

TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Department of Software Science

Artur Gummel 163303IAPM

**MODEL-BASED TESTING WITH TESTIT:
THE ROBOT OPERATING SYSTEM CASE-
STUDY**

Master's thesis

Supervisor: Jüri Vain

PhD

Co-supervisor: Gert Kanter

PhD student

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Artur Gummel 163303IAPM

**MUDELI-PÕHINE TESTIMINE
KESKKONNAGA TESTIT: ROBOTITE
OPERATSIOONISÜSTEEMI
JUHTUMIUURING**

magistritöö

Juhendaja: Jüri Vain

Doktorikraad

Kaasjuhendaja: Gert Kanter

Doktorant

Tallinn 2018

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Artur Gummel

07.05.2018

Abstract

The aim of this thesis is to confirm the possibility of adaptation of model-based testing to scalable long-term autonomy testing of ROS-based robot software.

This thesis includes the theoretical foundations of model-based testing, its usage in robotics, and the tools – UPPAAL, UPPAAL TRON, DTRON used for testing real-time systems. As for practical results of this thesis, the usability of test development toolchain has been demonstrated together with the model-based testing workbench TestIt for its application in robotics. The feasibility of studied approach has been proven by implementing the full workflow from test model specification till test suite execution. The implemented test cases achieved their goal by navigating an autonomous platform simulation in the confined area.

The thesis is in English and contains 40 pages of text, 4 chapters, 20 figures, 1 table.

Annotatsioon

Mudeli-põhine testimine keskkonnaga TestIt: robotite operatsioonisüsteemi juhtumiuuring

Käesoleva magistritöö eesmärgiks on mudelipõhise testimise kohandamine ROS-põhise robotitarkavara pikaajalise autonoomia testimiseks.

Töö sisaldab mudelipõhist testimise teoreetilisi aluseid, mudelipõhise testimise kasutamist robotikas ja testimise automatiseerimise vahendite UPPAAL, UPPAAL TRON, DTRON rakendusvõimaluste uuringut. Selle töö praktilise tulemusena demonstreeritakse robotite testimiskeskonna TestIt kooskasutuse võimalusi ja otstarbekust UPPAALi tööriistade perega ja seda just autonoomse navigatsiooni tarkvara testimisel. Lähenemise otstarbekuse näitamiseks on implementeeritud testide arendusprotsess testinõuete ja mudeli spetsifitseerimisest testide täitmiseni. Töös demonstreeriti, et realiseeritud testid, mille käigus toimub autonoomse platvormi navigeerimise piiratud alal, saavutavad oma eesmärgi.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 40 leheküljel, 4 peatükki, 20 joonist, 1 tabeli.

List of abbreviations and terms

Distributed system	A distributed system is a set of autonomous computers that are interconnected via a computer network and are equipped with the software needed to create an integrated environment [1].
Determinism	Determinism is a system property in which the output of the system and its subsequent state are uniquely determined by this state and input [2].
Non-Determinism	Non-Determinism is a system property in which the output of the system and its subsequent state are not uniquely determined by this state and input [2].
Conformance testing	Type of testing which purpose is to verify that the system complies with the specified requirements [3].
MBT	Model-Based Testing.
Adapter	Helps to convert symbolic inputs of the model in MBT to the format executable by SUT and SUT outputs back to symbolic form.
Real-time system	Type of hardware or software that works under time constraints.
Mission Critical system	Systems whose failure might cause catastrophic consequences: someone dying, damage to property, severe financial losses.
SUT	System Under Test.
UPPAAL	An integrated tool environment for modelling, simulation, and verification of real-time systems developed by Uppsala and Aalborg Universities [6].
UPPAAL TRON	Testing tool suited for black-box conformance testing of timed systems [7].
UPPAAL Model	Model of a real-time system represented as a network of extended timed automata.
UPTA	Uppaal Timed Automata.
DTRON	Distributed Testing Real-time systems Online is a command-line application based on UPPAAL TRON and intended for model-based testing of distributed systems [8].
ROS	Robot Operating System [9].
TestIt	A Scalable Long-Term Autonomy Testing Toolkit for ROS [11].

Table of contents

1 Introduction	11
1.1 Motivation	11
1.2 Related work.....	14
1.3 Thesis problem statement and main assumptions.....	15
1.4 Thesis structure.....	16
2 Preliminaries.....	17
2.1 Model-based testing.....	17
2.2 Uppaal Timed Automata	18
2.2.1 The Uppaal modelling language.....	20
2.2.2 Uppaal simulator	23
2.2.3 The Uppaal verifier.....	24
2.3 Uppaal TRON.....	26
2.4 DTRON	26
2.5 Behaviour trees	27
2.6 MBT for ROS-based robotics.....	29
3 Adaptation of the MBT workbench TestIt	31
3.1 TestIt architecture and design principles	31
3.2 Integrating DTRON with TestIt	32
3.2.1 Protocol buffers installation	32
3.2.2 Spread toolkit installation.....	33
3.2.3 Adapter	34
3.2.4 Dockerfile	38
3.3 Test generation for DTRON.....	38
3.3.1 Mapping topological maps to Uppaal TA	39
4 Case study: Autonomous platform navigating in the confined area.....	44
4.1 General description and test goals	44
4.2 Generating Uppaal TA models	45
4.3 Generating tests	45
4.4 Executing tests.....	46

4.5 Testing results.....	48
5 Summary.....	49
References	51
Appendix 1 – Example of Uppaal model	54
Appendix 2 – Dockerfile of the base image	56
Appendix 3 – Dockerfile of SUT	58
Appendix 4 – Dockerfile of testing base image	59
Appendix 5 – Oracle Test1	60
Appendix 6 – Oracle Test2.....	62
Appendix 7 – Oracle Test3.....	63
Appendix 8 – Configuration file.....	64
Appendix 9 – Example of passed Test1	67

List of figures

Figure 1. Model-based testing process [36].....	12
Figure 2. Example of the Model-based testing process.....	17
Figure 3. Example of the automaton with locations and edge.....	20
Figure 4. Example of synchronisation expression.....	21
Figure 5. Example of initial location.	21
Figure 6. Example of urgent location.	22
Figure 7. Example of committed location.	22
Figure 8. Example of invariant expression.....	22
Figure 9. Example of guard expression.	22
Figure 10. Example of update expression.	23
Figure 11. Example of selection expression.....	23
Figure 12. Example of reachability property.....	25
Figure 13. Example of safety properties.....	25
Figure 14. Example of liveness properties.	26
Figure 15. Example of DTRON configuration.....	27
Figure 16. Relationships between DTRON, TRON AND SUT.....	30
Figure 17. TestIt architecture.....	32
Figure 18. Example of willow garage map created by the author of this thesis (a) and taken from turtlebot gazebo package (b).	41
Figure 19. Example of Uppaal model processes.	43
Figure 20. Example of node map.	45

List of tables

Table 1. The node types of the BT [31].....	29
---	----

1 Introduction

1.1 Motivation

Due to a high level of integration of heterogeneous components in autonomous robotics reaching the sufficient level of quality presumes extensive testing of functional as well as performance aspects of robot software. That is not possible without systematic methodology supported by test automation tools.

The goal of using test automation tools is to simplify the preparation of tests as well as executing the tests automatically and tracing the root causes of detected bugs.

According to different sources the testing and verification may take up to 50% of development resources. In automotive and medical domain the system integration level test and verification cause project delays respectively in 63% and in 66.7% of cases [34]. Under these considerations, any increase in productivity of testing methods and tools has a strong impact on the productivity of the whole development process and on autonomous systems software assurance in general.

The highest level of test automation has been achieved by means of *model-based testing*.

Model-based testing (MBT) is considered to be generally black-box testing. MBT is divided into two types *offline* and *online* depending on whether tests suites are generated before or during the test execution. Both types consist of such process steps as a system and testing requirements specification, SUT modelling, test generation, test execution and analysing results (Figure 1). MBT is typically a part of model-based development techniques that provides the opportunities for test automation and reduces systems validation and verification effort [35]. MBT suggests the use of a formal model for specifying the expected behaviour of System Under Test (SUT) and the test purpose. For instance, the behaviours or model elements to be covered by the test are subject to test purpose specification. Both, the SUT model and test purpose specification are prerequisites for automatic test generation.

One standard use of MBT is conformance testing but it can be applied also in other types of testing and monitoring such as mutation testing, runtime monitoring, etc.

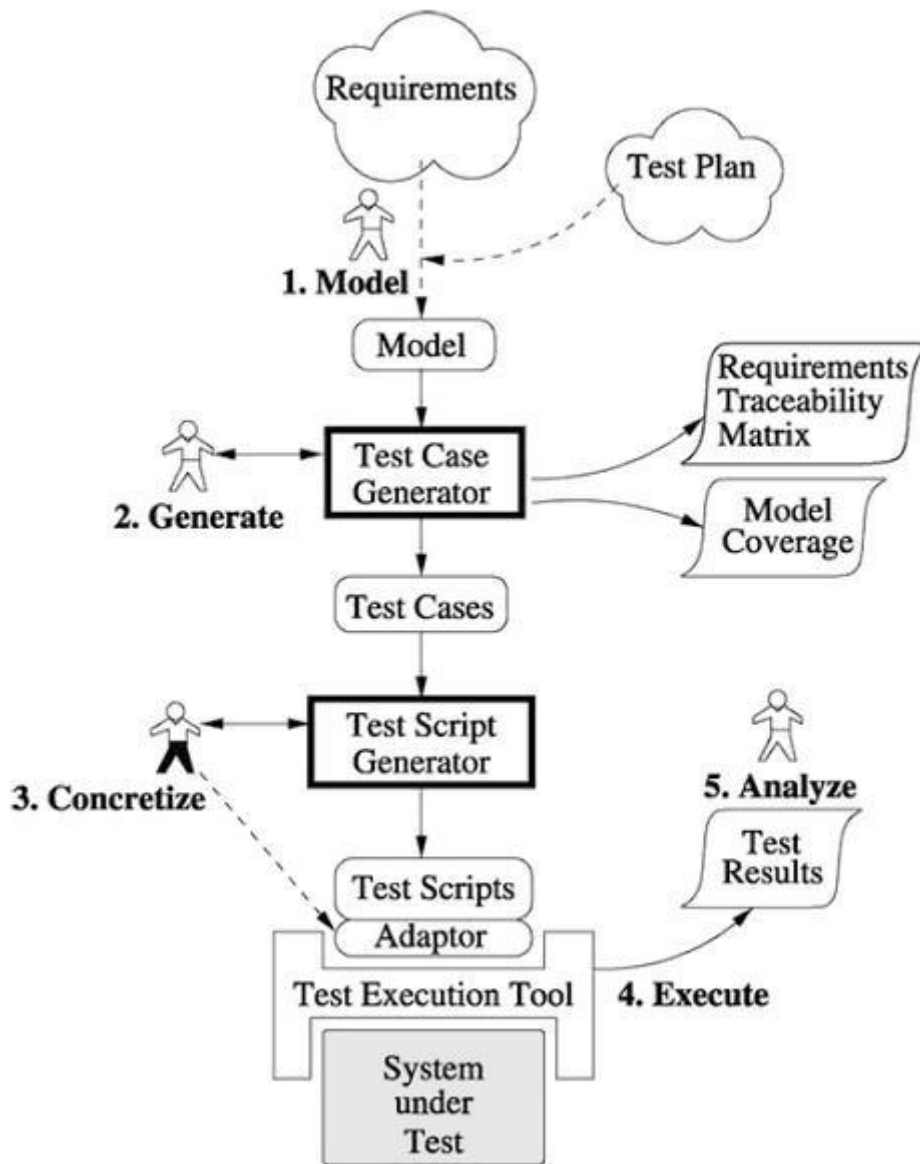


Figure 1. Model-based testing process [36].

Model-based testing has own advantages and disadvantages. The main goal of MBT is to check if real-time systems conform with requirements specification.

As advantages can be distinguished:

- model hides irrelevant details of implementation;
- automatic generation and execution of tests;

- easier test suite maintenance;
- systematic coverage of requirements;
- human errors are eliminated;
- relevant for regression testing where SUT model updates are much easier to do than rewriting test scripts.

As for disadvantages, following can be outlined:

- modelling is not the simplest part;
- modelling overhead needs an understanding of functionality, its representation on the right level of abstraction, knowledge of formal methods and test purpose;
- cannot verify all matches between environment and model.

Traditional testing or manual testing means executing the software in order to exercise and discover error without using any automation tools.

Manual testing also has its advantages and disadvantages:

Advantages of manual testing:

- still most handy and common method in the software industry;
- the formal spec is not needed;
- applicable directly on executable software;
- depends on tester's intuition and experience;
- can be done by any tester.

Disadvantages of manual testing:

- time-consuming, some automation tool exists (for running tests, organizing test data and reporting);
- not exhaustive, errors often survive;

- hard to reach 100% test coverage.

The aim of this thesis is to confirm the possibility of adaptation of model-based testing to scalable long-term autonomy testing of ROS-based robot software. To reach this goal the thesis focuses on the development of the integration and conformance testing toolkit TestIt.

1.2 Related work

This section gives an overview of the works that use model-based testing in distributed real-time systems with such tools as UPPAAL, UPPAAL TRON and DTRON.

First, we have a look at the development of related tools for model-based online black-box conformance testing of real-time systems.

To start with, the main primary tool used in this thesis is UPPAAL TRON. As reported in [13] it was presented in 2005 as a recent addition to the UPPAAL environment. It is an online testing tool which means that it is possible to generate and execute tests in real-time without breaks. With TRON it became possible to check the compatibility of inputs and outputs between a model and system under test. That functionality allows detecting errors of interaction between environment and implementation models in the early stages of design.

The paper [4] presents the DTRON tool which is a wrapper of UPPAAL TRON to support multicast messaging between the distributed test components. In this paper also examples of modelling and runtime limitations and considerations are given. The DTRON was created because of the need to work with complex human-assistive robots such as Scrub Nurse Robot [14]. There was a serious question how to be sure that software is at a high level of quality and at the same time how to guarantee the safety of robot actions and the development of this framework is adhered to these questions.

The paper [21] gives the complete overview of the DTRON framework. In the beginning, there is a theoretical basis of Cyber-Physical Systems principles which are used in the DTRON. Further, the DTRON software architecture with entire details of subsystems is described. Subsystems include the integration mechanism and the communication model. Most of the work is done by DTRON automatically. This concerns Reporters or Adapters

then communication via Spread toolkit and DTRON API. This makes developer's life easier because in TRON it would have to be done manually. DTRON has already been used in three distributed testing case studies. These are city street light controller network, interbank trading system and robot navigation system.

Next, we will consider the literature, which is directly related to model-based testing of robots.

The paper [14] includes results of using DTRON model-based distributed control framework for human-assistive robots. DTRON provided a flexible infrastructure to integrate robot's cognitive functions. Further, DTRON is easily adaptable and can be used on multiple hardware and operating systems.

The paper [15] is based on a previous paper [14] and is an extension to the Scrub Nurse Robot case-study. In this paper, authors are interested in conformance testing of using UPPAAL TRON tool to check the correctness of the system which is verified by on-line testing. In the article, the experience of researchers confirmed this.

The next paper [19] is closely related to this thesis. In this paper, model-based testing for robots built using Robot Operating System was applied with the aim to improve the software quality. According to the reported results, high code coverage was achieved. This helped to find unrelated problems in configurations, so it demonstrated that DTRON could be used as a validation tool as well. The main goal was to look at the robot behaviour when the environment is changing around the robot. In other words, to make the environment as close as possible to real conditions with which the robot needs to cope. In the article, it is described how to automatically generate the model from the topological map. This method should be adopted in this thesis as well and expanded to allow modifying it by structural coverage items – traps in the map model. In addition, the article gives an overview of other possible ways of testing robots built on ROS.

1.3 Thesis problem statement and main assumptions

The purpose of this thesis is to explore model-based testing tools for real-time distributed systems by integrating them into a ROS-based scalable long-term autonomy testing toolkit. The practical part of this thesis is to implement an adapter and interface between Distributed Testing Real-time system ONline (DTRON) and Robot Operating System

(ROS). In order to test a robot with generated Uppaal models by running integration and conformance tests, the Uppaal models should be generated automatically. Constructing Uppaal models manually is labour-intensive and time-consuming. The workflow incorporates mapping a topological map or the behaviour trees to model structures on the basis of which a full model will be created by adding timing and synchronization attributes. The input would be the behaviour of the robot, and as an output Uppaal model to which can add structural coverage items – traps. This approach is interesting because it should be possible to control the system under test with the aim to test it as fast as possible and to find as much as possible “defective” behaviours.

1.4 Thesis structure

The thesis is divided into four chapters. The first chapter gives an overview of the benefits of using model-based testing in comparison with manual testing, also provides an overview of using model-based testing in related works. The second chapter includes the theoretical foundations of model-based testing, its use in robotics, and the tools used in this work UPPAAL, UPPAAL TRON, DTRON, and behaviour trees. The third chapter includes the practical results about the adaptation of the model-based testing workbench TestIt. The fourth chapter contains a general description and test goals of autonomous platform navigating in the confined area.

2 Preliminaries

2.1 Model-based testing

A model is an abstract description of a system's behaviour. The model-based testing is a testing technique where the test cases are obtained from a formal specification or a model. MBT is typically black-box testing since models are usually built on the basis of the requirements that specify expected behaviour observable on the external interfaces of the system under test (SUT). The diagram of MBT main steps is depicted in Figure 2.

The purpose of modelling in conformance testing is to describe the system requirements. The model could be specialized depending on the tests cases, test environments and test strategies, but the model can be also more general covering several test cases. Test requirements are needed for test design. Based on the test requirements and model specification tests generation takes place. During the test execution, the output from the SUT is compared with the expected output described in the model. Depending on the result of the comparison, the test is completed with the result test decision *passed* or *failed*.

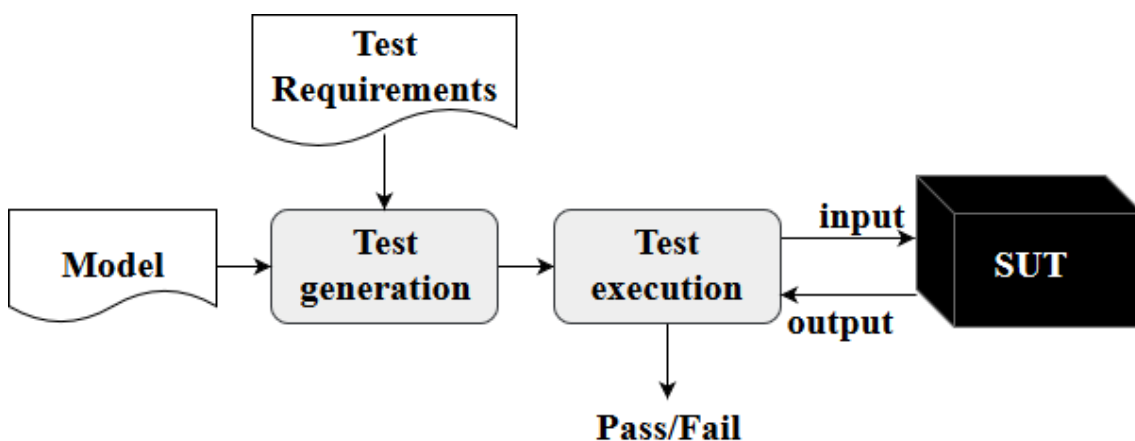


Figure 2. Example of the Model-based testing process.

Model-based testing is considered to be generally black-box testing which includes functional and non-functional tests without access to the internal structure of the SUT.

That means there is no knowledge of how the system is built, what code is running in it. We could only change the inputs and observe the behaviour of the system by outputs. The main purpose of black-box testing is error checking in functional and non-functional requirements of the SUT.

There are two types of model-based testing depending on how the test planning is done and test stimuli are generated. The first one is *offline* testing which means that the test suites are generated before executing the test while *online* testing means that the test suites are generated during the test execution.

We will consider online testing. It has several advantages. It is possible to run tests for several days. It is needed, for example, if we want to perform stress testing. Stress testing is a type of testing focused on determining how the system will perform under increased loads. Then, online testing can be easily adapted to non-determinism in real-time models that allows expecting an output in some interval of time and one more important feature the transition to the next step is carried out only after the output was obtained from the previous step.

The main reasons why it is worth using model-based testing are

- easier test suite maintenance because it is possible to link all the tests to system requirements.
- Human errors are eliminated since tests are generated automatically.
- With model-based testing, we could improve test quality since the computer can generate much more complex combinations of behaviours for the system under test compared to that a person can comprehend.
- When testing nondeterministic systems with online model-based testing the test suites could run as long as needed.

2.2 Uppaal Timed Automata

Uppaal is a toolkit for modelling, simulation (validation) and verification of real-time systems. It is jointly developed by Uppsala and Aalborg Universities [6]. “Real-time systems are defined as systems in which the correctness of an operation depends not only

on the logical result of computation, but also on the time at which the results are produced” [25].

Uppaal has three main parts a simulator, a model-checker engine and a graphical editor. The simulator allows to validate the system behaviour and determine at an early stage mismatch in the model because if there are errors in the model it is impossible to run the simulation. The model-checker engine helps to verify if the model satisfies certain correctness criteria expressed in Timed Computation Tree Logic (TCTL). The model description language used in the model editor was created to describe a nondeterministic system behaviour as networks of automata and it was extended with clock and data variables in order to be able to describe real-time systems.

The next definitions describe syntax and semantics on which the Uppaal model-checker is based:

“Definition 1 (Timed Automaton). *A timed automaton is a tuple (L, l_0, C, A, E, I) , where L is a set of locations, $l_0 \in L$ is the initial location, C is the set of clocks, A is a set of actions, co-actions and the internal τ -action, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and $I: L \rightarrow B(C)$ assigns invariants to location” [26].*

“Definition 2 (Semantics of TA). *Let (L, l_0, C, A, E, I) be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \rightarrow \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times (\mathbb{R}_{\geq 0} \cup A) \times S$ is the transition relation such that:*

- $(l, u) \xrightarrow{d} (l, u + d)$ if $\forall d': 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$, and
- $(l, u) \xrightarrow{a} (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ s. t. $u \in g, u' = [r \mapsto 0]u$, and $u' \in I(l')$,

where for $d \in \mathbb{R}_{\geq 0}, u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \setminus r$ ” [26].

2.2.1 The Uppaal modelling language

To support the model construction and update, the Uppaal modelling language has graphical form:

- A timed-automaton is defined as a graph with locations as nodes and edges as arcs between nodes (Figure 3).

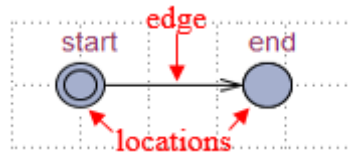


Figure 3. Example of the automaton with locations and edge.

- **Templates** are extended timed automata which can be instantiated with a set of parameters. The instances of templates are called processes. The executable model, defined in system declarations section can include one or more instances of each automaton template:

Example of system declaration: *system A, B;*

Example of parameter declaration: *int a, chan b, clock c;*

Example of instantiation: *A := B(i, 0, 1);*

- **Constants** must be of type integer:

Example of constant *a* with value 1 of type integer: *const int a = 1;*

- **Arrays** could be used for integer values, clocks, constants, and channels:

Example of array declaration: *chan c[5]; int[1,4] u;*

- **Initialisers** are needed to initialise integer variables and arrays.

Example of initialization: *int b := 2; int d[2] := {1, 2};*

By default, each variable is initialized with the minimal element of its type.

- **Bounded integer variable** with a range from -32768 to 32768 could be used in guards, invariants, assignments. The variables min and max bounds within this

range can be defined also in the declarations section and must be always satisfied, otherwise exceeding the bounds of type will cause an error.

Example of a bounded integer variable: `int[5, 10] d;`

- **Broadcast channels** consist of one sender and many receivers, the sender is never blocked.

Example of broadcast channel declaration: `broadcast chan c;`

- The **binary synchronisation** occurs when two synchronization actions with the same name, for example, channel `c`, from different processes synchronise where one is emitting (`c!`) and the other is receiving (`c?`), Figure 4.

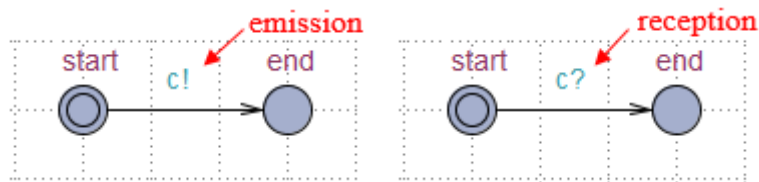


Figure 4. Example of synchronisation expression.

- **Initial location** in the template is used for initialisation of processes. The template must have only one location with the initial state and it is identified with a double circle, Figure 5.

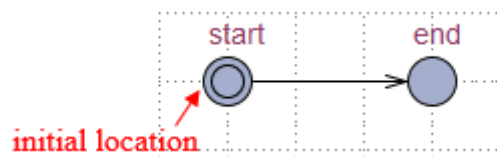


Figure 5. Example of initial location.

To support immediate actions the Uppaal modelling language has following features:

- **Urgent synchronisation channels.** It excludes any delay if a transition with urgent synchronisation action is enabled. It is not allowed to use clock guards on synchronisation transition with urgent actions, but invariants and data-variable guards are admissible.

- **Urgent Locations.** It excludes any delay in urgent location and because of that time is not allowed to pass. Urgent locations are identified with a “U” in a circle, Figure 6.

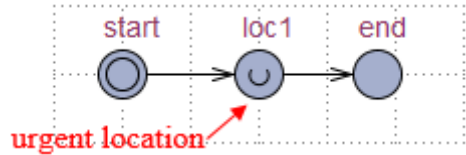


Figure 6. Example of urgent location.

- **Committed locations.** They exclude any delay in the location. The next transition must be taken from at least one of the committed locations and all other parallel automata executions in that time will be blocked. Committed locations are identified with a “C” in a circle, Figure 7.

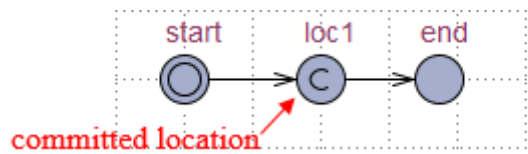


Figure 7. Example of committed location.

Uppaal modelling language has following expressions:

- **Invariant** specifies the condition under which the automaton can stay in that location, Figure 8.

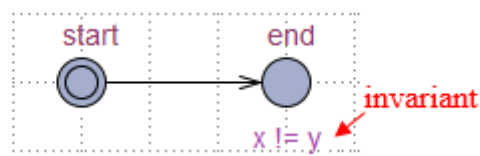


Figure 8. Example of invariant expression.

- **Guard.** Any expression that must be satisfied when passing from one location to another is a guard, Figure 9.

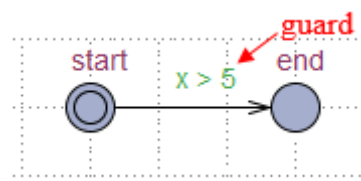


Figure 9. Example of guard expression.

- **Update.** By executing the transition from one state to another with update expression, a new value to the variable is calculated by the right-hand side expression of assignment and the variable is updated (Figure 10).

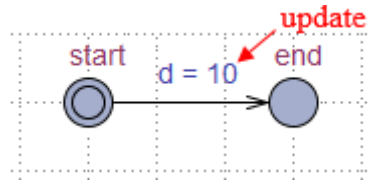


Figure 10. Example of update expression.

- **Selection.** During the transition from one state to another with selection expression, we could non-deterministically bind a value from a given range to an identifier, Figure 11.

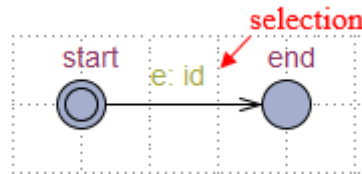


Figure 11. Example of selection expression.

2.2.2 Uppaal simulator

There are three possibilities how to use the simulator. The first way is to run the simulation manually by choosing which transition when and how to take next and by this to validate if that model works as supposed. The second opportunity is to simulate a system with a random mode, the simulator picks different transitions randomly. The third one is to import the file of witness or diagnostic trace that is produced by the verifier. This verification trace can be saved and used after verification for bug tracking.

The simulator window has four parts:

- **The control part** includes enabled transitions of the system, then simulation trace of different locations and buttons to control the simulation run either in manual or random mode.

- **The variable view** displays the values of integer and Boolean variables. The clock constraints are shown in interval form since the trace may have infinitely many time points for control and data states.
- In **the system view** all instantiated automata are displayed and active locations of their current states are highlighted.
- **The message sequence chart** is represented as a sequence diagram with transitions from one location to another and synchronisations between processes are visualized.

2.2.3 The Uppaal verifier

The Uppaal query language uses a simplified version of Timed Computation Tree Logic (TCTL). The syntax of a subset of the Timed Computation Tree Logic is defined as follows:

$$p ::= a.l|x + c \leq y + d | \neg p | p_1 \vee p_2 | A[] p_1 | A <> p_1 | E[] p_1 | z \text{ in } p$$

- p is a local property;
- a is process name and l is location;
- x, y are the variables or clocks;
- z is a clock;
- $c, d \in \mathbb{N}$ are the constants;
- p_1, p_2 are the TCTL formulae in Uppaal;
- The modality A means for all paths;
- The modality E means that exists a path;

The query language includes path formulae which quantify over traces or path and state formulae which describes individual states. With the state formulae, we could check if the timed automaton is in the location l , then check that value of a clock or some variable

satisfies value constraints and finally verify that there is no deadlock in the model. System is deadlocked if there is no transition from the current state or any of its delay successors.

The verifier is used to verify model with different properties:

- **Reachability properties.** $E \langle \rangle p$ “Exists eventually p ” means it is possible to reach a state in which p is satisfied. P must be found in at least one reachable state. An example is shown in the following Figure 12.

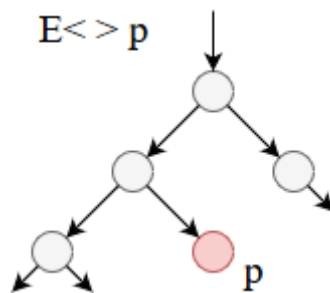


Figure 12. Example of reachability property.

- **Safety properties** assure that something bad will never happen. $A [] p$ “ p holds invariantly” that means p is true in all reachable states and $E [] p$ “ p is potentially always true” means there is still can be found a path in which p is true in all states. Example of safety properties is shown in the Figure 13.

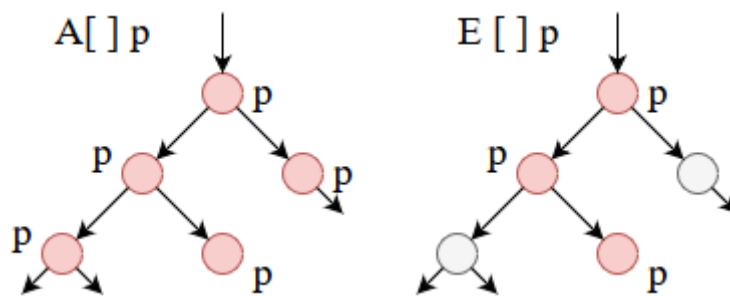


Figure 13. Example of safety properties.

- **Liveness properties** generally mean that after some action something should happen. $A \langle \rangle p$ “inevitable p ” that means for automaton it is possible to reach a state in all paths where p is true. $p \rightarrow q$ “ p lead to q ”, meaning in all paths where p becomes true then q will inevitable become true as well. Example of liveness properties is shown in the Figure 14.

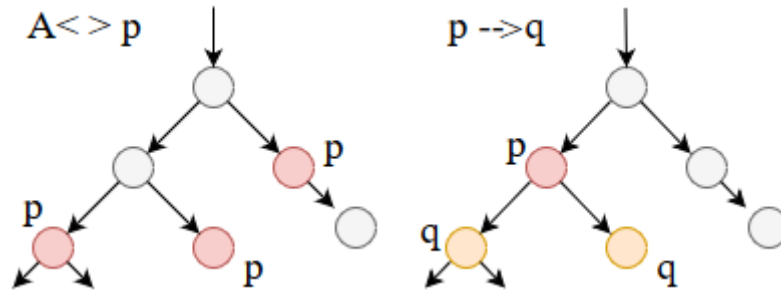


Figure 14. Example of liveness properties.

2.3 Uppaal TRON

Uppaal Testing Real-time systems ONLINE or simply TRON is an extension of the Uppaal engine for timed model-based testing. Real-time systems can be modelled, validated, and verified with Uppaal tool, but TRON is suitable for checking that the system under test behaves the same way as described in the Uppaal model.

The main point is on testing time and functional properties, where time properties are expressed in terms of time constraints. System input and output messages can be transmitted at different time points but must be controlled by TRON that it happens in the state with right invariant. TRON can use a different type of models for testing. This means that models can be both deterministic and non-deterministic.

In addition to generating test stimuli, TRON also works as an oracle, i.e. it monitors SUT inputs, outputs and checks their compatibility with those of the model.

2.4 DTRON

Distributed Testing Real-time systems ONLINE or DTRON is a command line application which extends TRON by enabling coordination and synchronisation of distributed tester components.

DTRON works together with Spread message serialisation service and Network Time Protocol. That co-use allows giving global timestamps to all events that arrive to the adapter of the system under test and via Spread service distribute these events to other subscribers. Thereby DTRON takes care of configuring the group memberships and message configuration.

The Spread toolkit provides a high performance messaging service that is resilient to faults across external or internal networks. Spread functions as a unified message bus for distributed applications, and provides highly tuned application-level multicast and group communication support. Spread services range from reliable message passing to fully ordered messages with delivery guarantees [30].

To start using DTRON it is necessary to create the Uppaal model and configure Spread toolkit. The DTRON starts to parse the model and searches for the input and output channels and related to them variables. Uppaal TRON API and Spread Group service register all found channels with prefix “i_” (inputs) and “o_” (outputs). Similarly, integer variables with their values could be registered to the publishable channel. This is done by appending a variable name to the channel name “i/o_channel_name_variable_name”. When the adapter receives a message through the Spread it will transmit the data between the input and output channels. The data in Spread is transmitted in byte array form. That allows transporting any type of data from different applications. DTRON configuration for remote testing is shown in the Figure 15.

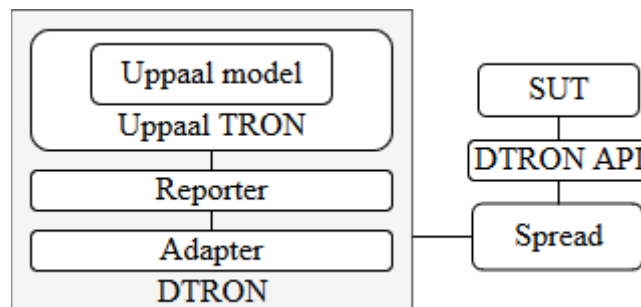


Figure 15. Example of DTRON configuration.

2.5 Behaviour trees

The Behaviour tree (BT) is an alternative Finite-State Machine (FSM) to represent the switching between different scenarios. The BT is presented as a directed rooted tree, whose nodes are possible variants of the agent behaviour, where the width of the tree indicates the number of available actions and the length of its branches define their complexity.

The BT was developed in the game industry as an alternative way to FSM to develop intelligent agents because while using FSM the complexity of the states increases rapidly.

The BT architecture is lacking this problem since for each state we do not need to prescribe its own decision logic.







The BT also has not less significant improvements over the FSM such as maintainability, scalability, reusability, goal-orientation, and parallelisation. Since the BT is represented as a tree structure, the structure defines transitions as well. This allows making nodes independent of each other that helps to change the code or modify tree structure without any difficulty. It is possible to make behaviour subtrees. As a result, that raises readability of the BT. Another feature due to independence of nodes and subtrees is reusability which means we could use the same piece of code or a structure in any other projects and do not have to write the same behaviours of agents from scratch. With goal-oriented feature we could create specific agent goal, add it to the subtree and the flexibility of the behaviour tree will not be affected. Not less important feature is parallelisation. With it, we could define which nodes with their all children will work in parallel and with that not lose the control of the BT. This is achieved due to the fact that parallel nodes work independently from each other.

The BT consists of control flow nodes and execution nodes. Each node has a parent except the root node and each parent has at least one child. The control flow nodes are divided into four categories *fallback*, *sequence*, *parallel*, and *decorator* while execution nodes have only two categories *action* and *condition*.

Execution nodes do not have any child they are necessary for calculations and returning the status value. A condition node monitors states and checks if an expected state has been met or not. It will return success if the state has been met and fail if not. An action node is responsible for changing the agent state. If it is possible to change the state then it will return success otherwise fail. The decorator node has a single child and it tries to change the behaviour of the child by changing the signal frequency or playing around with the return value. The parallel node sends a signal to all its N children to work in parallel and it will succeed only if M children defined by user return status success, fail if $N - M + 1$ return fail. In all other options, the status will be running. The sequence node ticks its children from the left to the right until one of them returns fail or running. If all children return status success only then the sequence also will return success. Otherwise, it will return fail or running looking at what status the parent has. The Fallback node works in opposite way compared with the sequence node. It returns fail if all

children return fail otherwise success or running if one of the children return success or running [31]. The summary of node types and how they work could be found in Table 1.

Table 1. The node types of the BT [31].

Node type	Symbol	Succeeds	Fails	Running
Fallback		If one child succeeds	If all children fail	If one child return Running
Sequence		If all children succeed	If one child fails	If one child returns Running
Parallel		If $\geq M$ children succeed	If $> N - M + 1$ children fail	If $< M$ children succeed or If $< N - M + 1$ children fail
Decorator		Custom	Custom	Custom
Action		Upon completion	If impossible to complete	During completion
Condition		If true	If false	Never

2.6 MBT for ROS-based robotics

The Robot Operating System (ROS) is a collection of tools that gives the possibility to implement the necessary behaviour of the robot to perform a specific task. The ROS quality assurance process requires that all packages should be tested. ROS supports different types of testing starting from the basic unit testing and finishing with integration tests.

Our task is to test a robot using model-based testing. To make it possible it will be necessary to implement a separate adapter for the communication between DTRON and ROS. For testing also a model of the topological map or behaviour tree, the environment has to be created where the robot will be tested. The following Figure 16 shows the required components for testing: the model, TRON, adapter, DTRON and ROS which in this case is a system under test.

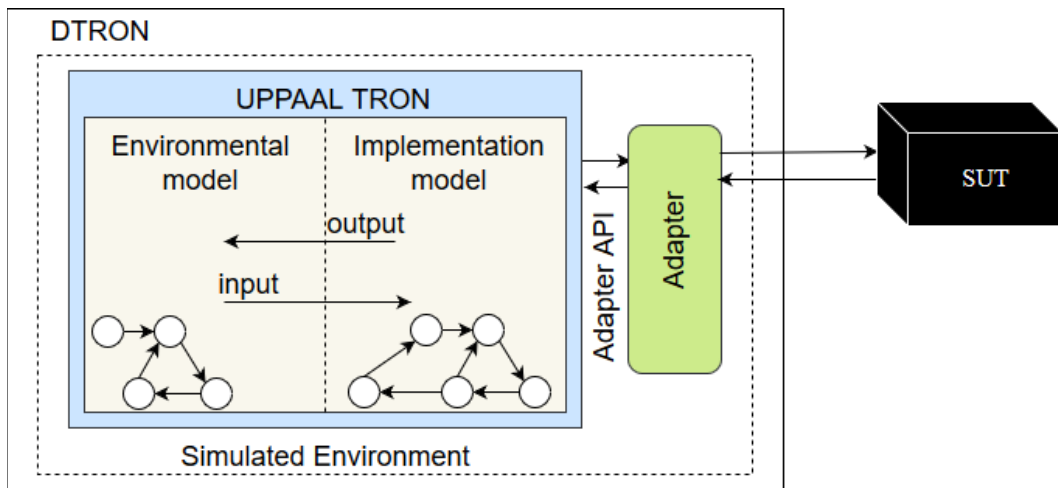


Figure 16. Relationships between DTRON, TRON AND SUT.

3 Adaptation of the MBT workbench TestIt

3.1 TestIt architecture and design principles

TestIt is a scalable long-term autonomy testing toolkit for robot operating system. Before proceeding with the design principles of TestIt, it is necessary to consider Docker platform because the TestIt has a similar one.

Docker is an open platform for developing, shipping, and running applications. With Docker, it is possible to separate application that allows to deliver software quickly and to be sure that on any machine it will work equally [33]. Docker uses a client-server architecture, communication between the client and the server is carried out through the REST API. All the commands from the client side are sent using command line interface (CLI) to the REST API to control or interact with the interfaces of Docker server.

What about TestIt here is an analogy with Docker. TestIt represents a daemon process which is a type of long-running program. Commands are also transmitted through command line interface. TestIt has its own qualities such as using parallel test servers so that testing is more efficient. The second feature is that TestIt does not depend on the simulator software and allows testing a robot developed in any simulation environment such as Gazebo, Morse, V-REP, Stage, UWSim. The model of robot behaviour can be created on the basis of the behaviour trees or SMACH [37]. Later the model could be converted to the specific model checker, for example, NModel, DIVINE or Uppaal. The main point of TestIt is a possibility to test systems with any models and any simulation environments without restrictions. The TestIt architecture is shown in Figure 17.

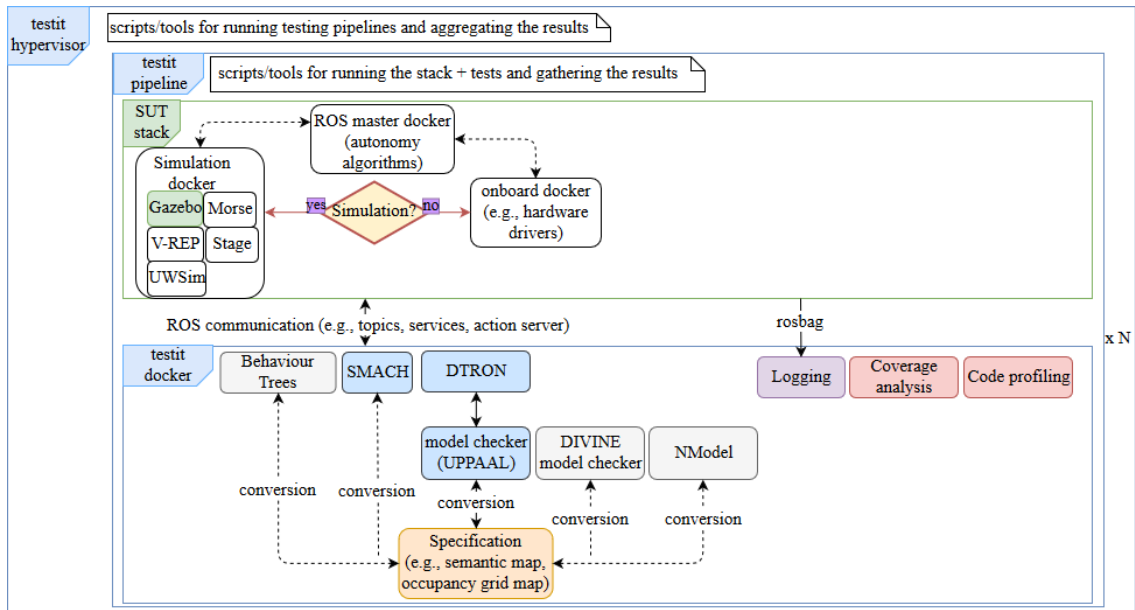


Figure 17. TestIt architecture.

3.2 Integrating DTRON with TestIt

The whole development will take place in Docker container platform. With Docker, we can describe the instruction of all commands in the file which must be executed in order to install required software or launch necessary services. This is done in order to facilitate the opportunity for other people to repeat the same actions to run tests with TestIt toolkit, as well as, reduce the number of errors when using the toolkit on different platforms.

To make communication between DTRON and SUT possible we need to create an adapter. The adapter will be responsible for sending and receiving messages in a suitable type and the Spread service will transmit these messages between two systems. Since Spread sends data in byte array form, it is necessary to structure it for easier processing. For this protocol buffers from Google will be used. Protocol buffers perform the function of serializing structured data. They are not based on any platform and programming language. Protocol buffers are also faster, smaller, and simpler than XML.

3.2.1 Protocol buffers installation

First, we need to build and install Protocol buffers version 2.3.0. This can be done using the following commands:


```
$ git clone https://github.com/ooici/protobuf-2.3.0.git
$ cd protobuf-2.3.0
$ ./autogen.sh && ./configure
$ make && make check && make install
```

Now we need to define message formats in a *.proto* file. At the beginning of the file we need to specify the version of protocol buffers since our version is 2.3.0, so we will indicate syntax as “*proto2*”. Then we need to add a message to each data structure that we want to serialize. Each field in the message must have a name and a type. We are going to have synchronization messages with the sender name and its variables where each variable will have a name and a value.

The protocol buffers file has the following structure:

```
syntax = "proto2";

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

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

The next step is to compile this file. Because the adapter is implemented in C++ language, then the classes need to be generated in C++ as well.

```
$ protoc -I="PATH_to_src_directory" --cpp_out="PATH_to_destination"
"PATH_to_proto_file"
```

This generates header and source files from protocol buffers file.

3.2.2 Spread toolkit installation

The next step is the installation of Spread toolkit. This can be done with the next commands:

```
$ git clone https://github.com/glycerine/spread-src-4.4.0.git
$ cd spread-src-4.4.0
$ ./configure && make && make install
```

After installation, it is necessary to configure the spread configuration file. The template of that file could be found in “*etc*” directory. Spread service needs to work with administrator rights that is why we need to make the following changes.

Open “*spread.conf*” file with any text editor, delete comment symbol # before the lines:

```
#DaemonUser = spread
#DaemonGroup = spread
```

After saving the configuration file need to create a new user and group in Ubuntu operating system, then add that user to the created group.

```
$ useradd spread && addgroup add spread
$ usermod -a -G spread spread
```

To run Spread daemon as root also needs to create the runtime directory:

```
$ mkdir -m 777 /var/run/spread
```

Now we can start the Spread daemon and be sure that it runs as root successfully.

```
$ cd spread-src-4.4.0/sbin
$ ./spread -c /spread-src-4.4.0/etc/spread.conf
```

3.2.3 Adapter

Adapter between SUT and DTRON is written in C++ language. The adapter converts incoming and outgoing messages to the appropriate format using protocol buffers. To implement the test adapter the adapter template was used as a basis. That template was created during the research of “Model-based integration testing of ROS packages: a mobile robot case study” [19].

At the very beginning, we need to establish a connection with Spread service to be sure that the service is running.

```

#include "sp.h"

bool ConnectionActive;
int ret;

ret = SP_connect(Spread_name, User, 0, 1, &Mbox, Private_group);
if( ret < 0 ) {
    ConnectionActive = false;
} else {
    ConnectionActive = true;
}

```

We need to remember that for each synchronized channel different adapters need to be implemented. But since we will only use one synchronized channel, then this does not concern us. Therefore, to receive messages from the inside and outside, we need to add channels to the group. This can be done by calling next function:

```

SP_join(*Mbox, channel_name);

```

Now we consider how to convert message data to the protocol buffers format and then send it. We will need to include generated header proto file from chapter 3.2.1 for message data serialization. And then using the Spread multicast function we can send the message to the specific group.

```

#include <xtaprotolib.pb.h>
#include "sp.h"

int ret;
Sync response;
response.set_name(group);
std::string data = "";
response.SerializeToString(&data);
ret = SP_multicast(*Mbox, AGREED_MESS, group, 1, strlen(data.c_str()),
data.c_str());

```

Reading the messages is almost as simple as sending. At first, we need to specify message structure to simplify message parsing.

```

struct SpreadMessage {
    int Type;
    char* Sender;
    char* Group;
    char* Msg;
};

```

Then via the Spread functions, we get the message and parse it, and lead to a readable form.

```

SpreadMessage spreadMessage;
static char message[102400];
char sender[MAX_GROUP_NAME];
char target_groups[100][MAX_GROUP_NAME];
int num_groups, service_type, endian_mismatch, ret;
membership_info memb_info;
int16 mess_type;
service_type = 0;
ret = SP_receive(*Mbox, &service_type, sender, 100, &num_groups,
target_groups, &mess_type, &endian_mismatch, sizeof(message), message);
if(ret < 0) {
    SP_error(ret);
}
if(Is_regular_mess(service_type)) {
    message[ret] = 0;
} else if( Is_membership_mess(service_type)){
    ret = SP_get_memb_info(message, service_type, &memb_info);
    if (ret < 0) {
        SP_error(ret);
    }
}
spreadMessage.Type = service_type;
spreadMessage.Sender = new char[MAX_GROUP_NAME];
spreadMessage.Sender = sender;
spreadMessage.Group = new char[MAX_GROUP_NAME];
spreadMessage.Group = target_groups[0];
spreadMessage.Msg = new char[102400];
spreadMessage.Msg = message;
Sync sync;
sync.ParseFromString(spreadMessage.Msg);
if (sync.name() != "") {
    printf("[Google protocol buffers]: Channel: '%s', Sender: '%s',
VariableName: '%s' VariableValue: '%d'\n", sync.name().c_str(), sender,
sync.variables(0).name().c_str(), sync.variables(0).value());
} else{
    printf("Received incorrectly formed message from %s in %s: %s\n", sender,
spreadMessage.Group, message);
}

```

We figured out how to send and receive messages. Now it is possible to establish communication with ROS for sending test inputs to the robot. The target may be the location on a map at which the robot needs to arrive and with the help of ROS *move_base* action server or *topological_navigation*, this can be done. After the adapter sends the coordinates of the location to the robot, it will wait for confirmation whether the robot reached this place or not. This will be necessary for further testing. If the robot was able to reach the intended location, the following location as navigation goal will be sent. If not, then the test will fail.

Following code is an example of the implementation of the robot navigation test scenario using *move_base* action server.

```
ROS_INFO("Received a message - %s!", name.c_str());
actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> ac_;
std::string state = boost::lexical_cast<std::string>(args["state"]);
std::string node_name = node_map_[state];

move_base_msgs::MoveBaseGoal goal;
double x, y;

nh_.getParam(node_name + "/x", x);
nh_.getParam(node_name + "/y", y);

goal.target_pose.header.frame_id = "/map";
goal.target_pose.pose.position.x = x;
goal.target_pose.pose.position.y = y;
goal.target_pose.pose.position.z = 0;

goal.target_pose.pose.orientation.x = 0;
goal.target_pose.pose.orientation.y = 0;
goal.target_pose.pose.orientation.z = 0;
goal.target_pose.pose.orientation.w = 1;

if (ac_.isServerConnected()) {
    ac_.sendGoal(goal);
    ac_.waitForResult();
    actionlib::SimpleClientGoalState state = ac_.getState();
    if (state == actionlib::SimpleClientGoalState::SUCCEEDED) {
        ROS_INFO("Robot reached goal x: %.3f y: %.3f", x, y);
    } else {
        ROS_ERROR("Action result was not SUCCEEDED!");
    }
} else {
    ROS_ERROR("Action server not connected!");
}
```

3.2.4 Dockerfile

The installation of necessary components of TestIt can be built automatically with Docker by reading the instructions from a Dockerfile which can be viewed in Appendix 2 – Dockerfile or in GitHub [41].

The instruction includes not only integration but also an example of using this toolkit. That example can be used as a template for testing any other robots with different behaviour scenarios. And expanded to be used with different simulators, simulation environments, and model checkers. Note that it will be necessary to download TRON manually. Impossible to download it directly because it is required to fill out the license form on the website. An instruction on how to do this is available at the end of the Appendix 2 – Dockerfile

For testing, separate Dockerfiles are prepared. The examples of how the installation instructions of testing and SUT look like can be viewed in Appendix 3 – Dockerfile of SUT and Appendix 4 – Dockerfile of testing base image respectively.

First, one has to check if Docker has been installed. Then you need to save instructions from Appendix 2 – 4 with names “*Dockerfile*” and type “*File*”. After that navigate to the directory where you saved the base image file and build it by typing:

```
$ docker build --no-cache -t testit:latest .
```

To build SUT and testing images type next commands:

```
$ docker build --no-cache -t testit_tb_sut .  
$ docker build --no-cache -t testit_tb_testit .
```

3.3 Test generation for DTRON

With an aim to test successful integration of DTRON with TestIt, and to provide an example of how TestIt toolkit can be used the following models will be created and tested.

3.3.1 Mapping topological maps to Uppaal TA

First of all, it is necessary to create a topological map. Topological maps could be represented using discrete units – *waypoints* represented graphically as nodes. The nodes connected with edges generally define a topological map.

The topological map used in this thesis is built with software stack developed in the STRANDS project. Before creating a topological map, we need a file of robot waypoints. Waypoints are specified by coordinates of locations to where the robot will have to go in the future while testing it. To run STRANDS project software ROS Kinetic Kame is used. For a robot navigation simulation, Gazebo simulator (version 7.12.0) is used which makes it possible to rapidly check if the waypoints are located at the right place or not and edit the environment for test needs.

As a map “*willow-2010-02-18-0.10*” was chosen and as a world “*willowgarage_world*” was selected. The term “world” is used to describe a collection of robots and objects (such as buildings, tables, and lights), and global parameters including the sky, ambient light, and physics properties [32]. The map, in turn, describes the virtual world in which the robot can navigate around.

It is possible to create own map but then it is necessary to consider the capabilities of the computer used for simulation. The system requirements for running gazebo simulator can be checked in gazebo official webpage.

First, one has to check if ROS kinetic release has been installed and it has “*turtlebot_gazebo*” package. Then you need the source files to have access to the ROS commands.

```
$ source /opt/ros/kinetic/setup.bash
```

Now you can launch gazebo simulator with existing world file:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch  
world_file:="PATH_to_world_file"
```

To navigate the robot in the simulator you need to open a new terminal and run the next command:

```
$ roslaunch kobuki_keyop keyop.launch
```

To start creating map open new terminal and type:

```
$ roslaunch turtlebot_gazebo gmapping_demo.launch
```

Rviz tool can be used to visualize the map building process. For that, open new terminal and type:

```
$ roslaunch turtlebot_rviz launchers view_navigation.launch
```

Now one can navigate the robot around the world and build a relevant map. When the map is satisfying, you will have to save it.

```
$ rosrun map_server map_server -f "map_name"
```

Remember that after saving the map and closing all terminals, it will be impossible to open the map file and continue building it. You have to start all over again. If your computer does not meet the minimum requirements of map building service then you should not waste time creating a map. For instance, for the author of this thesis it took eight hours to build that piece of map that can be seen in Figure 18 (a) and if to compare with Figure 18 (b) then clearly it was not worth of it, because in such an incomplete map as (a) the robot very often will lose the path and return errors. I used a computer with the next specifications Intel® Core™ i3-2350M CPU 2.30GHz, 8GB RAM, Intel® HD Graphics 3000.

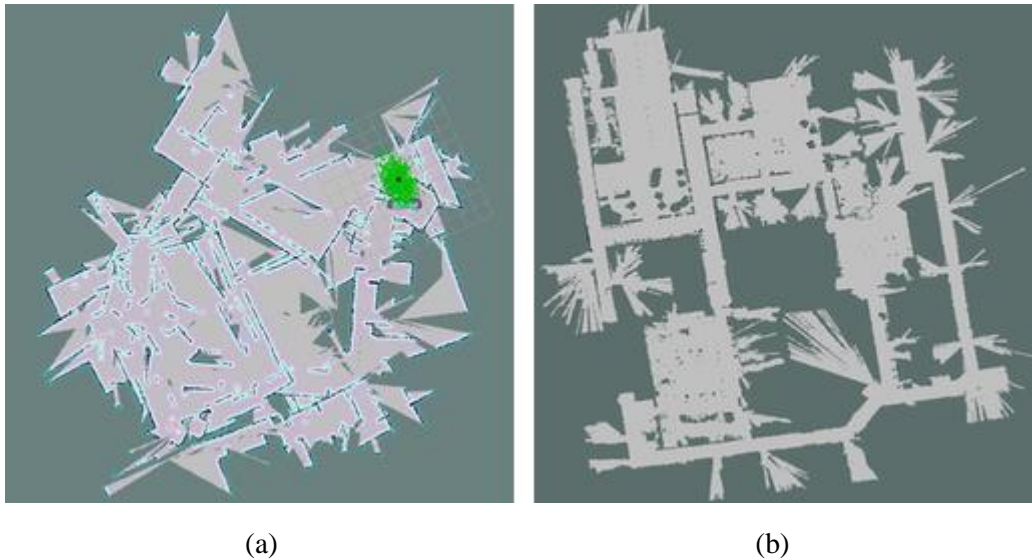


Figure 18. Example of willow garage map created by the author of this thesis (a) and taken from turtlebot gazebo package (b).

To generate the waypoints for the topological map we will navigate a robot around the world and in certain places ask the robot to return the position coordinates.

Source ROS files, export world and map files paths to parameters and launch gazebo simulator:

```
$ source /opt/ros/kinetic/setup.bash
$ export TURTLEBOT_GAZEBO_WORLD_FILE="PATH_to_world_file"
$ export TURTLEBOT_GAZEBO_MAP_FILE="PATH_to_map_file"
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

In the new terminal launch keyboard teleoperation:

```
$ roslaunch kobuki_keyop keyop.launch
```

We could manually call command to get robot position from the map and then save it to the file.

```
$ rosrn tf tf_echo /map /base_link
```

Another way is to use a script that will automatically, for example, save robot position every 10 seconds to the file in the following format:

```
position.x,position.y,position.z,orientation.x,orientation.y,orientation.z,orientation.w
```

Example of that script could be found in GitHub [39].

Now we can create a topological map file from the waypoint file using next command where “*input_file.csv*” is waypoints file name and “*output_file.tmap*” is the name of the topological map file.

```
$ rosrund topological_utils tmap_from_waypoints.py input_file.csv  
output_file.tmap
```

The topological map file has following structure:

```
node:  
  #node name  
  WayPoint1  
  waypoint:  
    #position of the node  
    position.x,position.y,position.z,orientation.x,  
    orientation.y,orientation.z,orientation.w  
  edges:  
    #List of connections from this node, action  
    WayPoint2, move_base  
  vertices:  
    #positions around the node  
    position.x1,position.y1  
    position.x2,position.y2  
    position.x3,position.y3  
    position.x4,position.y4  
    position.x5,position.y5  
    position.x6,position.y6
```

To be able to use topological navigation the topological map need to be added to the database.

Create a directory for your database to be stored and launch MongoDB:

```
$ mkdir /opt/ros/mongodb_store  
$ roslaunch mongodb_store mongodb_store.launch  
db_path:=/opt/ros/mongodb_store
```

Insert topological map to the database with the following command where “*PATH_to_TMAP*” is the path to the topological map file, “*dataset_name*” is the name of the dataset for the database, and “*map name*” is the topological map name for the database:

```
$ rosrn topological_utils insert_map.py "PATH_to_TMAP" "dataset_name"
"map_name"
```

Eventually, when the topological map is generated, we can create a model from that. For this purpose, a program in Java language was written. The source code is available in GitHub [38].

To generate the Uppaal model type next command where “*input_file.tmap*” has to be replaced with your topological map file name.

```
$ java -jar generateModelFromTmap.jar input_file.tmap
```

As a result, Uppaal model file will be generated with the structure shown in Appendix 1 – Example of Uppaal model. In this example only two waypoints between which the robot can move are shown. The model includes two processes as depicted in Figure 19. The first process displays the set of waypoints and relationships between them, and the second process synchronises channels to move the robot from one location to another.

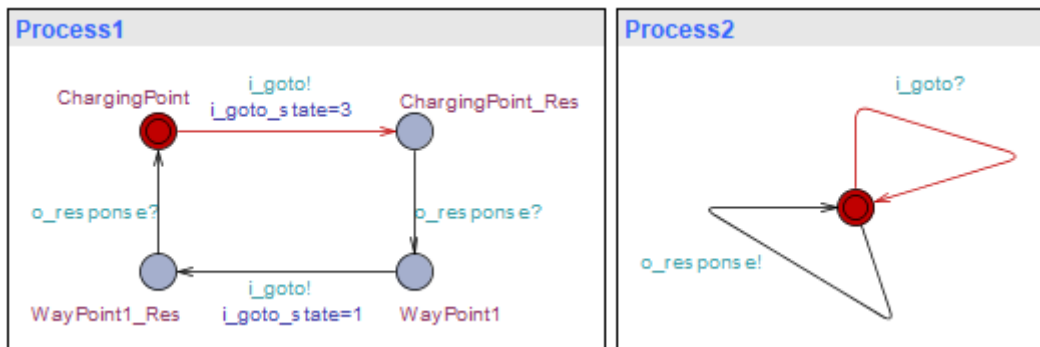


Figure 19. Example of Uppaal model processes.

4 Case study: Autonomous platform navigating in the confined area

4.1 General description and test goals

After the required models, files, scripts, adapters were created, it is necessary to test successful integration of DTRON with TestIt toolkit. For this purpose, an example was created.

In the example, we determined that we will use Gazebo simulator, as an autonomous platform Turtlebot 2 is used and navigation takes place in one part of the willow garage world. The system under test uses ROS Kinetic Kame to run the simulation. The whole system works under Ubuntu 16.04 xenial that runs in the Docker container.

Since we are working with Docker, we can guarantee that it will work on all devices in the same way. But only if all the required components are installed during the Docker image build process. In order to exclude the possibility of an error during the installation of a component, the components should be installed in the order of the queue or before the component will be used by another component. This is implemented in the form of using symbol “&&” between installation commands of different components in Dockerfile. If an error occurs, then the build process will stop, otherwise, we will be able to successfully run a Docker image.

Further, it was necessary to cover the world with waypoints to which the robot should get. Waypoints are located not only in the open space but also behind all possible obstacles. This allows testing whether the robot will find a way to the waypoint if suddenly there will be an obstacle on the way.

The major goals of current testing are as follows:

- To ensure that during the Docker build process there will be no errors.
- To make sure that TestIt starts testing after the SUT has been fully loaded.

- Communication between DTRON and SUT is established.
- The robot can reach a certain waypoint on the map described in the Uppaal model.
- Using a topological map navigation, the robot must first reach the intermediate waypoints before reaching the end waypoint.

4.2 Generating Uppaal TA models

For the current example in order to cover the map a model consisting of 31 waypoints was created. An example of the node map is shown in the Figure 20. In this case, nodes are connected in series, but for more thorough testing of the robot behaviour, the route needs to be changed. That is realizable by mixing waypoints from the source file and generating a new model. If we use a topological map navigation, then we need to change “edges” in the topological map file and therefore update the data in the database.



Figure 20. Example of node map.

How to generate the topological map and Uppaal model are described in subsection 3.3.1. All the generated files used for this example are available in GitHub [40].

4.3 Generating tests

The test Oracle monitors the navigation of the robot. We can transfer the waypoint coordinates which we want to check to the Oracle. If the robot reaches the waypoint, then

the test will return “*pass*”, and if not then the test will return “*fail*”. The test will also fail if the response timeout from the test is exceeded. The example of this test is shown in Appendix 5 – Oracle Test1.

For example, we want to check whether the robot can reach the “*WayPoint5*” with coordinates ($x: 15.568, y: 18.316$). The test can be started with the following command:

```
$ python oracle.py 15.568 18.316
```

We can leave the test running for a long time and at this time to test other robot behaviours since TestIt allows to run tests in parallel. During the test, the robot is navigating from waypoint to waypoint that takes a specified time interval. The Oracle will monitor whether the robot always succeeds in reaching the waypoint or not. The source code of that test is shown in Appendix 6 – Oracle Test2.

Using topological navigation it is even easier to find out if the robot has reached a certain waypoint because before reaching the endpoint, it is necessary to pass intermediate waypoints. Therefore, if the endpoint is reached, then the test will succeed, otherwise, it fails. Implementation of this test is available in Appendix 7 – Oracle Test3.

For example, if we want to make sure that the robot can get from the initial waypoint to the “*WayPoint8*” passing through intermediate points, the test can be started with the following command:

```
$ python oracle.py WayPoint8
```

4.4 Executing tests

To run the tests with TestIt, it is necessary to describe the execution instructions in the configuration file. For example, we need to specify in the file how to run SUT and test images, where the test scripts are located, what parameters need to be transferred to the scripts, duration, timeout and other parameters. Sample configuration file with explanations is given in Appendix 8 – Configuration file.

The precondition for running the tests is successfully installed TestIt base, SUT and TestIt test images. Further, configuration file needs to be generated. Then, using the following commands, we can begin testing.

Run with docker TestIt base image:

```
$ docker run --name testit -v /var/run/docker.sock:/var/run/docker.sock -it testit bash
```

It is necessary to mount Docker socket to run docker commands inside the container.

Navigate to the “*catkin_ws*” directory and source files:

```
$ cd /catkin_ws  
$ source devel/setup.bash
```

To start TestIt daemon, type next command:

```
$ roslaunch dtron turtlebot.launch
```

The file for navigation with simple *move_base* action server is specified by default. But it can be changed by passing the path to the configuration file in the parameter.

For example, if we want to test topological navigation:

```
$ roslaunch dtron turtlebot.launch  
config:=/catkin_ws/src/testit/dtron/turtlebot/cfg/config_top_nav.yaml
```

To set the state and to bring up open new terminal and type:

```
$ rosrn testit testit_command.py bringup
```

And finally, to start the test type:

```
$ rosrn testit testit_command.py test
```

4.5 Testing results

At the time of writing this thesis, the Docker build process of the images succeeds. In the future, may need to update the versions of the required software.

During testing was determined that it takes 160 seconds to start SUT, and 30 seconds to start TestIt tests. This was taken into account in the configuration file. Thus, we can be sure that Testit starts testing after the SUT has been fully loaded.

Since we were able to test the robot using *move_base* and *topological_navigation* action servers, then no errors were found with the communication between SUT and DTRON.

It takes an hour and a half for the robot to get around all the 31 waypoints defined in this example. In consequence, the conformance test was performed for ten hours. During this time the robot managed to make practically seven complete laps around the map and there was no error. Seven times to get around the map is certainly not enough, because an error can be detected on the 500th lap. This requires powerful computing resources and time, but this was not the purpose of this work. It should be noted that from time to time the robot can suddenly not understand where it is or where was not a wall the robot would think that there is a wall. But this is due to the simulator and has nothing to do with the use of TestIt toolkit.

Using a topological navigation, the robot was successfully able to reach the goal bypassing at first intermediate waypoints. And just like the *move_base* testing, the time required to get passed all waypoints is large. But here it takes even more time because of the many connections between the waypoints. These connections can be removed, but then it will not be different from the *move_base* navigation.

5 Summary

The thesis provides an overview of the benefits of using model-based testing in robotics in comparison with manual testing. The main advantages of MBT are that the tests are generated and executed automatically. Consequently, the human errors are eliminated, and the test suite maintenance is simple because of possibility to link all the test cases to system requirements.

In the work it is described in detail how Uppaal toolkit is applied for modelling, simulation, and verification of real-time systems. The toolchain used also includes a description of the TRON which is an extension of the Uppaal engine for timed model-based testing. Its performs conformance testing by checking if the SUT behaves the same way as described in the Uppaal model. The extension of TRON is DTRON. DTRON extends TRON by enabling coordination and synchronization of distributed tester components.

To describe the behaviour of the robot, it is possible to use behaviour tree which is an alternative Finite-State Machine to represent the switching between different scenarios. The main advantage of using behaviour tree is that for each state there is no need to create its own logic. Thereby, the complexity of the states does not increase rapidly, as it occurs in the FSM.

The practical part describes the adaptation of the MBT workbench TestIt which is a scalable long-term autonomy testing toolkit for robot operating system. The main point of TestIt is a possibility to test systems in parallel with any models and any simulation environments without restrictions.

Due to the fact that the model is generated automatically based on the required behaviour of the robot it is necessary to establish a connection between DTRON and SUT using test adapters. If the test inputs are not given simultaneously it is enough to create one adapter to receive messages from and send messages to SUT using only one synchronization channel. The adapter is responsible for sending and receiving messages in a suitable type and the Spread service transmits these messages between two systems. The data in Spread

is transmitted in byte array form and in order to have messages in standard format Google protocol buffers are used. Google protocol buffers do not restrict using any specific platform or programming language.

The Dockerfile includes the instructions of all necessary steps to install and configure required components of TestIt. This gives the opportunity to new users to proceed directly to testing the systems as quickly as possible rather than installing the necessary software and fixing the errors that occurred during the configuration.

To test the successful integration of DTRON with TestIt, and to provide an example of how TestIt toolkit can be used a topological map was created on the basis of which a model was created. To cover the map 31 waypoints were used between which the robot can move using *topological_navigation* and *move_base* action servers. For this were involved STRANDS project to build the topological map and Gazebo simulator to navigate the robot. Both methods were tested within ten hours, during which the robot did seven full tours around the map, and no error was detected. This does not mean that there are no defects in the SUT, we just made sure that the integration was successful, and it can be used for testing. But for more thorough testing this requires powerful computing resources and time.

References

- [1] Soome, Toomas. Hajussüsteemid, 2004. [WWW] <http://kodu.ut.ee/~mroos/hs/hs2.pdf>
- [2] Markvardt, Maili. Sissejuhatus mudelipõhisesse testimisse. [WWW] http://193.40.251.102/tiki-download_wiki_attachment.php?attId=314 (19.03.2018)
- [3] Tšukrejeva, Jekaterina. Tarkvarasüsteemi kvaliteet ja testimine. [WWW] <http://maurus.ttu.ee/sts/wp-content/uploads/2015/10/IDK0071-Loeng-7-Kvaliteet-ja-testimine.pdf> (19.03.2018)
- [4] Anier, Aivo. Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems, 2016. [WWW] <https://digi.lib.ttu.ee/i/?6133>
- [5] Kruusamägi, Age. Model-based testing of distributed systems: Tallinn streetlight system case-study, 2016. [WWW] <https://digi.lib.ttu.ee/i/?7133>
- [6] UPPAAL web page. [WWW] <http://uppaal.org/> (19.03.2018)
- [7] UPPAAL TRON web page. [WWW] <http://people.cs.aau.dk/~marius/tron/index.html> (19.03.2018)
- [8] DTRON web page. [WWW] <https://cs.ttu.ee/dtron/> (19.03.2018)
- [9] ROS web page. [WWW] <http://www.ros.org/> (19.03.2018)
- [10] Topological Map. [WWW] http://strands.readthedocs.io/en/latest/strands_navigation/wiki/Topological-Map-Definition.html (19.03.2018)
- [11] TestIt. [WWW] <https://github.com/GertKanter/testit> (19.03.2018)
- [12] T-UPPAAL: Online Model-based Testing of Real-time Systems. <http://people.cs.aau.dk/~marius/tron/ASE2004.pdf> (21.03.2018)
- [13] K. G. Larsen, M. Mikucionis, B. Nielsen, A. Skou. Testing Real-time Embedded Software using UPPAAL-TRON an industrial case study. [WWW] <http://people.cs.aau.dk/~marius/tron/emsoft05.pdf> (22.03.2018)
- [14] A. Anier, J. Vain. Model Based continual planning and control for assistive robots. [WWW] <http://dijkstra.cs.ttu.ee/~aivo/dtron/publications/healthinf-cr.pdf> (22.03.2018)
- [15] A. Anier, J. Vain. Timed automata based provably correct robot control. [WWW] https://cs.ttu.ee/dtron/publications/Aivo_bec2010_v6-final.pdf (21.03.2018)
- [16] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou. Testing Real-Time Systems Using UPPAAL. [WWW] <https://pdfs.semanticscholar.org/2067/a094dd7839d66ee7863c0255e50d2ebf8604.pdf> (21.03.2018)
- [17] J. Ernits, M. Veanes, J. Helander. Model-Based Testing of Robots with NModel. [WWW] <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/mbtora.pdf> (21.03.2018)
- [18] K. G. Larsen, M. Mikucionis, B. Nielsen. Online Testing of Real-time Systems Using UPPAAL. [WWW]

- <https://pdfs.semanticscholar.org/33ae/a20c39800cde2466aafe0f01d341c2a09bfb.pdf>
(21.03.2018)
- [19] J. Ernits, E. Halling, G. Kanter, J. Vain. Model-based integration testing of ROS packages: a mobile robot case study. In 2015 IEEE European Conference on Mobile Robots.
- [20] Marko Kääramees. A Symbolic Approach to Model-based Online Testing, 2012. [WWW] <https://digi.lib.ttu.ee/i/?806>
- [21] A. Anier, J. Vain, L. Tsiopoulos. DTRON: a tool for distributed model-based testing of time critical applications. [WWW] http://www.kirj.ee/public/proceedings_pdf/2017/issue_1/proc-2017-1-75-88.pdf
(22.03.2018)
- [22] Dejanira Araiza-Illan, Anthony G. Pipe and Kerstin Eder. Model-based Test Generation for Robotic Software Automata versus Belief-Desire-Intention Agents. [WWW] <https://arxiv.org/pdf/1609.08439.pdf>
- [23] Dejanira Araiza-Illan, Tony Pipe, Kerstin Eder. Model-Based Testing, Using Belief-Desire-Intentions Agents, of Control Code for Robots in Collaborative Human-Robot Interactions. [WWW] <https://arxiv.org/pdf/1603.00656.pdf> (21.03.2018)
- [24] Olli-Pekka Puolitaival. Model-based testing tools. [WWW] <https://www.cs.tut.fi/tapahtumat/testaus08/Olli-Pekka.pdf> (20.04.2018)
- [25] Professor John A. Stankovic. Real-Time Computing. [WWW] <https://pdfs.semanticscholar.org/0195/c0b09a69cfbca2d50e74671a82f224779653.pdf>
(20.04.2018)
- [26] G. Behrmann, A. David, K. G. Larsen. A Tutorial on UPPAAL 4.0. [WWW] <https://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf> (20.04.2018)
- [27] G. Behrmann, K. Larsen. Into to Uppaal. [WWW] <http://people.cs.aau.dk/~adavid/RTSS05/uppaal-intro.pdf> (21.04.2018)
- [28] Tomáš Poch. Uppaal. [WWW] <http://d3s.mff.cuni.cz/seminar/download/2007-10-03-Poch-uppaal.pdf> (21.04.2018)
- [29] Julián Proenza. The Uppaal Model Checker. [WWW] <http://ppedreiras.av.it.pt/resources/empse0809/slides/TheUppaalModelChecker-Julian.pdf>
(21.04.2018)
- [30] The Spread Toolkit. [WWW] <http://www.spread.org/> (21.04.2018)
- [31] M. Colledanchise and P. Ögren. Behavior Trees in Robotics and AI. [WWW] <https://arxiv.org/pdf/1709.00084.pdf> (22.04.2018)
- [32] Gazebo terminology. [WWW] http://gazebosim.org/tutorials?tut=build_world
(21.04.2018)
- [33] Docker overview. [WWW] <https://docs.docker.com/engine/docker-overview/#the-docker-platform> (23.04.2018)
- [34] “EU SPEEDS project. INRIA research report n. 8147 November 2012” [WWW] <https://hal-univ-tlse2.archives-ouvertes.fr/hal-01178467/document> (24.04.2018)
- [35] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Softw. Test., Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, 2012. [WWW] <https://doi.org/10.1002/stvr.456> (24.04.2018)
- [36] M. Utting and B. Legeard. “Practical model-based testing : a tools approach”. Morgan Kaufmann Publishers, 2006.

- [37] SMACH. [WWW] <http://wiki.ros.org/smach> (30.04.2018)
- [38] Source code of generating Uppaal model from tmap. [WWW]
<https://github.com/arturgummel/dtronpack/tree/master/generateModelFromTmap>
- [39] Source code of saving robot position to the file. [WWW]
https://github.com/arturgummel/dtronpack/blob/master/pose_publisher.cpp
- [40] Generated files (waypoints, map, model) [WWW]
<https://github.com/arturgummel/dtronpack/tree/master/generateModelFromTmap/examples>
- [41] Integration package with example. [WWW] <https://github.com/arturgummel/dtrontestit>

Appendix 1 – Example of Uppaal model

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE nta PUBLIC "-//Uppaal Team//DTD Flat System 1.1//EN"
"http://www.it.uu.se/research/group/darts/uppaal/flat-1_2.dtd">
<nta>
  <declaration>chan i_goto, o_response; int i_goto_state;</declaration>
  <template>
    <name x="5" y="5">robot_map</name>
    <declaration>// Place local declarations here.</declaration>
    <location id="id1" x="17" y="17">
      <name>ChargingPoint</name>
    </location>
    <location id="id2">
      <name>ChargingPoint_Res</name>
    </location>
    <location id="id3" x="18" y="15">
      <name>WayPoint1</name>
    </location>
    <location id="id4">
      <name>WayPoint1_Res</name>
    </location>
    <init ref="id1"/>
    <transition>
      <source ref="id1"/>
      <target ref="id2"/>
      <label kind="synchronisation">i_goto!</label>
      <label kind="assignment">i_goto_state=3</label>
    </transition>
    <transition>
      <source ref="id2"/>
      <target ref="id3"/>
      <label kind="synchronisation">o_response?</label>
    </transition>
    <transition>
      <source ref="id3"/>
      <target ref="id4"/>
      <label kind="synchronisation">i_goto!</label>
      <label kind="assignment">i_goto_state=1</label>
    </transition>
    <transition>
      <source ref="id4"/>
      <target ref="id1"/>
      <label kind="synchronisation">o_response?</label>
    </transition>
  </template>
  <template>
    <name>sut</name>
    <location id="id3" x="0" y="0"/>
    <init ref="id3"/>
    <transition>
```

```

    <source ref="id3"/>
    <target ref="id3"/>
    <label kind="synchronisation" x="10" y="93">i_goto?</label>
    <nail x="-8" y="-102"/>
    <nail x="127" y="-51"/>
  </transition>
  <transition>
    <source ref="id3"/>
    <target ref="id3"/>
    <label kind="synchronisation" x="-170" y="68">
      o_response!
    </label>
    <nail x="8" y="119"/>
    <nail x="-119" y="42"/>
  </transition>
</template>
<system>
  Process1 = robot_map();
  Process2 = sut();
  system Process1, Process2;
</system>
<queries/>
</nta>

```

Appendix 2 – Dockerfile of the base image

```
# VERSION 0.0.1
FROM ubuntu:xenial
MAINTAINER Artur Gummel <artur.gummel@ttu.ee>
LABEL Description="TestIt! ROS Testing toolkit base docker image"
RUN apt-get update && \
#install wget, vim, git, autoconf and scon
    apt-get install -y wget vim git autoconf scon && \
#install ROS Lunar desktop full
    sh -c 'echo "deb http://packages.ros.org/ros/ubuntu xenial main" >
/etc/apt/sources.list.d/ros-latest.list' && \
    sh -c 'apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
--recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116' && \
    apt-get update && \
    apt-get install -y ros-lunar-desktop-full && \
    rosdep init && \
    rosdep update && \
#create project directory and clone TestIt
    mkdir -p /catkin_ws/src && \
    /bin/bash -c "source /opt/ros/lunar/setup.bash && cd /catkin_ws/src
&& catkin_init_workspace" && \
    cd /catkin_ws/src && \
    git clone https://github.com/GertKanter/testit.git && \
#install Java 8, Lunar packages, mongodb
    apt-get install software-properties-common -y && \
    add-apt-repository ppa:webupd8team/java -y && \
    apt-get update && \
    echo debconf shared/accepted-oracle-license-v1-1 select true |
debconf-set-selections && \
    echo debconf shared/accepted-oracle-license-v1-1 seen true |
debconf-set-selections && \
    apt-get install oracle-java8-installer -y && \
    apt-get install -y ros-lunar-rviz && \
    apt-get install -y ros-lunar-map-server && \
    apt-get install -y ros-lunar-move-base-msgs && \
    apt-get install -y python-pymongo mongodb && \
    mkdir /opt/ros/mongodb_store && \
    apt-get install -y ros-lunar-navfn && \
    apt-get install -y ros-lunar-costmap-2d && \
#clone example for TestIt and STRANDS packages
    git clone https://github.com/arturgummel/dtrontestit.git
/catkin_ws/src/testit/dtron && \
    git clone https://github.com/arturgummel/dtronpack.git
/catkin_ws/dtronpack && \
```



```

/bin/bash -c "/catkin_ws/src/testit/dtron/scripts/build_stuff.sh"
&& \
cd /catkin_ws/src && \
git clone https://github.com/strands-project/strands_navigation.git
&& \
git clone https://github.com/strands-project/mongodb_store.git && \
git clone https://github.com/strands-project/strands_apps.git && \
git clone https://github.com/strands-project/strands_movebase && \
git clone https://github.com/strands-project/fremen.git && \
mv /catkin_ws/src/fremen/FremenServer
/catkin_ws/src/strands_navigation/ && \
rm -r /catkin_ws/src/fremen && \
git clone https://github.com/GT-RAIL/robot_pose_publisher.git && \
/bin/bash -c "source /opt/ros/lunar/setup.bash && cd /catkin_ws &&
catkin_make" && \
#configure Spread
useradd spread && usermod -a -G spread spread && \
mkdir -m 777 /var/run/spread && \
echo 'export PATH=$PATH:/catkin_ws/spread/sbin' >> ~/.bashrc && \
echo 'source /opt/ros/lunar/setup.bash' >> ~/.bashrc && \
#install docker
apt-get update && \
apt-get -y install apt-transport-https \
ca-certificates \
curl \
gnupg2 \
software-properties-common && \
curl -fsSL https://download.docker.com/linux/$(. /etc/os-release;
echo "$ID")/gpg > /tmp/dkey; apt-key add /tmp/dkey && \
add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/$(. /etc/os-
release; echo "$ID") \
$(lsb_release -cs) \
stable" && \
apt-get update && \
apt-get -y install docker-ce && \
#install required architecture for running TRON
dpkg --add-architecture i386 && \
apt-get update && \
apt-get install -y libc6:i386 libncurses5:i386 libstdc++6:i386
CMD bash

#download UPPAAL TRON manually from
http://people.cs.aau.dk/~marius/tron/download.html
#You will need to accept the license!
#unzip uppaal-tron-1.5-linux.zip in dtronpack directory:
#cd /catkin_ws/dtronpack && unzip uppaal-tron-1.5-linux.zip

```

Appendix 3 – Dockerfile of SUT

```
FROM ros:kinetic-robot-xenial
MAINTAINER Gert Kanter <gert.kanter@ttu.ee>
LABEL Description="TestIt! ROS Testing toolkit tutorial SUT image"
RUN apt-get update && \
    apt-get install -y ros-kinetic-turtlebot-navigation ros-kinetic-
turtlebot-gazebo wget xvfb && \
    /bin/bash -c "echo \"deb
http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs`
main\" > /etc/apt/sources.list.d/gazebo-stable.list" && \
    wget http://packages.osrfoundation.org/gazebo.key -O - | apt-key
add - && \
    apt-get update && \
    apt-get upgrade -y && \
    mkdir -p /catkin_ws/src && \
    /bin/bash -c "source /opt/ros/kinetic/setup.bash && cd
/catkin_ws/src && catkin_init_workspace" && \
    cd /catkin_ws/src && \
    git clone https://github.com/mission-control-ros/mission_control &&
\
    git clone https://github.com/ros/executive_smach.git && \
    git clone https://github.com/ros-perception/slam_gmapping.git && \
    cd slam_gmapping && \
    git fetch origin pull/56/head:nodelet_fix && \
    git checkout nodelet_fix && \
    cd .. && \
    /bin/bash -c "source /opt/ros/kinetic/setup.bash && cd /catkin_ws
&& catkin_make"
CMD ['/bin/bash', '-c "source /catkin_ws/devel/setup.bash && rosrn
mission_control start_move_base_in_docker.sh && tail -f /dev/null"']
```

Appendix 4 – Dockerfile of testing base image

```
FROM testit:latest
MAINTAINER Artur Gummel <artur.gummel@ttu.ee>
LABEL Description="TestIt! ROS Testing toolkit docker image"
ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y --no-install-recommends apt-
utils && \
    echo 'export PATH=$PATH:/catkin_ws/spread/sbin' >> ~/.bashrc && \
    echo 'source /opt/ros/lunar/setup.bash' >> ~/.bashrc && \
#insert map to database
    /bin/bash -c "/catkin_ws/src/testit/dtron/scripts/insertmap.sh"
CMD bash
```

Appendix 5 – Oracle Test1

```
#!/usr/bin/python
import roslib
import actionlib
import rospy
import testit_oracles.testit_gazebo
import sys
from move_base_msgs.msg import *
from geometry_msgs.msg import PoseStamped

def resultCallback(data):
    goalStatus = data.status.text
    rospy.loginfo(goalStatus)
    if goalStatus == 'Goal reached.':
        return True
    else:
        return False

def goalCallback(data, waypointX, waypointY):
    x = float(data.pose.position.x)
    y = float(data.pose.position.y)
    rospy.loginfo('Current goal X: %f Y: %f', x, y)

    if waypointX - 0.1 <= x <= waypointX + 0.1 and waypointY - 0.1 <= y \
        <= waypointY + 0.1:
        msgResult = rospy.wait_for_message('/move_base/result',
            MoveBaseActionResult)
        reachedLocation = resultCallback(msgResult)
        if reachedLocation:
            rospy.loginfo('WayPoint x: %f y: %f reached successfully',
                x, y)
            return 0 #success
        else:
            rospy.loginfo('Did not reach WayPoint x: %f y: %f', x, y)
            return 1 #fail
    else:
        rospy.loginfo('Wait for a next goal')
        return 3 #continue

if __name__ == '__main__':
    rospy.init_node('testit_tb_tutorial')
    rate = rospy.Rate(2)
    waypointX = float(sys.argv[1])
    waypointY = float(sys.argv[2])
```

```
while not rospy.is_shutdown():
    msg = rospy.wait_for_message('/move_base/current_goal',
                                  PoseStamped)
    result = goalCallback(msg, waypointX, waypointY)
    if result == 0:
        sys.exit(0) #success
    elif result == 1:
        sys.exit(1) #fail
    rate.sleep()
```

Appendix 6 – Oracle Test2

```
#!/usr/bin/python
import roslib
import rospy
import sys
from move_base_msgs.msg import *
from geometry_msgs.msg import PoseStamped

def resultCallback(data):
    goalStatus = data.status.text
    if goalStatus == "Failed to find a valid plan. Even after executing
recovery behaviors.":
        return False
    else:
        return True

def goalCallback(data):
    x = float(data.pose.position.x)
    y = float(data.pose.position.y)
    rospy.loginfo('Current goal X: %f Y: %f', x, y)
    msgResult = rospy.wait_for_message("/move_base/result",
MoveBaseActionResult)
    reachedLocation = resultCallback(msgResult)
    if not reachedLocation:
        rospy.loginfo("Did not reach WayPoint x: %f y: %f", x, y)
        return 1 #fail
    else:
        return 3 #continue

if __name__ == "__main__":
    rospy.init_node("testit_tb_tutorial")
    rate = rospy.Rate(2)
    counter = 10
    if len(sys.argv) == 2:
        counter = int(sys.argv[1])
    while not rospy.is_shutdown():
        msg = rospy.wait_for_message("/move_base/current_goal",
PoseStamped)
        result = goalCallback(msg)
        if result == 3:
            counter -= 1
            if counter <= 0:
                sys.exit(0) #success
        elif result == 1:
            sys.exit(1) #fail
        rate.sleep()
```

Appendix 7 – Oracle Test3

```
#!/usr/bin/python
import rospy
import actionlib
import rospy
import sys
from topological_navigation.msg import *

def goalCallback(data):
    target = data.goal.target
    goalId = data.goal_id.id
    rospy.loginfo('Current goal %s', target)

    msgResult = rospy.wait_for_message('/topological_navigation/result'
        , GotoNodeActionResult)
    rospy.loginfo(msgResult)
    reachedLocation = msgResult.result.success
    reachedTargetId = msgResult.status.goal_id.id
    if reachedLocation and goalId == reachedTargetId:
        rospy.loginfo('%s reached successfully', target)
        return 0 #success
    elif not reachedLocation and goalId == reachedTargetId:
        rospy.loginfo('Did not reach %s', target)
        return 1 #fail
    else:
        rospy.loginfo('Wait for a next goal')
        return 3 #continue

if __name__ == '__main__':
    rospy.init_node('testit_tb_tutorial')
    reachWayPoint = sys.argv[1]
    rate = rospy.Rate(2)
    while not rospy.is_shutdown():
        msg = rospy.wait_for_message('/topological_navigation/goal',
            GotoNodeActionGoal)
        if reachWayPoint == msg.goal.target:
            result = goalCallback(msg)
            if result == 0:
                sys.exit(0) #success
            elif result == 1:
                sys.exit(1) #fail
        rate.sleep()
```

Appendix 8 – Configuration file

```
tests:
- tag: "Scenario #1" # identifier for reporting
  pipeline: "" # empty for any
  source: "/testit_tests/01" # test scenario source directory (SMACH
state machine, UPPAAL model etc) inside TestIt docker container
  launch: "" # how to execute this test (run command) in TestIt
container, if empty, then assumed that test is not explicitly executed
(already started at runSUT and oracle is used to determine pass/fail)
  oracle: "./testit_tests/01/oracle/oracle.py 16.646 21.344" #
determining whether pass/fail, if empty = "launch" execution result will
be used to determine pass/fail
  timeout: 300 # time in seconds for timeout (0 for no timeout)
  timeoutVerdict: False # if timeout occurs, declare the test as this
(False = fail, True = success)
  bagMaxSplits: "" # empty = use default
  bagDuration: "" # empty = use default
- tag: "Scenario #2" # identifier for reporting
  pipeline: "" # empty for any
  source: "/testit_tests/01" # test scenario source directory (SMACH
state machine, UPPAAL model etc) inside TestIt docker container
  launch: "" # how to execute this test (run command) in TestIt
container, if empty, then assumed that test is not explicitly executed
(already started at runSUT and oracle is used to determine pass/fail)
  oracle: "./testit_tests/01/oracle/oracle.py 25.226 27.470" #
determining whether pass/fail, if empty = "launch" execution result will
be used to determine pass/fail
  timeout: 1200 # time in seconds for timeout (0 for no timeout)
  timeoutVerdict: False # if timeout occurs, declare the test as this
(False = fail, True = success)
  bagMaxSplits: "" # empty = use default
  bagDuration: "" # empty = use default
- tag: "Scenario #3" # identifier for reporting
  pipeline: "" # empty for any
  source: "/testit_tests/02" # test scenario source directory (SMACH
state machine, UPPAAL model etc) inside TestIt docker container
  launch: "" # how to execute this test (run command) in TestIt
container, if empty, then assumed that test is not explicitly executed
(already started at runSUT and oracle is used to determine pass/fail)
  oracle: "./testit_tests/02/oracle/oracle.py" # determining whether
pass/fail, if empty = "launch" execution result will be used to determine
pass/fail
  timeout: 1800 # time in seconds for timeout (0 for no timeout)
  timeoutVerdict: False # if timeout occurs, declare the test as this
(False = fail, True = success)
  bagMaxSplits: "" # empty = use default
  bagDuration: "" # empty = use default
```



```

configuration:
  bringupSUT: "" # how to bring up a pipeline server/docker SUT (general
case), you can use "[[]]" for replacing
  bringupSUTDelay: 0 # duration to wait after command
  bringupSUTTimeout: 1 # seconds (0 for no timeout, but you have to
specify bringup_finish_trigger then or tests will not be run)
  bringupSUTFinishTrigger: "-" # command to test whether startup is
finished, "-" = no trigger
  runSUT: "docker run --rm --net=rosnetwork --env
ROS_HOSTNAME=[[masterHost]] --env
ROS_MASTER_URI=http://[[masterHost]]:11311 --name [[masterHost]] -dt
testit_tb_sut:latest /bin/bash -c \"source /catkin_ws/devel/setup.bash &&
roslaunch mission_control start_move_base_in_docker.sh && tail -f
/dev/null\"" # run SUT
  runSUTDelay: 90 # duration to wait for SUT to come up (roscore
initialization)
  runSUTTimeout: 90
  runSUTFinishTrigger: "-"
  stopSUT: "docker kill [[masterHost]]"
  stopSUTDelay: 0 # duration to wait after stopping the SUT
  stopSUTTimeout: 5
  stopSUTFinishTrigger: "-"
  teardownSUT: "" # how to clean up after finishing (shut down
server/docker) (general case)
  teardownSUTDelay: 0 # duration to wait after teardown
  teardownSUTTimeout: 5
  teardownSUTFinishTrigger: "-"
  bringupTestIt: "" # bring up the pipeline server (in the cloud for
example)
  bringupTestItDelay: 0 # duration to wait after command
  bringupTestItTimeout: 1
  bringupTestItFinishTrigger: "-"
  runTestIt: "docker run --rm --
volume=/catkin_ws/src/testit/dtron/turtlebot/testit_tests:/testit_tests
--net=rosnetwork --env ROS_VERSION=[[rosVersion]] --env
ROS_HOSTNAME=[[testitHost]] --env
ROS_MASTER_URI=http://[[masterHost]]:11311 --name [[testitHost]] -dt
testit_tb_testit /bin/bash -c \"
/catkin_ws/src/testit/dtron/turtlebot/scripts/run_adapter.sh && tail -f
/dev/null\"" # how to bring up a pipeline TestIt (general case), you can
use "[[]]" for replacing
  runTestItDelay: 10 # duration to wait after command
  runTestItTimeout: 10
  runTestItFinishTrigger: "-"
  stopTestIt: "docker kill [[testitHost]]" # general case pipeline
stopping
  stopTestItDelay: 0 # duration to wait after command
  stopTestItTimeout: 5
  stopTestItFinishTrigger: "-"
  teardownTestItDelay: 0 # duration to wait after command
  teardownTestItTimeout: 5 # empty string = use default
  teardownTestItFinishTrigger: "-" # command to test whether startup is
finished, "-" = no trigger

```

```

bagEnabled: False # True=rosvbag record, False=don't bag
bagMaxSplits: 5 # total bag duration = maxsplits*duration
bagDuration: 15 # seconds
pipelines:
- tag: "Pipeline #1" # identifier for reporting
  rosVersion: "lunar"
  ssh: "-" # "-" means no ssh command wrapping, execute docker commands
on localhost, bringup/teardown are not wrapped, run/stop + test commands
are wrapped
  masterHost: "sut1"
  testitHost: "testit1"
  masterIP: "-" # where SUT roscore is running (used if masterHost is
not defined) ("- means none)
  testitIP: "-" # where TestIt docker container is running (used if
testitHost is not defined) ("- means none)
  testItVolume: "$(rospack find dtron)/turtlebot/testit_tests/" # where
TestIt volume is located in the pipeline (test scenarios + bags are
stored there)
  bringupSUT: "" # empty string = use default
  bringupSUTTimeout: "" # empty string = use default
  bringupSUTFinishTrigger: "" # empty string = use default
  runSUT: "" # empty string = use default
  teardownSUT: "" # custom teardown for this pipeline
  teardownSUTTimeout: "" # empty string = use default
  teardownSUTFinishTrigger: "" # empty string = use default
  bringupTestIt: "" # empty string = use default
  bringupTestItTimeout: "" # empty string = use default
  bringupTestItFinishTrigger: "" # empty string = use default
  runTestIt: "" # empty string = use default
  teardownTestIt: "" # custom teardown for this pipeline
  teardownTestItDelay: "" # duration to wait after command
  teardownTestItTimeout: "" # empty string = use default
  teardownTestItFinishTrigger: "" # empty string = use default

```

Appendix 9 – Example of passed Test1

```
$ roslaunch dtron turtlebot.launch
config:=/catkin_ws/src/testit/dtron/turtlebot/cfg/config_top_nav.yaml
process[testit_daemon-2]: started with pid [3580]
[INFO] [1525544472.691523]: Loading configuration from
/catkin_ws/src/testit/dtron/turtlebot/cfg/config_top_nav.yaml...
[INFO] [1525544472.845837]: TestIt daemon started...
[INFO] [1525544478.730549]: Start all pipelines...
[INFO] [1525544478.737346]: [Pipeline #1] Setting state to BRINGUP
[INFO] [1525544478.739631]: Pipeline #1 starting...
[INFO] [1525544478.743507]: [Pipeline #1] Executing bringup SUT...
[INFO] [1525544478.755034]: [Pipeline #1] Done!
[INFO] [1525544478.758033]: [Pipeline #1] Waiting for delay duration
(0)...
[INFO] [1525544478.759049]: [Pipeline #1] Waiting for the bringup to
finish...
[INFO] [1525544479.744866]: ...
[INFO] [1525544479.761099]: [Pipeline #1] Done!
[INFO] [1525544479.763621]: [Pipeline #1] Executing bringup TestIt...
[INFO] [1525544479.770802]: [Pipeline #1] Done!
[INFO] [1525544479.771541]: [Pipeline #1] Waiting for delay duration
(0)...
[INFO] [1525544479.773444]: [Pipeline #1] Waiting for the bringup to
finish...
[INFO] [1525544480.775225]: [Pipeline #1] Done!
[INFO] [1525544481.750981]: Pipeline #1 finished with True
[INFO] [1525544486.275770]: Acquiring pipeline for test 'Scenario #1'
[INFO] [1525544486.279273]: Acquired pipeline Pipeline #1
[INFO] [1525544486.281376]: [Pipeline #1] Running SUT...
[INFO] [1525544486.283662]: [Pipeline #1] Executing SUT to run...
[INFO] [1525544486.285002]: [Pipeline #1] Executing "docker run --rm --
net=rosnetwork --env ROS_HOSTNAME=sut1 --env
ROS_MASTER_URI=http://sut1:11311 --name sut1 -dt testit_tb_sut:latest
/bin/bash -c "source /catkin_ws/devel/setup.bash && rosrun
mission_control start_move_base_in_docker.sh && tail -f /dev/null"
588077b6545aa550d2c91e944c6eb286c624534a4fb531ad891437503c0d47bc
[INFO] [1525544489.225981]: [Pipeline #1] Waiting for delay duration
(90)...
[INFO] [1525544579.314949]: [Pipeline #1] (run) ..
[INFO] [1525544594.335467]: [Pipeline #1] (run) ..
[INFO] [1525544609.354133]: [Pipeline #1] (run) ..
[INFO] [1525544624.371705]: [Pipeline #1] (run) ..
[INFO] [1525544639.387914]: [Pipeline #1] (run) ..
[INFO] [1525544654.408228]: [Pipeline #1] (run) ..
[INFO] [1525544669.423119]: [Pipeline #1] Execution done!
[INFO] [1525544669.424464]: [Pipeline #1] Running TestIt...
```

```

[INFO] [1525544669.426452]: [Pipeline #1] Executing TestIt to run...
[INFO] [1525544669.429798]: [Pipeline #1] Executing "docker run --rm --
volume=testit_tests:/testit_tests --net=rosnetwork --env
ROS_VERSION=lunar --env ROS_HOSTNAME=testit1 --env
ROS_MASTER_URI=http://sut1:11311 --name testit1 -dt testit_tb_testit
/bin/bash -c
"/catkin_ws/src/testit/dtron/turtlebot/scripts/run_top_nav.sh && tail -f
/dev/null""
f9c9cd3719a934800415c61b46bc954d2cd18e33ea2f15d90abfccc165ca0104
[INFO] [1525544673.507411]: [Pipeline #1] Waiting for delay duration
(25)...
[INFO] [1525544698.538570]: [Pipeline #1] (run) ..
[INFO] [1525544713.578253]: [Pipeline #1] (run) ..
[INFO] [1525544723.609492]: [Pipeline #1] Execution done!
[INFO] [1525544723.637867]: [Pipeline #1] Executing tests in TestIt
container...
[INFO] [1525544723.668648]: [Pipeline #1] Launching test 'Scenario #1'
[INFO] [1525544725.342714]: [Pipeline #1] Executing oracle...
[INFO] [1525544767.081646, 11.602000]: Current goal X: 14.650612 Y:
16.331187
[INFO] [1525544767.097278, 11.602000]: Wait for a next goal
[INFO] [1525544811.783285, 22.207000]: Current goal X: 18.351491 Y:
15.117878
[INFO] [1525544811.784432, 22.207000]: Wait for a next goal
[INFO] [1525544881.373912, 43.458000]: Current goal X: 15.568533 Y:
18.316429
[INFO] [1525544881.374627, 43.458000]: Wait for a next goal
[INFO] [1525544918.031074, 56.733000]: Current goal X: 17.957311 Y:
22.658847
[INFO] [1525544918.031991, 56.733000]: Wait for a next goal
[INFO] [1525544959.964177, 71.405000]: Current goal X: 16.646096 Y:
21.344108
[INFO] [1525544991.509076, 81.247000]: Goal reached.
[INFO] [1525544991.514035, 81.247000]: WayPoint x: 16.646096 y: 21.344108
reached successfully
[INFO] [1525544991.515369, 81.247000]: GOT RETURN STATEMENT 0
[INFO] [1525544992.379063]: [Pipeline #1] TEST PASS!
[INFO] [1525544992.380725]: [Pipeline #1] Stopping TestIt container...
[INFO] [1525544992.388351]: [Pipeline #1] Executing TestIt to stop...
[INFO] [1525544992.391408]: [Pipeline #1] Executing "docker kill testit1"
testit1
[INFO] [1525544995.207010]: [Pipeline #1] Waiting for delay duration
(0)...
[INFO] [1525544995.211155]: [Pipeline #1] (stop) ..
[INFO] [1525545000.240120]: [Pipeline #1] Execution done!
[INFO] [1525545000.244690]: [Pipeline #1] Stopping SUT...
[INFO] [1525545000.246446]: [Pipeline #1] Executing SUT to stop...
[INFO] [1525545000.251781]: [Pipeline #1] Executing "docker kill sut1"
sut1
[INFO] [1525545002.968978]: [Pipeline #1] Waiting for delay duration
(0)...
[INFO] [1525545007.976677]: [Pipeline #1] Execution done!

```