The approach was incremental in the sense that abstract model was generated automatically and once verified, it served as a starting point for the simulation model adding low-level information. We presented a case study of a mink farm and our studies showed that as lower level information is added, the validation time and complexity increased substantially. Consequently, the proposed approach could be used to detect configuration fault early during the modelling and validation stages saving reconfiguration cycles and extensive simulation experiments. It also eliminates the need for more intensive field tests and only a final validation test is required. As against existing results wherein only model-based simulations have been studied with a goal for single robot design, the proposed approach evaluated the feasibility of the entire farm operation by integrating complimenting techniques: formal verification, simulation, and model-based testing. Partial validation by simulation experiments and field tests is supported by exhaustive state space exploration using formal model checking, thereby providing re-use capability to system components that constitute the configuration builds through system life time.

References

[1] CLERQC, M., A. Vats, and A. Biel. "Agriculture 4.0: The Future of Farming Technology." Nova Iorque: Oliver Wyman (2018).

[2] Hamrita, Takoi K., E. W. Tollner, and Robert L. Schafer. "Toward fulfilling the robotic farming vision: Advances in sensors and controllers for agricultural applications." IEEE Transactions on Industry Applications 36.4 (2000): 1026-1032.

[3] Ramachandran, V., R. Ramalakshmi, and Seshadhri Srinivasan. "An Automated Irrigation System for Smart Agriculture Using the Internet of Things." 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV). IEEE, 2018.

[4] Dlodlo, Nomusa, and Josephat Kalezhi. "The internet of things in agriculture for sustainable rural development." 2015 international conference on emerging trends in networks and computer communications (ETNCC). IEEE, 2015.

[5] TongKe, Fan. "Smart agriculture based on cloud computing and IOT." Journal of Convergence Information Technology 8.2 (2013).

[6] Auernhammer, Hermann. "Precision farming—the environmental challenge." Computers and electronics in agriculture 30.1-3 (2001): 31-43.

[7] Blackmore, Simon. "Precision farming: an introduction." Outlook on agriculture 23.4 (1994): 275-280.

[8] Ahmed, Faraz, et al. "Survey on Precision Farming using Mobile Applications." Global Journal of Computer Science and Technology (2019).

[9] Potena, Ciro, et al. "AgriColMap: Aerial-Ground Collaborative 3D Mapping for Precision Farming." IEEE Robotics and Automation Letters (2019).

[10] K. Visalini, Giovanni Palmieri, K. Bekirogulu, S. Thiyaku, B. Subathra, and Seshadhri Srinivasan, "Sensor Placement Algorithm with Range Constraints in Precision Agriculture", IEEE Aerospace and Electronic Magzine, 2019.

[11] Sowmya, B. J., et al. "IOT and Data Analytics Solution for Smart Agriculture." The Rise of Fog Computing in the Digital Era. IGI Global, 2019. 210-237.

[12] Grossi, Marco, et al. "Sensors and Embedded Systems in Agriculture and Food Analysis." Journal of Sensors 2019 (2019).

[13] Elijah, Olakunle, et al. "An overview of Internet of things (IoT) and data analytics in agriculture: Benefits and challenges." IEEE Internet of Things Journal 5.5 (2018): 3758-3773.

[14] Zhang, Dan, and Bin Wei. "From Manual Farming to Automatic and Robotic Based Farming." Robotics and Mechatronics for Agriculture (2017): 121.

[15] Kounalakis, Tsampikos, Georgios A. Triantafyllidis, and Lazaros Nalpantidis. "Weed recognition framework for robotic precision farming." 2016 IEEE International Conference on Imaging Systems and Techniques (IST). IEEE, 2016.

[16] Ampatzidis, Yiannis, Luigi De Bellis, and Andrea Luvisi. "iPathology: robotic applications and management of plants and plant diseases." Sustainability 9.6 (2017): 1010.

[17] Bach, Alex, and Victor Cabrera. "Robotic milking: Feeding strategies and economic returns." Journal of dairy science 100.9 (2017): 7720-7728.

[18] Halachmi, I., and M. Guarino. "Precision livestock farming: a 'per animal'approach using advanced monitoring technologies." Animal 10.9 (2016): 1482-1483.

[19] Pellon URL. https://www.pellon.fi/en/pig_husbandry/ feeding/ pig_robot/

[20] Wbur URL. https://www.wbur.org/hereandnow/ 2017/08/10/ midwest-dairies-robots/

[21] Behrmann, G., David, A., Larsen, K.G. (2004) A tutorial on uppaal. In: M. Bernardo, F. Corradini (ed.), SFM-RT 2004, *LNCS*, vol. 3185, Springer Verlag. 200 - 237.

[22] Alur, R., Courcoubetis, C., and Dill D.L. (1990) Model-checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press. 414–425.

[23] Uppaal URL. http://www.uppaal.org/

[24] Bengtsson J., Yi W. (2004) Timed Automata: Semantics, Algorithms and Tools. In: Desel J., Reisig W., Rozenberg G. (eds) Lectures on Concurrency and Petri Nets. ACPN 2003. Lecture Notes in Computer Science, vol 3098. Springer, Berlin, Heidelberg. 87-124.

[25] Amazon Web Services URL. https://aws.amazon.com/

[26] TestIt URL. https://github.com/GertKanter/testit/

[27] Dtron URL. https://cs.ttu.ee/dtron/

[28] Gazebo simulator URL. http://gazebosim.org/

[29] Docker URL. https://www.docker.com/

[30] Christiansen, M.P., Larsen, P.G., Jørgensen, R.N. (2016) Agricultural Robotic Candidate Overview using Co-model Driven Development. International Conference on Integrated Modeling and Analysis in Applied Control and Automation. Bruzzone, Dauphin-Tanguy, Junco and Longo (ed.), 41- 47.

Appendix 4

Publication IV
J. Vain, G. Kanter, and A. Anier. Learning timed automata from interaction traces. In *14th IFAC Symposium on Analysis, Design, and Evaluation of Human Machine Systems, HMS 2019*, volume 52-19, pages 205–210, 2019

Preprints, 14th IFAC Symposium on
Analysis Design and Evaluation of Human Machine Systems
Tallinn, Estonia, Sept. 16-19, 2019

Learning Timed Automata from Interaction Traces [⋆]

J.Vain, [*] **G.Kanter,** [*] **A.Anier** [*]

[*] *Tallinn University of Technology, Akadeemia tee 15A, Tallinn, 12618 Estonia (e-mail: juri.vain@ taltech.ee).*

Abstract:
The design of load-critical human-machine systems presumes thorough modelling and analysis of interaction profiles the systems are meant to withstand at peak loads. The need for mathematical modelling of interactions is often ignored due to significant modelling effort and lack of relevant tools. We propose an algorithm for automatic learning a subclass of Uppaal timed automata models from system and its environment interaction logs. The learning method relies on synchronous communication assumption that is characteristic to communication protocols of networked HMS distributed components. The method is demonstrated on IEEE1394 protocol learning example. Beside enhancing automatic test generation, the learned model allows verifying test feasibility and test optimization already in early phases of test design.

Keywords: Machine learning, timed automata, human-machine interaction, leader election protocol, load testing.

1. INTRODUCTION

Human-machine systems, such as early warning of natural disasters, rescue planning from disaster areas, and other are not only time critical but should also withstand sharp load fluctuations in the crisis situation. The performance issues manifest themselves when the number of entries, e.g. phone calls, to the system suddenly grows in short time and the services availability may drop noticeably causing denial of life critical services. Anticipating sharp load fluctuations in such circumstances without thorough modelling and analysis of user behaviour is often source of ambiguous requirements and rises the risk of infeasible solutions in critical HMS design. One frequent reason of insufficient availability and scalability of an interaction intensive HMS, is the design practice where instead of systematic model-based development (MBD) often agile and heuristic approaches are preferred. According to Dias Neto et al. (2007) the most frequent reasons why rigorous model-based system development methods have not been easily accepted in software industry are:

- considerable modelling effort,
- model-based approaches have poor integration with current software development practices,
- model-based methods lack empirical evaluation from industrial environments.

Model-based analysis of HMS environment behaviour and proper specification of system requirements are not needed only for design activities but also for testing to what extent the system implementation conforms to its requirements. Performance testing of HMS presumes generating the test cases that represent different load patterns from human controlled environment. One most natural source for extracting data about load characteristics is the interaction log which shows how real users have interacted with the systems of similar purpose and expectedly would interact with the system under design. Shaw (2000) states that in performance testing, it is important that the traffic generated from workload models mimic the load generated by real users as closely as possible. Otherwise it is not possible to draw any reliable conclusions from the test results. In this paper we address the problem of reducing the modelling effort in model-based testing of load critical HMS and search for solutions of constructing test models by means of machine learning methods. Learning the system load profiles from realistic i/o logs provides, in the first place, the test data to be tried with the system under test (SUT). Second outcome of the learning method is extracting the test scenarios that determine the order and frequency of applying test data in the situations under test. The model learning algorithm proposed in this paper for automatic model-based test generation presumes exploring the time stamped interaction logs that are recorded when monitoring the traffic between the ports of distributed HMS and its environment or traffic between its components. The learning algorithm outputs the model in the form of a subclass of Uppaal timed automata (TA) (Behrmann et al., 2004) where all input actions in system side and corresponding output actions in the environment side are assumed to be synchronous. Same applies symmetrically to output actions of the system and corresponding input actions from the environment. The learning approach we propose is demonstrated on IEEE1394 distributed leader election protocol. This protocol is used in ad hoc radio communication networks that are often part of the critical HMS infrastructures. Another reason of choosing this case study for learning models for load testing is that all the nodes in IEEE1394 leader election infrastructure follow the same protocol and it is

[⋆] This research was partially supported by the Estonian Ministry of Education and Research institutional research grant no IUT33-13.

easy to validate the model learned against the model composed manually by the description of standard IEEE1394. Finally, the modularity and symmetry of studied protocol allows easy rescaling of models by changing the number of nodes and running different load patterns that put system under the stress. The load tests are considered to be passed if the load patterns are responded properly by HMS and its reconfiguration procedures.

2. TIMED AUTOMATA LEARNING: RELATED WORK

The construction of automata models by observing system i/o behaviour is regarded as an automata learning problem (Freund et al., 1993). For finite-state reactive systems, the *active learning* means constructing a (usually deterministic) finite automaton from the answers to a finite set of membership queries, each of which asks whether a certain sequence of input symbols (observed events) is accepted by the automaton or not. There are several techniques (for overview we refer to Angluin (1987), Kearns and Vazirani (1994)) which use the same basic principles. They differ by how membership queries are chosen and how an automaton is constructed from the answers. The techniques guarantee that a correct automaton will be constructed if sufficient information is obtained. In order to check the sufficiency of learning sets, the equivalence queries are used that ask whether a hypothesized automaton accepts the correct sequences of symbols. Such a query is answered either by "yes" or by a counterexample on which the hypothesis and the correct language disagree. In Grinchtein et al. (2004) the algorithm of Angluin (1987) is extended to *event recording timed automata.* This class is restricted to be event-deterministic meaning that timing constraints for the occurrence of an action depend only on the past sequence of actions, and not on their relative timing. Another timed automata learning method has been proposed in Tappler et al. (2018). This model generation approach is based on genetic programming method. The technique is entirely passive, i.e. the model is learned from a set of timed traces (observations), collected beforehand by random testing. As an alternative to learning methods discussed above a passive learning method of Uppaal TA is proposed in this work. While referred approaches construct ordinary timed automata (without data variables), our algorithm learns Uppaal TA which are extension of ordinary timed automata. Due to the application specifics of learning outcome, that is HMS load test model construction, the method is built upon following assumptions:

(1) The algorithm must be incremental to implement *online learning* strategy, i.e. the learner does not have a possibility to back-track and ask equivalence or membership queries about the arbitrary length prefixes of the learning input trace. The algorithm constructs a complete (relative to the input trace) non-deterministic timed I/O automaton, i.e. observation sequences learned are reproducible by automaton.
(2) Since the aim is to learn interactions between multiple automata instead of learning a single automaton, the algorithm has to construct the *parallel composition of interacting* Uppaal automata. Two alternative cases of automata synchronization within their composition can be considered:
    - *Case* 1: the processes communicate via i/o variables and synchronize only by means of *clock constraints*. That is because the forward stability of synchronization hypothesis cannot be guaranteed in the *incremental* and unsupervised learning of unpredictable interactions. To ensure the incremental learning (for details we refer to Ade et al. (2013)) it has to be guaranteed that the model built based on past observations will not be backtracked during learning. If a new observation would violate the synchrony hypothesis (made based on past observations) then potentially extensive backtracking is required to change the synchrony attributes in the model.
    - *Case* 2: If the communication of processes over i/o variables is assumed to be synchronous the forward stable synchronization assumption is due to the fact that observable communication actions always incorporate both communicating parties and the learner can record the fact of synchrony in the first occurrence of such an input-output action pair.
(3) The algorithm needs to use predicate abstraction for clustering the events and for encoding their occurrence conditions on clock and state variables. The transition guards and location invariants, both possibly non-deterministic, are constructed in the form of linear interval constraints. Monotonous expansion of the interval bounds during leaning helps keeping the balance between the learning algorithm complexity and the precision of the resulting model.

The algorithm which covers the *Case* 1 has been proposed in Vain et al. (2009). It assumes that system input-output actions are asynchronous and their timing refers to global observer's clock. These assumptions are motivated again by the application context, i.e. learning surgeon's and scrub nurse's collaborative movements during surgeries. Here, in case of unsupervised learning, the learner does not have knowledge if the motions observed simultaneous once are causally related or they just happened to be simultaneous accidentally. Therefore, in general, the traces used for robot training are not supposedly *forward stable* (Morin, 2008) regarding synchronization. The idea of learning timed automata from logs to encode load patterns is not completely new. Probabilistic Timed Automata (PTA) have been proposed in Abbors et al. (2013). The probability estimates accumulated by log analysis give metrics for prioritizing the test sequences to cover most typical load situations. On the other hand, the bugs occurring in behaviours of low probability may remain undetected and can cause considerable damage in rare but critical situations. This motivates the usage of *non-deterministic* TA instead of PTA in testing critical HMS. We assume implicitly that non-deterministic choices encoded in the model are *strongly fair* (Francez et al., 1979). Another assumption is synchrony of communication actions. The data communication between the parallel components of the system once observed synchronous remains synchronous in further communication. Thus, the traces are assumed to be forward stable regarding synchrony and the use of synchronization constructs of Uppaal TA models is justified. Last but not least, the traces extracted from logs and used for learning are assumed to be recorded at the same ports of SUT that will be used later as test ports.

3. PRELIMINARIES: UPPAAL TIMED AUTOMATA

Uppaal TA (Behrmann et al., 2004) are defined as a closed network of extended timed automata that are called *processes*. The processes are combined into a single system by synchronous parallel composition. The nodes of the automata graph are called *locations* and directed vertices between locations are called *edges*. The *state* of an automaton consists of its current location and assignments to all variables, including clocks. Synchronous communication between processes is done by synchronisation links called *channels*. A channel relates a pair of edges in parallel processes where synchronised edges are labelled with symbols for input and output actions (denoted `ch?` and, respectively, `ch!`). Formally, Uppaal TA is a tuple (L, E, V, CL, $Init$, Inv, T_L), where:

- L is a finite set of locations,
- E is the set of edges defined by $E \subseteq L \times G(CL, V) \times Sync \times Act \times L$, where:
  - · $G(CL, V)$ is the set of constraints in guards,
  - · $Sync$ is a set of synchronisation actions over channels and
  - · Act is a set of integer and boolean assignments and clock resets.
- V denotes the set of integer and boolean variables.
- CL denotes the set of real-valued clocks $CL \cap V = \emptyset$
- $Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks.
- $Inv : L \to I(CL, V)$ is a function that assigns an invariant to each location, with $I(CL, V)$ denoting the set of invariants over clocks CL and variables V.
- $T_L : L \to \{ordinary, urgent, committed\}$ assigns the type to each location

4. UNSUPERVISED LEARNING OF UPPAAL TIMED AUTOMATA WITH SYNCHRONOUS COMMUNICATION ASSUMPTION

The learning algorithm introduced in this paper though inspired by Vain et al. (2009) relies on different assumptions. It takes the log of input-output events monitored on ports of system and constructs the Uppaal TA model where processes communicate over shared variables and synchronize using both clock constraints and channels. The set of model locations is not known in advance, it is generated in the course of learning process by identifying the equivalence classes of system inputs. Since the interactions between system and its environment need to be learned we distinguish the components of environment by ports of the system they have direct access. The automata that model environment components are assumed to be output deterministic, i.e. there is one-to-one correspondence between the locations of environment automata and the equivalence classes of environment components' outputs.

4.1 Assumptions of model construction

The assumptions coming from the learning context are summarised in the following: (i) The model learned for model-based testing includes two partitions: the components of SUT and the components of environment Env. The interactions between the components of SUT and Env are observable as events that update the values at components' ports. Each port is allocated to only one component either that of SUT or Env. Thus, a link between components is identified by the pair of ports it is connecting. (ii) The communication between ports is unidirectional. A connection between ports P_i^E and P_j^S belonging to environment component P^E and SUT component P^S respectively, is modelled in the Uppaal TA as a channel $ch_{ij} = \langle P_i^E, P_j^S \rangle$ from the i-th output port of P^E to the j-th input port of P^S. (iii) The learning objective is to construct a model that represents how environment chooses inputs of SUT after SUT has responded to the earlier stimuli from environment. Since the model learning algorithm is targeted to load test generation, the specific control structure of SUT model can be ignored. We simply assume the input enabledness of SUT w.r.t. all of its ports. Technically, it suffices introducing an automaton for each SUT component where the automaton has canonical control structure like proposed in Brinksma (1989). The SUT component automaton responds either by writing data to its output ports or by executing unobservable internal actions that results in a special *timeout* event issued by network monitor. (iv) From Environment perspective the set E of events observable at ports consists of two subsets $E = E^{SUT} \cup E^{Env}$, where $E^{SUT}$are the events produced by SUT (*observable* to Env events), and the events E^{Env} produced by the components of the environment (*test controllable* events). It is assumed that set E^{SUT} includes also timeout events (TO), i.e. initiated when SUT does not respond to input within given time bound. (v) The event log $Lg(E)$ starts with the Environment event and ends with the event produced by SUT, i.e. $e_1 \in E^{Env}$ and $e_n \in E^{SUT}$, where $n = |Lg(E)|$ (vi) The observations of events e_i in the log are recorded as triples $\langle P, TS, X \rangle$, where - the pair of ports ($P = \langle Port_i, Port_j \rangle$) identifies the channel ch_{ij} between send and receive *processes* (further, for shorthand we use channel label ch_{ij} instead of the pair of ports); - TS is the network monitor's clock timestamp; - the vector X of data variables communicated between ports is denoted by $X = X^i$ if data propagate from $Port_i \in P^E$ to $Port_j \in P^S$ and by $X = X^o$ if data propagate from $Port_i \in P^S$ to $Port_j \in P^E$; - timeout as special event is recorded in the form of triple $\langle ., TS, * \rangle$, where for all i, $x_i^{out} = *$ ("." and "*" are wildcard symbol for channel and for variable x_i^{out} value). To treat the symbol * uniformly with numeric values we extend the semantics of standard functions min and max as follows: $min\,(*,\, x) = min\,(x,\, *) = max\,(*,\, x) = max(x, *) = x$.

4.2 Building blocks of the constructed model

In this subsection the Uppaal TA elements from which the learned model is built are introduced. (i) Two types of edges of environment automata are distinguished by their channels direction: the edges are *controllable* if they model controllable events and *observable* if they model observable events. (ii) The locations with observable incoming edges and controllable outgoing edges are called *active* and the locations are *passive* if they have controllable incoming edges and observable outgoing edges. (iii) The non-deterministic invariant and guard conditions are specified with closed intervals $[lb, ub]$ with lb being lower and ub upper bound respectively. (iv) The updates on outgoing edges in the environment automata are identified by values of X^i sent by the environment, and by values of X^o of the observable event. The guard condition of the observable edge that is incoming to current active location is identified as in Algorithm 1.

Notation

- `g,asg,inv,ch` are meta-variables that denote the model syntax elements such as guard condition, assignment, invariant and channel respectively;
- $\bar{x}$ and $\bar{cl}$ denote valuation of variables cl and x;
- h is the stack of generated location names;
- Interval extension to the length R: $[x^-, x^+]^{\updownarrow R} = [x^- - \delta, x^+ + \delta]$, where $\delta = R - (x^+ - x^-)$, and x^-, x^+ are interval lower and upper bound respectively;
- . denotes the concatenation of terms in syntactic expressions (term is either an atom if within quotes, e.g. '`clock <=` ', or the value of an expression otherwise);
- .. denotes unspecified value (used when the model terms have not been fully constructed yet).

4.3 Implementation of the Algorithm

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

Algorithm 1 Learning Uppaal TA with synchronous communication assumption

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

5. CASE STUDY: MODEL LEARNING FOR PERFORMANCE TESTING OF IEEE1394 PROTOCOL

The IEEE 1394 protocol standard specifies leader election protocol. The leader is needed to control a communication bus after network reset. Initially all nodes in the network have equal status, and they know only which neighbour nodes they are connected to. The leader, after elected, becomes the root of the tree provided the network is connected and acyclic. Each node proceeds in two phases depending on the number of children and the number of neighbours still to negotiate with. If there is more than one neighbour, the node waits for requests from its neighbours to become their parent. If there is only one neighbour and this neighbour is not a child, then the node sends a request to this neighbour to become its parent. This implies that leaf nodes are the first to communicate with their neighbours, and the spanning tree is built from the leaves towards root. Furthermore, the protocol may not proceed in one run because the parent requests are not atomic and contention may arise (two nodes simultaneously send the parent requests to each other). Since only one of the conflicting nodes can be a parent, the contention must be resolved. This is achieved by random waiting time before next attempt. Both conflicting nodes choose randomly whether to wait for a long or short time interval. If, after the wait period is over, there is a parent request from the other node, then the node becomes a parent. If there is no such request, then the node resends its own parent request and contention may result again. In Figure 1 an example of network consisting of four nodes is depicted where central

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

node Node #0 is the system under test and the other nodes Node #1 to Node #3 constitute the Environment the test needs to emulate. Thus, the learning goal is to construct the model of Environment i/o behavior by studying the logs recorded on interfaces between SUT and Environment nodes. Let Req_{ij} and Ack_{ij} denote parent request from node i to node j and its acknowledgement back from node j to node i; $\mathsf{TO}ij$ denotes delay of retrying either Req_{ij} or Req_{ji} after detecting contention between Req_{ij} and Req_{ji}.

The Algorithm 1 learns from traces depicted in Figure 2 and constructs the model in Figure 3 that describes interactions of environment (Node #1, Node #2 and Node #3) with SUT (Node #0). When browsing the events of each log Algorithm 1 simultaneously extends the automata involved in current interaction event.

6. CONCLUSION

We propose an algorithm of learning interaction models that are formalized as a subclass of Uppaal TA with synchronous communication assumption. The constraints and assumptions to the algorithm are due to the specifics of algorithm application. The learned model of interaction between a system and its environment is relevant for generating load tests of load critical HMS and its components. The method is exemplified with the learning of the IEEE1394 protocol model. Regardless the limited size of the example the learning method is scalable thanks to

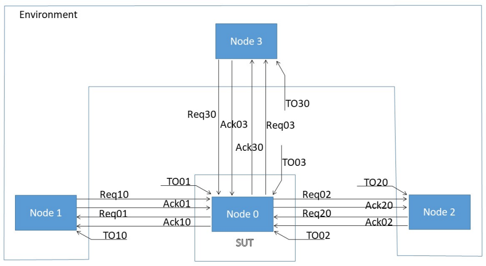


Fig. 1. IEEE1934 test case: data flow between SUT and Environment

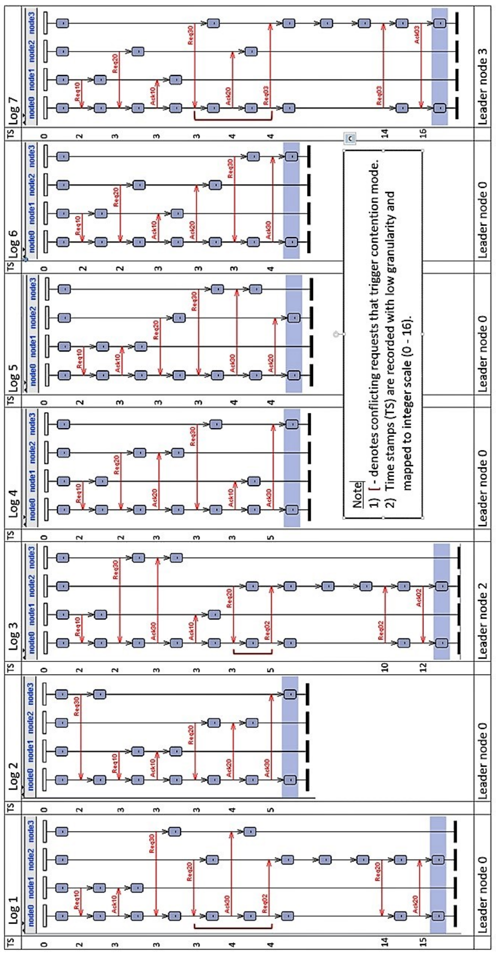

Log 7
Log 6
Log 5
Log 4
Log 3
Log 2
Log 1
TS
node0
node1
node2
node3
Req10
Req20
Req30
Ack10
Ack20
Ack30
Req03
Ack03
Req02
Ack02
Leader node 3
Leader node 0
Leader node 2
Note
1) [- denotes conflicting requests that trigger contention mode.
2) Time stamps (TS) are recorded with low granularity and mapped to integer scale (0 - 16).

Fig. 2. IEEE1934 network monitoring logs

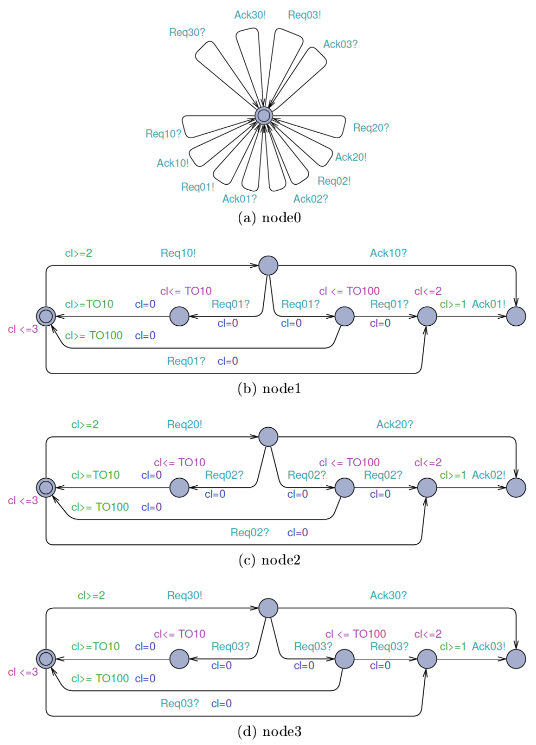


Fig. 3. Model of IEEE1394 leader election protocol constructed by learning

the interval abstraction mechanism built in the algorithm. The key features of the algorithm are following:

(i) learning is incremental, i.e., the model elements already learned cannot be removed during learning process, the equivalence classes of model elements can only be extended to certain bound and new ones created if the log analysis detects the observation instances not maching with already recoded classes;
(ii) the model learned can be applied for verifying the feasibility of tests generated from the learned model;
(iii) the learning algorithm can be driven by parameters such as feature vector, state space granularity, depth of control state history vector, domain scaling etc. This allows generating families of models with different level of state and time abstraction.

Proposed model learnign approach is approximative in the sense that it interpolates the border cases of equivalence classes encoded in time invariants of Uppaal TA model locations. Therefore, for method validation it has been checked that the traces used in the learning set and the traces generated by the learned Uppaal TA model are equivalent modulo timing equivalence defined by interval abstraction parameter R applied in the algorithm.

ACKNOWLEDGEMENTS

This research was supported by the Estonian Ministry of Education and Research institutional research grant IUT33-13.

REFERENCES

Abbors, F., Ahmad, T., Truscan, D., and Porres, I. (2013). Model-Based Performance Testing of Web Services Using Probabilistic Timed Automata. In K.H. Krempels and A. Stocker (eds.), *Proceedings of the 9th International Conference on Web Information Systems and Technologies*, 99–104. Webist.

Ade, M., GHRIET, P., Deshmukh, P., and SCOE&T, A. (2013). Methods for incremental learning: A survey. *International Journal of Data Mining & Knowledge Management Process*, 3(4), 119–125.

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and computation*, 75(2), 87–106.

Behrmann, G., David, A., and Larsen, K.G. (2004). A tutorial on UPPAAL. In M. Bernardo and F. Corradini (eds.), *SFM–RT 2004*, volume 3185 of *LNCS*, 200–237. Springer Verlag.

Brinksma, E. (1989). Formal approach to conformance testing. In *Proc. Int. Workshop on Protocol Test Systems*, 311–325. North-Holland.

Dias Neto, A.C., Subramanyan, R., Vieira, M., and Travassos, G.H. (2007). A survey on model-based testing approaches: a systematic review. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, 31–36. ACM.

Francez, N., Hoare, C., Lehmann, D.J., and De Roever, W.P. (1979). Semantics of nondeterminism, concurrency, and communication. *Journal of Computer and System Sciences*, 19(3), 290–308.

Freund, Y., Kearns, M., Ron, D., Rubinfeld, R., Schapire, R.E., and Sellie, L. (1993). Efficient Learning of Typical Finite Automata from Random Walks. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, 315–324. ACM, New York.

Grinchtein, O., Jonsson, B., and Leucker, M. (2004). Learning of event-recording automata. In Y. Lakhnech and S. Yovine (eds.), *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, 379–395. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kearns, M.J. and Vazirani, U.V. (1994). *An introduction to computational learning theory*. MIT press.

Morin, R. (2008). Semantics of deterministic shared-memory systems. In *CONCUR 2008-Concurrency Theory*, 36–51. Springer.

Shaw, J. (2000). Web application performance testing—a case study of an on-line learning application. *BT Technology Journal*, 18(2), 79–86.

Tappler, M., Aichernig, B.K., Larsen, K.G., and Lorber, F. (2018). Learning timed automata via genetic programming. *CoRR*, abs/1808.07744. URL `http://arxiv.org/abs/1808.07744`.

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

Appendix 5

Publication V
J. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ros packages: a mobile robot case study. In *2015 IEEE European Conference on Mobile Robots*, 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.

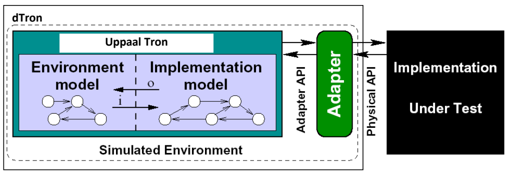


Fig. 1. The test setup involving the Uppaal Tron test engine and the distributed adapter library dTron.

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.

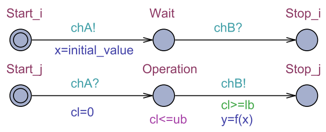


Fig. 2. A sample system with two Uppaal timed automata with synchronisation channels `chA` and `chB`. The automaton at the top denotes Process_i and the one below Process_j. In addition to the automata, the model also includes the declarations of channels `chA` and `chB`, integer constants `lb=1`, `ub=3`, 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].

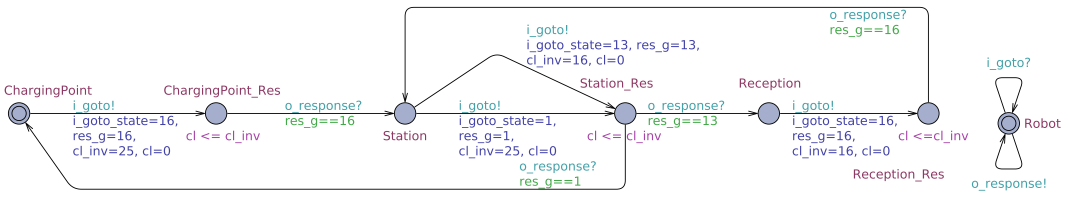


Fig. 3. Automatically generated timed automaton representation of the topological map containing locations “ChargingPoint”, “Station” and “Reception”.

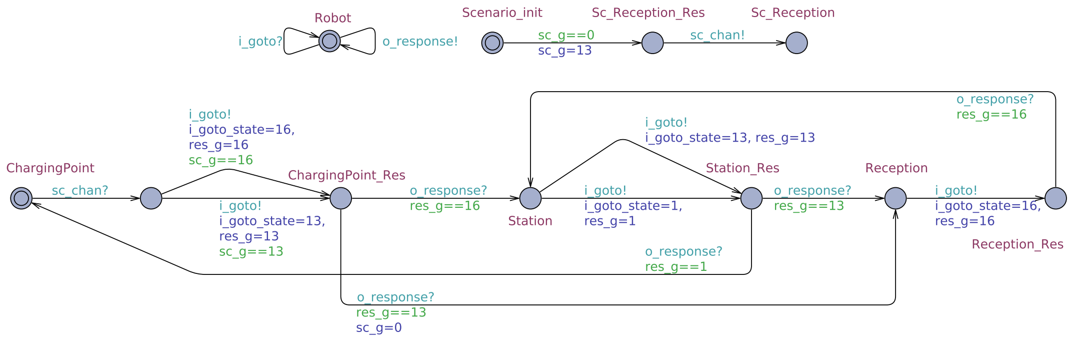


Fig. 4. Automatically generated timed automaton denoting the topological map (below) and the desired scenario (above).

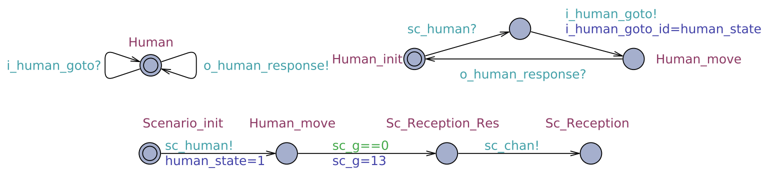


Fig. 5. A scenario involving models of humans

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 6

Publication VI
J. Vain, G. Kanter, and S. Srinivasan. Model based testing of distributed time critical systems. In *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 99–105. IEEE, 2017

Model Based Testing of Distributed Time Critical Systems

J¨uri Vain[1], Gert Kanter[2], Seshadhri Srinivasan[3]

[1]*Department of Software Science Tallinn University of Technology Tallinn, Estonia; juri.vain@ttu.ee*
[2]*Department of Software Science, Tallinn University of Technology, Tallinn, Estonia; gert.kanter@ttu.ee*
[3]*Berkeley Education Alliance for Research, Singapore; Seshadhri.srinivasan-bears@berkeley.sg*

Abstract — **Model-based testing incorporates steps such as test model construction, test purpose specification, test generation, deployment and execution. While the verification has not been traditionally an obligatory part of this process we extend the test development model by introducing model-based techniques, a tool and verification conditions of provably correct test development for time critical distributed systems. We demonstrate how Uppaal Timed Automata models and related tool family supports the development and verification of symbolic tests. Since distributed testing needs additional test deployment effort, we present the test controllability criteria for remote and distributed testing, provide an algorithm of distributing remote tests to improve the test performance and propose a technique of proving the correctness of distributed tests in terms of bisimulation equivalence between the remote and distributed tests.**

I. INTRODUCTION

In the process of developing complex networked systems such as Cyber-Physical Systems (CPS) the problems of inherent concurrency over wide spectrum of services and heterogeneous architectures need to be addressed. The heterogeneous components introduce functional, timing, safety, performance, and security features on multiple scales. In safety/mission/business-critical applications the networking of feature-rich components needs to be paired with predictability of system's emerging behaviour to guarantee required QoS. This is almost impossible to achieve without design validation methods that are scalable and relevant to holistic design views. While the features of functionality have gained major attention in traditional software development approaches, achieving the predictable timing of critical services in the presence of heterogeneous and evolving distributed architectures remains still a challenge. Therefore, validation methods like bench testing and encasing alone, although helpful and widely used, have become inadequate for full–fledged networked systems. As stated in [1] CPS software quality and software process productivity issues can be mitigated with model-based (MB) techniques and tools that operate on relevant level of abstraction. Model-based testing (MBT) as one group of these techniques provides the black-box testing solution for reducing software testing effort [2]. MBT suggests the use of abstract models for specifying the expected behaviour of the system under test (SUT) and automatically generating tests from models. According to Utting et al. [3] MBT incorporates steps such as SUT modelling, test purpose specification, test generation, test deployment and execution. Though MBT workflow relies inherently on the techniques of model engineering, the verification of the test development process and its products is not generally an obligatory part of MBT. On the other hand, the provably correct development (PCD) disciplines studied in [4], capitalize on the development process paired with verification and design correctness assurance steps. Applying PCD processes to testing is motivated by the need need to improve the trustability of testing results by showing their formal correctness through entire test development and execution process. In this paper we focus on the model-based online testing of distributed systems with timing constraints capitalizing on the correctness criteria and proving them through MBT workflow.

MBT is generally understood as conformance testing where the SUT is assumed to be a black-box where only its inputs and outputs are externally controllable and observable respectively. The internal behavior of the system is abstracted away. The aim of black-box conformance testing according to [3] is to check if the behaviour observable on the system interfaces conforms to the system requirements specification. During MBT a tester executes selected test cases (extracted from the system requirements model) by running SUT in the test harness and emits a test verdict (pass, fail, inconclusive). The verdict shows test result in the sense of conformance relation between SUT and the requirements model. A "classical" conformance relations is Input-Output Conformance (IOCO) introduced by Tretmans [5]. The behaviour of a IOCO-correct implementation should respect after some observations following restrictions:

(i) the outputs produced by SUT should be the same as allowed in the requirements model;

(ii) if a *quiescent state* (a situation where the system can not evolve without an input from the environment) is reached in SUT, this should also be the case in the model;

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

The set of tests that forms a *test suite* is divided 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 SUT and from the *coverage criteria*

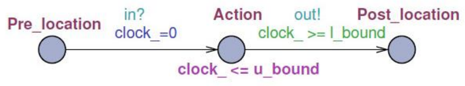


Fig. 1. Elementary modelling fragment *Action*

that constrain the behaviour with only those addressed by the test purpose. In this paper Uppaal Timed Automata (Uppaal TA) [6] are used to model SUT behaviour. This choice is motivated by the need to test the SUT with timing constraints so that the impact of communication delays between the SUT and the tester can be taken into account when the test cases are generated and executed against a distributed time critical SUT.

Another crucial aspect that needs to be addressed when testing networked systems is *non-determinism* of SUT with respect to test inputs. It includes both *functional* and *timing* non-determinism. In nondeterministic systems *online testing* (generating test stimuli *on-the-fly*) is applicable in contrast to that of deterministic systems where test sequences can be generated *offline*, i.e. before running the tests. For instance, the source of non-determinism is communication latency between the tester and the SUT that may lead to mixed interleaving of test inputs and outputs to/from remote ports. The phenomena that are related to communication delays and non-determinism faced in distributed testing have been outlined and $\triangle$-testability criterion for such systems introduced in [7]. $\triangle$ describes the communication latency that ensures that SUT input and output messages do not get mixed when the tester communicates with spatially distributed components of SUT.

In the rest of the paper we demonstrate how Uppaal TA models and Uppaal tool family provide support for provably correct test development, i.e. constructing SUT models, specifying coverage criteria using these models, extracting symbolic tests from them, and adding test deployment information for running test on networked non-deterministic systems with timing constraints. Finally, we present the test controllability criteria for remote and distributed testing, provide an abstract tester distribution algorithm and a proof technique to demonstrate the bisimulation equivalence (relative to test i/o actions) between the centralized tests and distributed tests derived from the centralised ones.

II. Constructing Test Models

A. *Modelling Timing Aspects of SUT*

Model-based testing of timing aspects presumes formal representation of quantitative time related notions attributed to events and actions of SUT. Uppaal TA provide representation of timing constraints using clock variables, and syncronizations expressed either by clock conditions or special synchronization labels *channels*. In [8] we have suggested a modelling pattern called *Action* (depicted in Figure 1) for constructing the test models using few basic model elements and composition rules.

An Action models a fragment of system behavior on a given level of abstraction as an *atomic* state transition. The Action in the SUT or Tester model component is triggered by its input event $in?$ and Action termination (expectedly within some bounded time interval called *response time*) generates an output event $out!$ of that component. The SUT input events (*stimuli* in the testing context) are generated by Tester, and the output events (SUT *responses*), if not *silent events*, are assumed to be observable to the Tester. Synchronous interactions between SUT and Tester components are represented in Uppaal TA model by *channels* that link input/output events occuring simultaneously in the parallel components of SUT and Tester. The test data communicated over test interfaces are modelled by using global data structures such as variables and arrays accessible by both SUT and Tester.

Main timing attribute of the$Action$ is its duration. Due to timing imperfections and abstraction it is often feasible to represent the duration in the SUT model as a closed interval $[l\_bound,\ u\_bound]$, where l_bound and u_bound denote lower and upper bound repectively. The duration interval $[l\_bound,\ u\_bound]$ is expressed in Uppaal TA as a pair of predicates on clocks as shown in Figure 1. The clock reset $clock\ =\ 0$ on the edge $Pre\_location\ \rightarrow\ Action$ makes the time constraint specification $local$ to the $Action$ and independent from accumulated durations of Actions executed earlier. The clock invariant $clock\_\ <=\ u\_bound$ of location $Action$ forces the $Action$ to terminate latest at time instant u_bound after the clock reset. The 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 control states: *passive* and *active*. Note that in Uppaal TA, the states defined as valuations of data variables are called *locations* and there are many clock states bounded by duration interval that corresponds to a location. So, in passive states SUT is waiting for the test stimuli and in its active state SUT can choose a move to the next state. Due to the TA semantics the transition to new location can be delayed as long as the active state invariant is true. The location can be left earliest when the guard of outgoing transition $Action\ \rightarrow\ Post\_location$ evaluates to $true$. For instance, in Figure 1, the locations $Pre\_location$ and $Post\_location$ are passive while $Action$ is an active location.

We construct the test model consisting of one or many SUT and Tester automata by composing instances of the Action pattern using *sequential*, *alternative* and *parallel* composition.

Definition 1 *Composition of Action patterns*

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

- sequential composition $F_i\,;F_j$ is Uppaal TA fragment where $l_i^{post} = l_j^{pre}$;

- alternative composition $F_i + F_j$ is Uppaal TA fragment where $l_i^{pre} = l_j^{pre}$ and $l_i^{post} = l_j^{post}$;
- synchronous parallel composition $F_i \| F_j$ is Uppaal TA fragment where F_i and F_j are in different automata so that their transitions $l_i^{pre} \rightarrow Action_i$ and $l_j^{pre} \rightarrow Action_j$ are sychronized, and also $Action_i \rightarrow l_i^{post}$ and $Action_j \rightarrow l_j^{post}$ are synchronized, .

We define well-formedness (*wf*) of test models according to the following inductive definition.

Definition 2 *Well-formedness*
- atomic Action pattern is well-formed;
- sequential composition of well-formed models is well-formed;
- alternative composition of well-formed models is well-formed if the output labels are distinguishable.
- parallel composition of well-formed models is well-formed provided their execution intervals are consistent, i.e. $[l\_bound_i, u\_bound_i] \bigcap [\_bound_j, u\_bound_j] \neq \emptyset$.

As shown in [9] large class of Uppaal TA modes of practical value can be transformed to bisimilar (w.r.t. i/o actions at test interfaces) well-formed representation.

In the rest of the paper we assume generally that M^{SUT} is well-formed and denote it by $wf(M^{SUT})$. An example of the well-formed model is depicted in Figure 2 left.

B. Correctness of Test Models

Model based test generation algorithms presume that SUT models have properties which guaratee the feasibility of generated tests. For instance, among these properties are *connectedness*, *input complete*ness, *output observability* and *strong responsiveness*. In this section we demonstrate how these properties formulated originally for IOTS (Input-Output Transition System) models [10] can be verified also on well-formed Uppaal TA models.

a) Connected Control Structure and Output Observability: Uppaal TA model is *connected* if there exists an executable path from any location to any other location. Since SUT represents an open system that is interacting with its environment we need for verification of the connectedness by model checking a *nonrestrictive environment* model which provides test stimuli and receives test responses in any possible order the SUT model can interact with its environment. Such an environment model is *canonical tester* proposed in [10]. A canonical tester can be created for well-formed SUT model in Figure 2 left using the pattern shown in Figure 2 right.

The canonical tester composed in parallel with SUT model implements *random walk* test strategy that is useful in endurance testing. On the other hand, it may be very inefficient when the testing purpose is to cover few selected elements of SUT model. Having synchronous parallel composition of SUT and the canonical tester (shown in Figure 2) the connectedness of SUT can be model checked with query (1) expressed in Computation Tree Logic CTL. Query (1) expresses the absence of deadlocks in interaction between SUT and the canonical tester.

$$A[]\, not\, deadlock \tag{1}$$

The *output observability* means that all state transitions of the SUT model are observable and identifiable by its outputs. By Definition 2 the output observability of SUT models is a co-product of well-formedness because each input event and Action location has an outgoing edge that generates a locally (w.r.t. the Action input event) unique output event.

b) Input Enabledness: Input enabledness of SUT models means that no blocking of SUT due to irrelevant test inputs can occur. Straightforward way of transforming a SUT model (if the number of inputs is bounded) to input enabled automaton by introducing self-looping transitions with input labels that are not represented on other transitions that depart from the same location. That makes SUT modelling tedious and causes an exponential increase of its size. 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 encoding input symbols as boolean variables $I_1 \ldots I_n \in \Sigma^{in}$. Then the pre-locations l_i^{pre} of actions A_i (see Figure 2) need 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 $l_i^{pre} \rightarrow l_i^{pre}$ that models the acceptance of all inputs in $\Sigma^{in}(M^{SUT})$ except I_j, $j = 1, k$ and specify the guard of this auxiliary edge with boolean expression: $not(\underset{j=1,k}{\vee} I_j)$.

Provided the branching factor $\mathcal{B}$ of the edges that are outgoing from l_i^{pre} is, as a rule, substantially smaller than the size of input alphabet $|\Sigma^{in}(M^{SUT})|$, we can save compared to straightforward algorithm $|\Sigma^{in}(M^{SUT})| - \mathcal{B}(l_i^{pre})$ edges at each pre-location of the Action pattern. 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 3 (time constraints are ignored here for clarity). Symbols $I1$ and $I2$ in the figure denote predicates $Input == i1$ and $Input == i2$ respectively.

c) Strong Responsiveness: Strong responsiveness (SR) means that SUT model always reaches a *quiescent state* (location) after executing finite number of transitions. In other words, there is not reachable *livelocks* (a loop that includes only ε-transitions) in the SUT model. 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 default. Strong responsiveness is verified by Algorithm 1.

Algorithm 1 *Checking strong responsiveness*

1) By input enabled Action pattern in Figure 3 the input events of M^{SUT} are encoded by using channel *in?* and

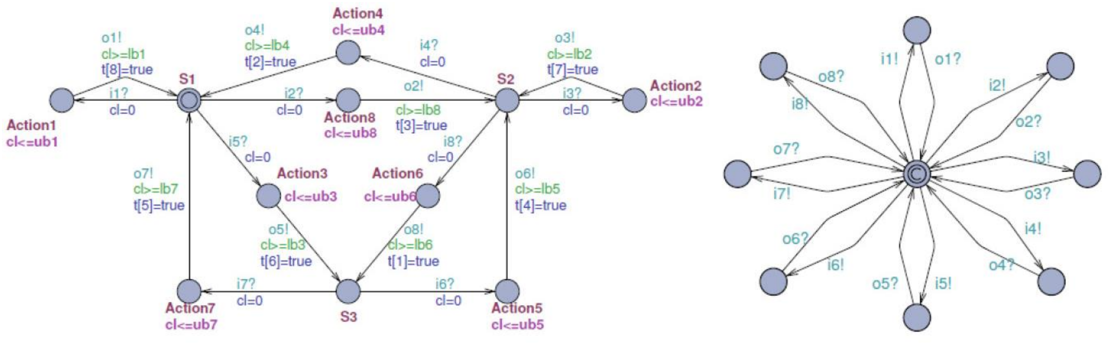


Fig. 2. Synchronous parallel composition of SUT and its canonical tester models

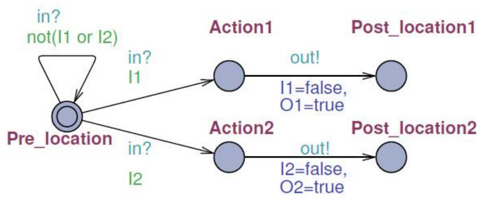


Fig. 3. Input-enabled model fragment

a boolean variable I_i that represents the condition that the input value is i_i. Since input occurence in Uppaal TA can be expressed now as state property, we have to keep the input value indicating predicate $true$ in the destination location of the edge that is labeled with given event and reset to $false$ immediately when leaving this location. For same reason the ε-transitions need to be labeled with update $EPS = true$ and following output edge with update $EPS = false$.

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 initial location of M^{SUT}. The query checks if any ε-transition is reachable from l_0 (this is necessary condition for violating SR-property).
3) Remove all non ε-transitions and locations that remain isolated after step 2.
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.

Note that all steps except step 2 of Algorithm 1 are of linear complexity in the size of the M^{SUT} .

III. Correctness of Tests

A. Structural coverage of Tests

The purpose of MBT is to detect the violation of conformance relations between the model and SUT. Therefore, tests are expected to cover possibly many elements and behaviors of the SUT model within the time and computation resources allocated for testing. The structural coverage criteria are defined in terms of structural elements (states, transitions, conditions) of the SUT model. Rushby et al have suggested in [13] to express the structural coverage of state machine models by means of *trap*-variables. To specify the coverage items in the model the expressions assigning boolean value *true* to trap-variables are added to the edges of the SUT model. When using traps as coverage items we declare the test being coverage correct if the test chooses SUT inputs so that all traps are *true* when test terminates.

Definition 3 *(Trap) coverage correctness of the test*
We say that a test is coverage correct (with respect to the specified set of traps) if the test run covers all the transitions that are labelled with traps in the SUT model.

Definition 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 in terms of model checking queries and model building constraints.

Direct way of verifying the coverage correctness of the test in Uppaal tool is to model check:

$$M^{SUT} \| M^{Test} \models A\Diamond\forall(i : int[1, n])t[i]) \qquad (2)$$

where $t[i]$ denotes i-th element of the array t of traps. The test model of query (2) is the synchronous parallel composition of SUT and test automata. An example of this composition is depicted in Figure 4.

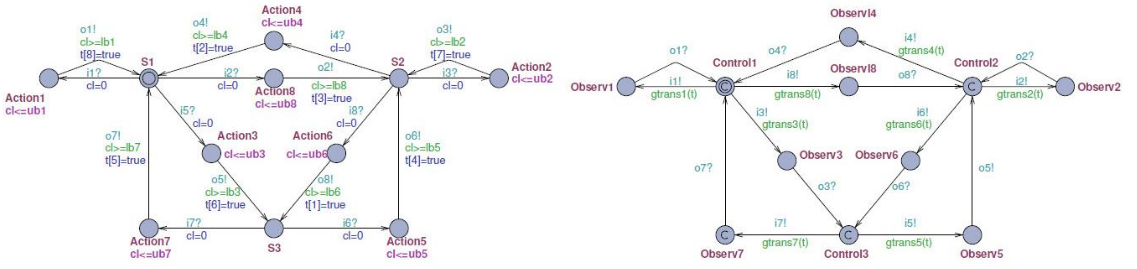


Fig. 4. Synchronous parallel composition of SUT and test automata

B. Boundedness of tests

Testing time depends on the length and duration of executing the test scenarios satisfying given coverage criteria.

Definition 5 *Length-wise boundedness of the test*

A test is length-wise k-bounded (consisting of k stimuli/responses at most) if there exists an upper bound k to all test runs of the test model $M^{SUT}||M^{Test}$.

When proving the length-wise boundedness of tests using Uppaal model checker the properties of well-formed SUT models simplify the task considerably. Having a $wf(M^{SUT})$ we set the lower and upper bounds lb_i and ub_i of its actions to 1, i.e., $lb_i = ub_i = 1$ for all $i \in 1, \dots |Action|$. Then the length of the test sequence and its duration are numerically equal. When checking an integer bound k of test sequences the following model checking query proves the coverage of n traps with at most k stimuli and responses

$$wf(M^{SUT}||M^{Test}) \models A\Diamond \forall(i : int[1,n])t[i]) \ \wedge TM \leq k, \tag{3}$$

where TM is a clock variable introduced to represent the progress of global time in the model. Note that in the test model M^T we assume the duration of input selection actions is negligible compared to that of durations of M^{SUT} actions. It means *Action* locations in M^{Test} are of type "committed" (instantaneous).

Generalizing this approach for SUT models with arbitrary time constraints we can assume that all edges of $wf(M^{SUT})$ are attributed with time constraints satisfying conditions of well-formedness described in Section 2.1.

Definition 6 *Time-wise boundedness of the test*

A test is time-wise bounded if there exists an upper bound *TH* to the duration of all test runs of the test model $M^{SUT}||M^{Test}$.

Verification of time-wise boundedness requires model checking of (4):

$$wf(M^{SUT}||M^{Test}) \models A\Diamond \forall(i : int[1,n])t[i]) \ \wedge TM \leq TH \tag{4}$$

Since not all of M^{SUT} edges need to be labeled with traps (and not covered by test) we apply compaction procedure to M^{SUT} to abstract away from the excess of information (for IOCO testing) in M^{SUT}. With the compaction procedure we merge the sequences of trapless edges with the trap-labelled edge the trapless ones are preciding. As the result, the sequence of trapless actions is merged with the trap labeled action making an agregate. The first constituent edge of the aggregate contributes its input event and the last edge its output event. The other i/o events of the aggregate will be hidden from the symbolic model and will be implemented in the test adapter described in Section IV. Theb the lower and upper bounds for the composite action are computed: 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 SUT model it can be proved that the duration TH of coverage correct tests have length satisfying following bound condition:

$$\sum_i lb_i \leq TH \leq \sum_i ub_i + \max_i(ub_i), \ \ i = 1, n \tag{5}$$

where n is the number of traps in M^{SUT}. In case of non-deterministic SUT models, for showing the length- and time-wise poundedness of generated tests we need an assumption of fairness of M^{SUT}. A model M is *k-fair* iff the difference in the number of executions of alternative transitions of non-deterministic choices (sharing same departure location) never exceeds the bound k. During the test run the test execution environment DTRON [9] is capable 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 $\sum_i lb_i$.

IV. Correctness of test deployment

A. *Test adapters*

The execution of symbolic tests presumes test adapters that map symbolic i/o alphabet used in the test model $M^{SUT} \| M^{Test}$ to SUT executable inputs. Similarly, real outputs from SUT need to be transformed back to symbolic outputs to be compared for conformance checking. Both mappings performed by test adapters may introduce additional delays that are not reflected neither in SUT nor abstract test models. Also, distributed test configurations may introduce delays due to signal propagation time that can alter ordering of test stimuli and responses specified in the model. Network monitors, e.g. the one included in DTRON tool [9] 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 suite, the latency estimates need to be incorporated also in the test 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 for remote testing of distributed SUT is $\Delta - testability$. The concept proposed in [7] introduces parameter Δ which defines minimum delay between consecutive test stimuli necessary to maintain the ordering of input-output events at remote SUT ports. When verifying the correctness of test deployment with test adapters as well as for remote testing 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 $wf(M^{SUT+\triangle})$.

The procedure of constructing the test adapters in testing framework DTRON are described in detail in [9].

B. *Distributed test deployment*

In remote testing sending test inputs and waiting for outputs is assumed to take not less than 2Δ. This is time needed for signal travelling from tester to SUT ports and back to tester. Thus, when testing non-determinsitic systems, for test to be on-line controllable we face 2Δ delay barrier. The consequence is that for realtime systems that are expected to react to new inputs sooner than 2Δ after providing some output, the remote testing is not applicable. A remote test is 2Δ- controllable if in the presence of network (incl. test harness) delays less than Δ wrong interleavings of inputs and outputs are excluded.

The performance and reaction time restrictions caused by 2Δ delay in remote testing can be mitigated by decomposing the central remote test to multiple local tests which are attached to the SUT test ports directly. Instead of bidirectional communication between a remote test and the SUT, only unidirectional synchronization between the local tests is required. In the approach of [11] the local tests are generated in two steps: at first, a centralized remote test is generated by applying the reactive planning online-tester synthesis method of [12], and second, a set of synchronizing local tests is derived by decomposing the test into a set of location specific test instances. The decomposition preserves the test correctness so that if the original remote test is 2Δ- controllable then the distributed test is Δ-controllable and bisimilar to original with respect to the test interface i/o actions.

Algorithm 2 *Test distribution*

Let M^{CT} be a model of a centralized remote tester, $Loc(SUT) = l^i,\ i \in [1, n]$ a set of n spatially separated port locations of SUT, and P^i a set of SUT test ports accessible in the location l^i.

Step 1: Copy an instance M^{LT}_i of the centralized remote tester M^{CT} for each $l^i \in Loc(SUT)$.

Step 2: Parsing the edges of the$M^{LT}_i, i \in [1, n]$:

Case 1: **If** the edge of M^{LT}_i is labeled with an i/o action a^i_k interacting with any local port p of location $l^i \in Loc(SUT)$, **then** refine the edge with a^i_k so that the i/o event is broadcasted also to the instances a^j_k of same action a_k in other locations $l^j \in Loc(SUT),\ i \neq j$.

Case 2: **If** the edge of M^{LT}_i is labeled with an output action a^i_k interacting with the port p of location $l^j \in Loc(SUT)$, where $i \neq j$, **then** substitute the output action a^i_k with an input action $a^i_{k'}$ and synchronize it with the instance a^j_k in M^{LT}_j via broadcast channel (created in *Step* 2 *Case* 1).

C. *Correctness of Test Distribution*

To verify the correctness of distributed test deployment we check the (relative with respect to test i/o actions) bisimulation equivalence between the model $M^C = M^{CT} \| M^{SUT}$ of the centralized remote test and the model $M^D = M^{DT} \| M^{SUT}$ where M^{DT} is derived from M^{CT} by distributing it to n local tests $M^{DT} = \|_i M^{LT}_i, i \in [1, n]$. For that the models M^C and M^D are composed by synchronous parallel compositions so that one has a role of words generator on the test i/o alphabet and other the role of words acceptor. If the i/o language acceptance is established in one direction then the roles of models are inverted. Second adjustment of models to be made for bisimulation analysis is the reduction of message propagation delays in the models to uniform basis. Without loss of generality we can assume it is 0 in both models.

The mapping $M^{CT} \xmapsto{Algorithm2} M^{DT}$ is correct if M^{CT} and M^{DT} are observation bisimilar, i.e. if M^C and M^D are resp. generating and accepting Uppaal TA on common test i/o alphabet $\Sigma^i \cup \Sigma^o$ then all timed words $TW(M^C)$ generated by M^C are recognizable by M^D and all timed words $TW(M^D)$ generated by M^D are recognizable by $TW(M^C)$.

D. *Verification:*

Step 1: Constructing synchronous parallel composition of the generating and accepting automata:

Split all the edges e of M^C that carry the label of a test i/o action into two edges e' and e'' connected via committed location, where e' copies the labeling of e and e'' is labeled with a unique auxiliary output action (channel) name $Ch!$. Do the same splitting with the corresponding edges e of M^D with the difference that instead of auxiliary actions $Ch!$ the edges e'' are labeled with their co-actions $Ch?$. This ensures that when ever a test i/o action is enabled in M^C the same action must be enabled also in M^D provided models are composed in parallel with auxiliary syncronization conditions, denoted $M^C\|_{aux}M^D$. If this is not the case then the deadlock in $M^C\|_{aux}M^D$ is reachable. Note that due to the semantics of Uppaal TA the synchronization via channels is symmetric and it suffices checking the deadlocks without inverting the direction of auxiliary syncronization channel between M^C and M^D.

Step 2: Bisimilarity proof by model checking

The composition of bisimilar testers must be non-blocking if the testers composed with SUT model separately are non-blocking, i.e. if the following holds:

$M^C \models not\ deadlock \bigwedge M^D \models not\ deadloc \implies M^C\|_{aux}M^D \models not\ deadlock.$

Conclusion and Discussion

This work has been motivated by the need to increase the trust on testing results paired with the predictability of systems emerging behaviour to guarantee multi-critical QoS in complex systems. We have focused on the formal correctness aspects of model-based online testing of distributed systems with timing constraints. The approach demonstrated is capitalizing on the correctness criteria and model checking based verification technique that covers main steps of MBT workflow. Due to the space limit the scalability and complexity issues have been addressed only partially and the authors see further opportunities in integration of these techniques to improve the scalability of test verification. As an extension of our current work along this line we see possible improvements in pairing our provably correct test development approach with contract based design methodology, aspect-oriented modelling, compositional verification, and model abstraction techniques.

References

[1] E. Lee, “Cyber physical systems: Design challenges,” in *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing* (ISORC), 2008, pp. 363–369.

[2] R. V. Binder, “2011 model-based testing user survey: Results and analysis”, online at http://robertvbinder.com/wp-content/uploads/rvbpdf/arts/MBT-User-Survey.pdf, Jan. 2012, last Accessed 11.5.2015.

[3] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” Software Testing, Verification and Reliability, vol. 22, no. 5, pp. 297–312, 2012, online; last accessed : 18.05.2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[4] R. Jeords, C. Heitmeyer, M. Archer, and E. Leonard, “A formal method for developing provably correct fault-tolerant systems using partial refinement and composition,”, in FM2009, Lecture Notes in Computer Science 6664, Springer 2011, pp. 173–189.

[5] J. Tretmans, “Testing concurrent systems: A formal approach,” in CONCUR99 Concurrency Theory, Lecture Notes in Computer Science 1664, Springer Berlin Heidelberg, 1999, pp. 46–65.

[6] G. Behrmann, A. David, K. A. Larsen, “A tutorial on uppaal,” in Formal Methods for the Design of Real-Time Systems. Springer, Berlin Heidelberg, 2004, pp. 200–236.

[7] A. David, K. G. Larsen, M. Mikucionis, O. L. N. Timo, A. Rollet, “Remote Testing of Timed Specifications,” Lecture Note in Computer Science 8254, Springer, 2013, pp. 65–81.

[8] J. Vain, A. Anier, E. Halling, “Provably correct test development for timed systems,” Frontiers in Artificial Intelligence and Applications 270, IOS Press, Amsterdam, 2014, pp. 289–302.

[9] A. Anier, “Model based framework for distributed control and testing of cyber-physical systems,” Ph.D. dissertation, Dept. Comp Sci., Tallinn Univ. of Techn., Estonia, 2016.

[10] J. Tretmans, “Test Generation with Inputs, Outputs, and Quiescence”, Lecture Notes in Computer Science 1055, Springer, 1996, pp. 127–146.

[11] J. Vain, E. Halling, G. Kanter, A. Anier, and D. Pal, “Automatic Distribution of Local Testers for Testing Distributed Systems,”, Artificial Intelligence and Applications 291, IOS Press, Amsterdam, 2016, pp. 297–310.

[12] J. Vain, M. Kääramees, and M. Markvardt, “Online testing of nondeterministic systems with reactive planning tester,” in Dependability and Computer Engineering: Concepts for Software-Intensive Systems, IGI Global, Hershey 2012, pp. 113–150.

[13] G. Hamon, L. De Moura, and J. Rushby, “Automated test generation with sal”. CSL Technical Note (2005)

Curriculum Vitae

1. Personal data

Name	Gert Kanter
Date and place of birth	17 September 1983 Tallinn, Estonia
Nationality	Estonian

2. Contact information

Address	Tallinn University of Technology, School of Information Technology, Department of Software Science, Ehitajate tee 5, 19086 Tallinn, Estonia
Phone	+372 620 2325
E-mail	gert.kanter@taltech.ee

3. Education

2012–. . .	Tallinn University of Technology, School of Information Technology, Computer Science, PhD studies
2009–2012	Tallinn University of Technology and University of Tartu, Faculty of Information Technology, Software Engineering, MSc
2003–2008	Tallinn University of Technology, Faculty of Information Technology, Telecommunication, BSc

4. Language competence

Estonian	native
English	fluent
Finnish	proficient
Russian	basic skills

5. Professional employment

2015– . . .	Tallinn University of Technology, Lecturer in Computer Science
2012–2015	ELIKO Technology Competence Centre in Electronics-, Info- and Communication Technologies, Researcher
2003–2011	Hardmeier A&D OÜ, Software Engineer

6. Honours and awards

- 2019, Dependable Systems, Services and Technologies (DESSERT 2019) conference best paper award

7. Defended theses

- 2012, Robot Manipulator Control Using Stereo Visual Feedback, MSc, supervisor Prof. Jüri Vain, Tallinn University of Technology, Department of Computer Science

8. Field of research

- Model-based testing
- Autonomous systems testing
- Software testing

9. Scientific work
Papers

1. G. Kanter and J. Vain. Model-based testing of autonomous robots using TestIt. *Journal of Reliable Intelligent Environments*, 6(1):1–17, 2020
2. G. Kanter and J. Vain. Testit: an open-source scalable long-term autonomy testing toolkit for ros. In *Proceedings of the 10th International Conference Dependable Systems, Services and Technologies, DESSERT'2019*, pages 45–50, 2019
3. G. Kanter, J. Vain, S. Srinivasan, and S. Ramaswamy. Provably correct configuration management of precision feeding in agriculture4.0. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 1631–1637, 2019
4. J. Vain, G. Kanter, and A. Anier. Learning timed automata from interaction traces. In *14th IFAC Symposium on Analysis, Design, and Evaluation of Human Machine Systems, HMS 2019*, volume 52-19, pages 205–210, 2019
5. J. Ernits, E. Halling, G. Kanter, and J. Vain. Model-based integration testing of ros packages: a mobile robot case study. In *2015 IEEE European Conference on Mobile Robots*, pages 1–7. IEEE, 2015
6. J. Vain, G. Kanter, and S. Srinivasan. Model based testing of distributed time critical systems. In *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)*, pages 99–105. IEEE, 2017

Elulookirjeldus

1. Isikuandmed

Nimi	Gert Kanter
Sünniaeg ja -koht	17.09.1983, Tallinn, Eesti
Kodakondsus	Eesti

2. Kontaktandmed

Aadress	Tallinna Tehnikaülikool, Tarkvarateaduse Instituut, Ehitajate tee 5, 19086 Tallinn, Estonia
Telefon	+372 620 2325
E-post	gert.kanter@taltech.ee

3. Haridus

2012–. . .	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Info- ja kommunikatsioonitehnoloogia, doktoriõpe
2009–2012	Tallinna Tehnikaülikool ja Tartu Ülikool, Infotehnoloogia teaduskond, Tarkvaratehnika, MSc
2003–2008	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Telekommunikatsioon, BSc

4. Keelteoskus

eesti keel	emakeel
inglise keel	kõrgtase
soome keel	kesktase
vene keel	algtase

5. Teenistuskäik

2015– . . .	Tallinna Tehnikaülikool, Arvutiteaduse lektor
2012–2015	ELIKO Tehnoloogia Arenduskeskus OÜ, Teadur
2003–2011	Hardmeier A&D OÜ, Tarkvaraarendaja

6. Autasud

- 2019, Dependable Systems, Services and Technologies (DESSERT 2019) konverentsi parima teadusartikli auhind

7. Kaitstud lõputööd

- 2012, Visuaalsel tagasisidel põhinev robotmanipulaatori juhtimine, MSc, juhendaja Prof. Jüri Vain, Tallinna Tehnikaülikool, Arvutiteaduse Instituut

8. Teadustöö põhisuunad

- mudelipõhine testimine
- autonomoosete süsteemide testimine
- tarkvara testimine

9. Teadustegevus
Teadusartiklite, konverentsiteeside ja konverentsiettekannete loetelu on toodud ingliskeelse elulookirjelduse juures.