TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Kairit Sims 183839IVSM

# DISTRIBUTED MODEL-BASED TESTING OF TALLINN CITY INFORMATION SYSTEM TEELE

Master’s thesis

Supervisor: Jüri Vain
PhD

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kairit Sims 183839IVSM

# MUDELIPÕHINE HAJUSTESTIMINE TALLINNA INFOSÜSTEEMI TEELE NÄITEL

Magistritöö

Juhendaja: Jüri Vain
PhD

Tallinn 2020

# 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: Kairit Sims

12.05.2020

# Abstract

The goal of thesis "Distributed Model-Based Testing of Tallinn City Information System Teele" is to validate and evaluate the distributed model-based testing methodology and tools in real software development project. First part of thesis introduces model-based testing process and UPPAAL, UPPAAL TRON and DTRON tools. Second part of thesis uses distributed model-based testing methodology and tools in Tallinn City information system Teele development project which was previously tested only manually. As a result, Teele's project team was rather not satisfied with new model-based testing process not because the testing process itself but with its suitability specifically for current development project. In addition, the tools used are complex to learn and to use especially when there is not enough supporting materials and examples available for quick bringing into use.

This thesis is written in English and is 27 pages long, including 4 chapters, 18 figures and 6 tables.

# Annotatsioon

Mudelipõhine hajustestimine Tallinna infosüsteemi Teele näitel

Magistritöö eesmärgiks on valideerida ja hinnata mudelipõhise hajustestimise metoodikat ja tööriistu tarkvaraarenduse projektis. Esimene pool tööst tutvustab mudelipõhise hajustestimise olemust ja protsessi ning UPAAL, UPPAAL TRON ja DTRON tööriistu. Teine osa tööst kirjendab mudelipõhise hajustestimise kasutuselevõttu Tallinna infosüsteemi Teele tarkvaraarenduse projektis, mida varasemalt testiti manuaalselt. Tulemusena selgus, et Teele arendusmeeskond pigem ei ole mudelipõhise hajustestimise metoodikaga rahul mitte metoodika enda tõttu, vaid probleeme tekitas selle sobivus arendusprotsessi. Lisaks oli kasutatavate tööriistade kiire kasutusele võtmine keeruline, sest puudub toetav materjal ning testijad peavad arvestama ka tööriistade versioonide ühilduvusega.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 27 leheküljel, 4 peatükki, 18 joonist, 6 tabelit.

# List of abbreviations and terms

| | |
|---|---|
| FSA | finite state automata |
| MBT | model-based testing |
| SUT | system under test |
| TA | timed automata |
| TCTL | timed computation tree logic |

# Table of Contents

# List of figures

# List of tables

# 1 Introduction

Software testing is an important part of software development lifecycle mainly carried out in verification and validation phase. There is no universal method for testing because different methods are applied at different phases of development and have different objectives [1].

One of the testing methods is model-based testing (MBT) where tests are based on models of system under test (SUT) and its environment. Since the test cases are generated automatically based on models, it is said that model-based testing decreases manual effort and increases test coverage [2]. Common case is that the test models belong to the class of finite state automata (FSA) that are relevant for model state space exploration and for generating executable test sequences as a result of such exploration. Therefore, by automating model state space exploration strategies it is possible to achieve high test coverage than by manual testing. After the abstract test cases are generated based on the model, they are made executable on the system under test through test adapters, or alternatively, converted to test scripts executable directly in some continuous integration test environment [3].

Distributed testing is used to test software in the distributed test environment. Distributed testing simulates real-word user traffic and gives overview of how the SUT will act under test stimuli injected from (possibly geographically) different locations. Distributed testing generally assumes that parts of system under test interact with each other during a test run. This makes synchronization the most crucial part of distributed testing [4].

The goal of current thesis is to validate and evaluate the distributed model-based testing methodology suggested in [5] in real software development project. The project to be focused on is Tallinn City new information system called Teele. Teele is a web application developed to support the legislation processes of Tallinn City. In this system, legal acts can be created, processed and taken into force by the city government of council. There are many functionalities in Teele but to set focus of current thesis, only proceeding part of the system where the users can access the services remotely is taken under observation.

There are not many freeware tools to support distributed model-based testing. For modelling the system under test UPPAAL modelling tool is taken into use since it supports modelling parallel processes, timing constraints and has mature tool support for verification and testing [6]. For execution of the generated test cases UPPAAL TRON is used. This tool is based on UPPAAL model checking engine and is developed for black-box conformance testing of timed systems online [7]. For distributed test execution DTRON, the extension of UPPAAL TRON is used [8].

## 1.1 Related studies

This section gives an overview of existing studies related to distributed model-based testing and tools used in the current study.

Model-based testing has been widely studied field for years. There is a diverse selection of studies that cover different aspects of model-based testing. Distributed model-based testing has been studied to a smaller extent and the materials found have rather been about model-based remote testing of distributed systems than distributed testing itself.

The nature of model-based testing is explained in the book titled “Practical Model-Based Testing” by Mark Utting and Bruno Legeard [9]. This book gives a great overview of how model-based testing differs from other testing methods and how it can be part of typical software development lifecycle. Authors bring examples of how model-based testing is practiced but they do not cover MBT tools used in current study [9].

Another extensive overview of tools used in model-based testing is given in [10]. For example, fMBT developed by Intel uses models written in Python. Another popular model-based testing tool is Modbat which is based on extended finite state machines [10]. None of the tools introduced in [10] is not supporting distributed testing.

The UPPAAL modelling tool and modelling language are explained in paper entitled “A Tutorial on UPPAAL” which is written by Gerd Behrmann, Alexandre David and Kim G. Larsen. In addition to explaining the tool itself they are giving an overview of timed automata as well as modelling patterns and examples [11].

Another use case based study on using Uppaal tool family for MBT is presented in master thesis written in Tallinn University of Technology by Age Kruusamägi. It is titled "Model-based testing of distributed systems: Tallinn streetlight system case-study". This thesis describes model-based testing of distributed system, in contrast to distributed model-based testing on non-distributed system as current study. Still, the tools used are mostly the same. Author of the thesis finds that using model-based testing on large scale systems can be too time consuming because for every host there must be its own model. Author suggests using generalized modelling templates for modelling identically behaving parts of the system [3].

"Model Based Testing of Distributed Time Critical Systems" is a paper by Jüri Vain, Gert Kanter and Seshadhri Srinivasan. They are describing verification technique based on model checking and covering part of model-based testing workflow. They also suggest an algorithm to use decomposition on the model of system under test. This algorithm takes model of the system as an input and decomposes it into set of location specific test models that are attached to system ports. With this method they are suggesting reducing delays and timing issues in distributed testing [12].

As mentioned above, for distributed model-based test execution an extension DTRON of UPPAAL TRON has been developed. Paper titled "DTRON: a tool for distributed model-based testing of time critical applications" written by Aivo Ainer, Jüri Vain and Leonidas Tsiopoulos gives expert overview of the distributed model-based testing and the DTRON tool itself. It is built on UPPAAL and TRON tools which are briefly introduced as well [13].

## 1.2 Problem statement

The goal of current thesis is to apply the distributed model-based testing methodology introduced in [12] for a real software testing project and to evaluate its usability. During the research the answers to following questions will be searched for:

- Is distributed model-based testing methodology suitable for usage in projects like Teele?
- Under what conditions the MBT methodology used in the thesis has advantages over traditional manual testing?

- Are the tools used mature enough to run out current research?

To conduct the research following steps are covered:

- selecting the part of Teele web application relevant for validating distributed model-based testing method
- writing the specification of requirements of the selected part of SUT
- constructing the formal model of the requirements specifications
- verifying the correctness of test formation on the model
- generating the tests based on model
- implementing the test adapters
- running tests on system under test
- providing the test reports and analyzing the advantages and drawbacks of used methods

For validating the results different aspects can be evaluated. Test coverage can be compared to the coverage that system had before taking model-based testing into use. Human effort spent on different phases of testing can be evaluated. Number of found bugs and errors in the system can be analysed. General usability aspects [14] of testing web applications with the methodology of distributed model-based testing can be assessed.

## 1.3 Overview

Current thesis consists of four main chapters. Chapter 2 introduces distributed model-based testing theory, explains model-based testing process step-by-step and gives overview of suitable tools. Each Section of Chapter 2 represents one step of the model-based testing process. In Chapter 3 the testing method and tools introduced in Chapter 2 are applied in practical testing of Tallinn City system Teele. Detailed descriptions are available in the sections of Chapter 3. Chapter 4 analyses the results and evaluates usability. Chapter 5 draws conclusions and articulates suggestions for future work.

# 2 Distributed model-based testing

Model-based testing is a method where test cases are generated based on models of the system under test (SUT). Model-based testing is inherently black-box testing meaning that tests only control SUT inputs and can observe SUT outputs but do not have reference to its internal behaviour. The goal of black-box testing is to analyse system interfaces behaviour and the conformance to its requirements [12].

According to the statistical studies referred in [15] model-based testing finds the same amount or more faults in system than manual testing. Model-based testing is evaluated to be more cost effective, efficient, and saving testers' time and effort. To use model-based testing a tester must create and maintain the models and generate test cases from them. As shown in [16] MBT takes less time than manually designing and maintaining the test suite but the advantages of using MBT for complex systems may show up after few iterations in regression testing. Another benefit of model-based testing is improved testing quality as test case generator is based on algorithms and generation is systematic. Because of the same reason model-based testing increases test coverage because it is possible to generate more test cases and optimize them in terms of test sequence length and execution time [9].

Model-based testing process described in [9] is shown in the following figure.

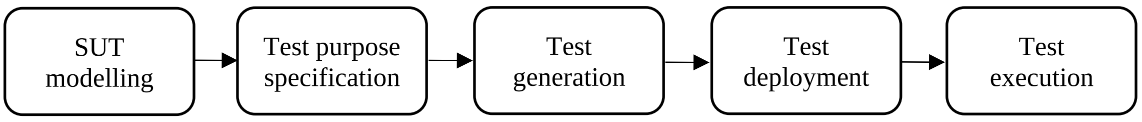


Figure 1. Model-based testing process

The process of model-based testing starts with creating the models of the SUT or part of the SUT that tests will focus on. Models are described as abstract timed automata. Secondly, test purpose needs to be specified that defines a finite set of executable test cases [5]. Next, test cases are generated from the model by considering the chosen test purpose specification. When test cases are generated test adapter should be written to run the tests on SUT. Test adapter is code for transforming symbolic inputs in the model to be executable by SUT inputs and the SUT outputs back to symbolic form. Based on the output received from SUT and the one expected by model the conformance relation is decided. Last step of the model-based testing process is to run the tests on SUT [9].

The described process can be applied also in distributed testing where tests use different ports of SUT during the same test run. For that the test model must be partitioned according to the distribution of SUT architecture and available test ports. Most critical factors about distributed testing are synchronization and timing of local test components. Therefore, it is necessary to add a test coordination mechanism and timing constraints to take into account delays between distributed SUT and tester components [13].

In the following subsections distributed model-based testing process is taken under more detailed observation and the tools supporting distributed model-based testing are introduced along the description of that process.

## 2.1 SUT modelling

SUT can be modelled using different modelling notations, for example state-transition-based notations which are the most commonly used for model-based testing. This class of models describes transitions between states of SUT. The examples of state transition models are UML state machines that are developed within finite state automata (FSA) theory. FSA notation uses graphs where nodes are representing states of the system and arcs are representing transitions between states [9].

Adding clock variables and data variables to finite state automata extends FSA to timed automata where clocks are used for mapping the model dynamics to time domain and specifying synchronisation constraints to concurrent events in the model. A tool to support described modelling notation is UPPAAL. UPPAAL is developed by Uppsala University and Aalborg University for verifying systems that are modelled as networks of timed automata. The model state can be extended with integer variables, structured data types and synchronisation channels [11]. An example of the parallel composition of two timed automata modelled with UPPAAL is given in Figure 2.

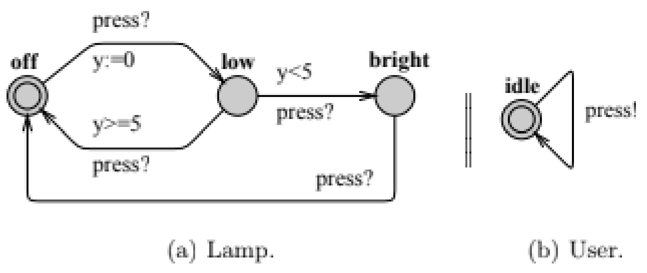


Figure 2. Example of timed automaton [11]

In UPPAAL graphical notation of the nodes are called locations. The edges between nodes represent discrete state transitions. The edges have four different types of attributes: in guard expression the conditions must be satisfied to make the edge enabled and in select expression comma separated list (name: type) of variables is defined that are updated non-deterministically with any value from given type. Synchronisation channels are for describing simultaneously executable edges. Channel labels have suffixes either “!” or “?” which denote synchronous output or input action, respectively. Finally, update expression is a list of comma separated assignment expressions to model the state transitions in terms of new values of variables and clocks [11].

## 2.2 Test purpose specification

Test purpose specification is second formal representation of test description along with modelling language introduced previously [17]. The test purpose specification must be written in machine readable form. In UPPAAL the property specification logic language TCTL (timed computation tree logic) allows specifying test scenarios in declarative form. Alternatively, adding auxiliary logic constraints in test model guard conditions and invariants allows restricting the model behaviours only to those that satisfy test purpose. While adding auxiliary constraints to the model enables both online and offline test generation, the diagnostic traces that are generated by TCTL model checking as symbolic test sequences can be generated only offline [11].

TCTL consists of path formulae and state formulae. State formulae describe the properties that can be interpreted only at model states and can be evaluated without looking at the behaviour of the model. It is also possible to express deadlock with state formulae using system predicate *deadlock*. If there is no enabled outgoing edge from the state, it is a deadlock state. Path formulae can be categorized into reachability, safety and liveness formulae [11]. This categorization is explained in Table 1.

Table 1. Path formulae in TCTL [11]

| Category | Concept | Formula | Explanation |
|---|---|---|---|
| Reachability | Possibly | E <> φ | Some state satisfying state formula φ should be reachable |
| Safety | Invariantly | A [] φ | State formula φ is true in all reachable states |
|  | Possibly invariantly | E [] φ | There exists an execution path where formulae φ is always true |
| Liveness | Eventually | A <> φ | State formula φ is eventually satisfied in all possible futures |

## 2.3 Test generation

In model-based testing, the test cases can be generated offline or online. In case of offline generation, test cases are generated before they will be executed. Online testing combines test generation and execution, i.e. test inputs are generated on-the-fly depending on the SUT current state and the test goal. Offline test generation does not pose constraints on test generation performance, but it may presume extensive model state space exploration that is typically needed in TCTL model checking. But its advantage is that when the SUT design or its requirements change it is possible to re-generate test cases by rerunning the model checker and there is no need for manual updating of test scripts.

Online testing has its own advantages, especially when testing systems, that have non-deterministic behaviour or the models of which are non-deterministic due to abstracting from some of SUT state variables. Online testing is also computationally cheaper because deciding on the next test input presumes exploration of much smaller portions (which are around current state) of the model state space [18]. As current thesis uses UPPAAL tool family, the test generation and execution tools UPPAAL TRON and DTRON are applied for online distributed model-based testing.

## 2.4 Test deployment

Before executing the test model in DTRON tool, DTRON needs to be configured according to the accessibility and the location of SUT test ports. Before elaborating the deployment process, the architecture and functioning principles of UPPAL TRON and DTRON are introduced.

UPPAAL TRON is test execution environment for UPPAAL models. UPPAAL TRON is suitable for black-box conformance testing of timed systems only locally [7]. DTRON is a tool for distributed testing real time systems online and it incorporates UPPAAL (possibly many instances of) TRON [8]. UPPAAL, UPPAAL TRON and DTRON relationships in testing process is shown in Figure 3 [19]. DTRON takes UPPAAL models as an input and uses Spread toolkit for messaging with DTRON API which is connected to SUT via its ports [13]. As only online testing method will be used henceforth, we defer from describing internal test generation mechanism by TCTL model checking in the rest of the thesis.

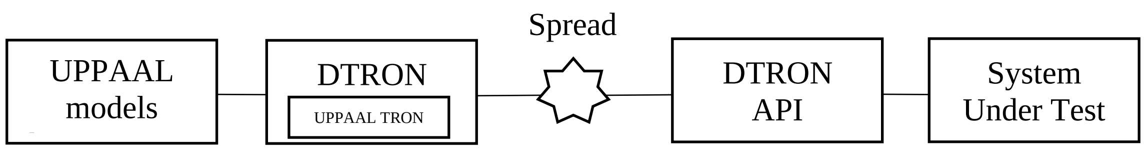


Figure 3. UPPAAL, UPPAAL TRON and DRON relationship

For test deployment the adapters are used between the DTRON API and SUT ports. An adapter is piece of code implemented for mapping model inputs to SUT and SUT outputs back to model interpretable symbolic form. UPPAAL TRON uses reporter in addition to adapters and assigns indexes instead of channel names to channels to optimize communication. This decreases the workload for test developer. DTRON has been developed to lessen the effort of building adapters and configuring the whole testing setup. DTRON adapters look models for channel names that have prefixes “i_” (input) or “o_” (output) whereas each channel pair needs separate adapter [13].

Since in distributed testing the timing is critical factor, additional delays may be introduced when processing the test inputs and returned outputs in adapters. To minimize these delays an adapter should be written rather simple. Otherwise delays may have impact on the test execution and tests may give false-negative results [13].

## 2.5 Test execution

Executing test cases with DTRON is performed using Spread message serialization. Distributed execution uses multiple ports to interact with SUT. Distributed execution configuration is shown schematically in Figure 4 [13]. Each port of SUT is connected to DTRON API which, in turn, is connected to Spread [13].

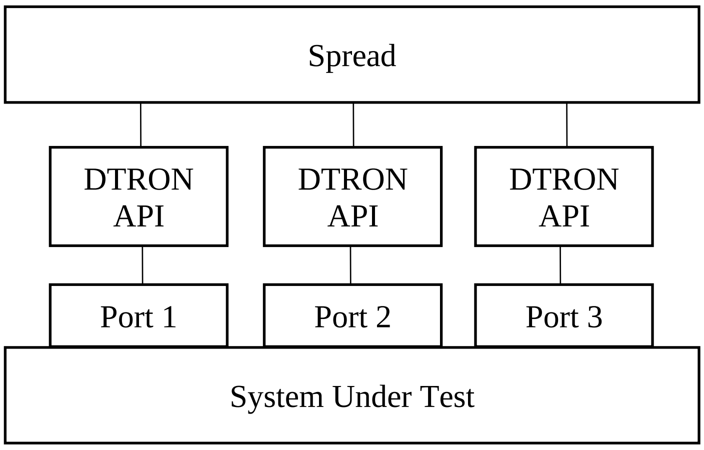


Figure 4. Distributed test execution architecture

As shown experimentally in [13] the reaction time DTRON needed for computing new test input after receiving an output from SUT is within the limits of 10 milliseconds that is sufficient regarding the performance requirements of testing non-hard real-time systems such as web based application Teele.

# 3 Tallinn city information system Teele

Teele is Tallinn City Council information system for processing legal acts throughout their lifecycle. Teele is used by Tallinn City Council, government and office, as well as in district councils, district administrations and departments. Teele was taken into use in January 2020 and its main goal is to optimize the processes of legislation, reduce bureaucracy and make city's administration transparent to citizens.

Teele is composed of different modules that are covering respective legislation processes. To set the focus for current thesis, only proceedings module of Teele will be considered as SUT. During the proceedings legal acts are getting approvals or disapprovals from various specialists from Tallinn city structure. To put a legal act into force it is mandatory to get its approvals from all specialists who are authorised to approve it.

## 3.1 Use cases

In this subsection the use cases of proceedings module will be introduced.

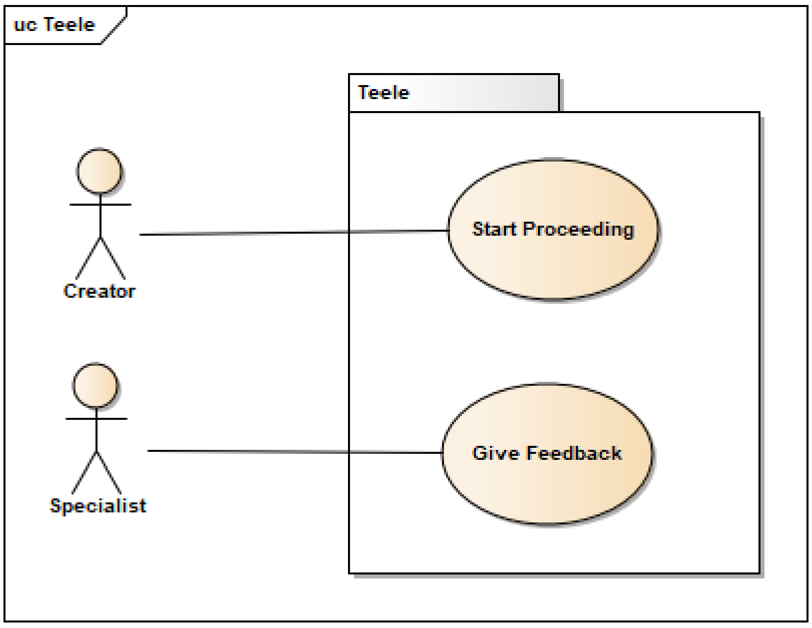


Figure 5. Use case diagram

Table 2. Use case - start proceeding

| | |
|---|---|
| Use case: | Start Proceeding |
| Description: | User has written new legal act and to set it into force user must start proceeding |
| Actors: | Creator |
| Preconditions: | User has written the legal act |
| Postconditions: | Proceeding is started |
| Flow: | 1. User clicks “Start proceeding”<br>2. System creates an active proceeding and notifies first specialist to give his/her feedback |

Table 3. Use case - give feedback

| | |
|---|---|
| Use case: | Give Feedback |
| Description: | User must give feedback to legal act in proceeding |
| Actors: | Specialist |
| Preconditions: | Proceeding has started, specialist is notified and must give feedback |
| Postconditions: | Feedback to legal act has been submitted |
| Flow: | 1. User attaches a comment and clicks “Approve”<br>2. System saves comment and approval and sends notification thereafter to next specialist to give feedback |
| Alternative flow: | *When user does not approve the legal act:*<br>a. User attaches comment and clicks “Disapprove”<br>*b.* System saves comment and disapproval and updates proceeding status to “ended”<br>*When Specialist was the last one to give the feedback:*<br>a. System saves comment and approval and updates proceeding status to “ended” |

## 3.2 SUT modelling

For modelling the part of Teele information system which use cases are described in 3.1 three models are needed. Each model describes the behaviour of one actor. Actors in current part of SUT are system, procedure creator and specialist. All named parties are also defined in UPPAAL model system declarations section as depicted in Figure 6. System declaration is used for defining the UPPAAL model configuration that consists of the parallel composition of automata template instantiations called processes [11]. In this example each actor constitutes one process.

```
system Creator, Specialist, System;
```

Figure 6. System declarations

UPPAAL also has global declarations for global variables, clocks, synchronisation channels and constants [11]. Global declarations used in the thesis are depicted in Figure 7.

```
const int n = 5;
typedef int[1,n+1] next_specialist = 1;
typedef int[1,n] specialistId_t;

clock gcl;

bool active = false;

chan i_start;
chan i_approve;
chan i_disapprove;
chan i_stop;
chan o_notify [n+1];
chan o_ended;

int i;
```

Figure 7. Global declarations

Constant *n* together with *next_specialist* and *specialistId_t* is used for defining the number and order of specialists who will participate in document proceeding. Global clock is defined as *gcl*. With Boolean *active* it is possible to describe status of the proceeding, for example if proceeding has started Boolean *active* will be equal to *true*. Synchronisation channels are following*: i_start; i_approve; i_disapprove; i_stop; o_notify[n+1]; o_ended.* Prefix "i_" determine inputs and prefix "o_" outputs. Integer *i* in global declarations is used for verification.

### 3.2.1 Model of the system under test

Tallinn city information system Teele is a web application. In Figure 8, the SUT is represented as UPPAAL TA model.

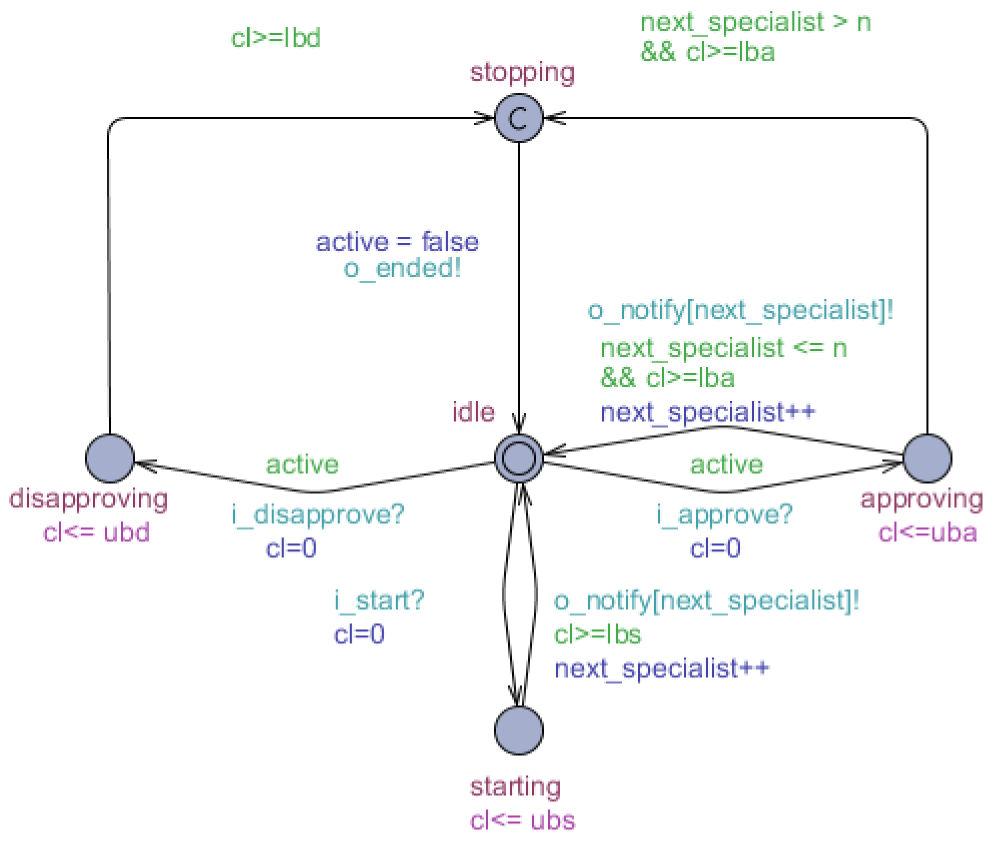


Figure 8. UPPAAL TA model of the system under test

SUT template has five locations: *idle*, *starting*, *approving*, *disapproving* and *stopping* which describe general modes of SUT operation. The SUT template uses the global constant *n* which denotes the number of specialists, global integer variable *next_specialist* and a global Boolean variable *active* which are self-explanatory. Local clock of SUT is defined in System local declarations section as *cl*. Similarly, the constants which determine the time it takes to perform different tasks are defined in local declarations section.

```
clock cl;

const int lbs=1, ubs=2;    //starting
const int lbt=1, ubt=2;    //stopping
const int lba=1, uba=2;    //approving
const int lbd=1, ubd=2;    //dissaproving
```

Figure 9. System local declarations

Initial location of SUT template is *idle*. SUT is in this location when it has not had any signal from creator of specialists. When SUT gets a signal from creator to start proceeding, SUT moves to *starting* location and sets local clock to zero. When SUT is in *starting* location, it is possible to go back to *idle* with sending notification to first specialist. When specialist gives feedback, he/she may approve or disapprove the document. When SUT gets the signal to approve, it moves to *approving* location where SUT needs to decide if there are more specialists to give feedback. If there is, SUT moves again to *idle* location and sends notification to following specialist. When current specialist was last to give feedback SUT moves to *stopping* location where the proceeding comes to end. If specialist decides to disapprove legal act, SUT moves to *disapproving* location and from there to *stopping* because proceeding cannot continue when someone has disapproved the legal act. From *ending* location signal *ended* is sent to creator to mark the end of the proceeding.

### 3.2.2 Model of the creator

Creators are officials of Tallinn city council or government whose tasks include creating new legal acts. When a new legal act is written it is creators' task to start proceeding and involve different specialists into the proceeding. The model template of the creator action is given in the Figure 10.

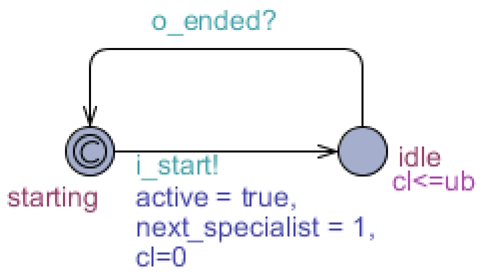


Figure 10. Model of the creator

Creator template has two locations: *starting* and *idle*. Initially creator is at *staring*, which is committed location. It means that *starting* is instantaneous internal action, i.e. it does not need triggering by external events [20]. When creator decides to give new document into proceeding, he/she sends *i_start* signal to SUT. After this action SUT will start the proceeding and creator will stay in *idle* location. Creator can move back to *starting* location when *o_ended* signal arrives from SUT. This marks the end of previous proceeding. Creator has local clock *cl* and local constants *lb* and *ub* which specify a non-deterministic interval [*lb*, *ub*] when new proceeding can be initiated.

```
clock cl;
const int lb=30, ub=60;
```

Figure 11. Creator local declarations

### 3.2.3 Model of specialist

Specialists are experts and leaders of different fields at Tallinn city structure. Specialists' role in legal act proceeding is to give feedback about the content of the act. Specialists receive notification when their opinion is needed. They can say whether they approve or disapprove the act. The model template of specialist behaviour is depicted in the Figure 12.

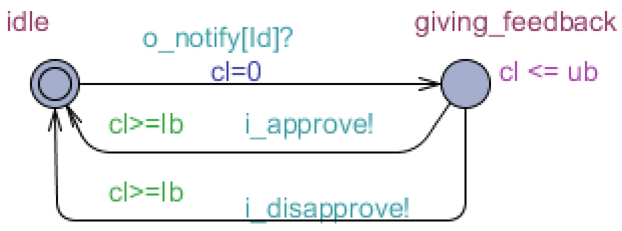


Figure 12. Model of the specialist

Specialist template has two locations: *idle* and *giving_feedback*. Initial location is *idle* where specialist is when there is no legal act to give feedback to. The control moves to *giving_feedback* location when notification is received. In this location specialist has two options – to *approve* or to *disapprove*. After making decision and sending corresponding signal to SUT, specialist is back in *idle* location. A local clock *cl* is defined in specialist local declarations shown in Figure 13. Constants for determining duration of *giving_feedback* are also defined in local declarations section.

```
clock cl;
const int lb=3, ub=5;
```

Figure 13. Specialist local declarations

## 3.3 Test purpose specification

To ensure the correctness of created test model UPPAAL has built-in verifier for checking whether the specification properties are satisfied in the model. System, Creator and Specialist templates modelled in previous section must satisfy the properties given in Table 4, Table 5 and Table 6.

Reachability properties are for verifying that it is possible to reach all locations of System, Creator and Specialist.

Table 4. Reachability properties

| **Property** | **Explanation** |
|---|---|
| `forall(i: specialistId_t) System.approving && next_specialist == i --> Specialist(i+1).giving_feedback` | When Specialist has approved the legal act, System sends notification to next specialist to give feedback |
| `System.disapproving --> Creator.idle and forall (id: specialistId_t) Specialist(id).idle` | When Specialist disapproved the legal act, System moves to *disapproving*, Creator and Specialists are in *idle* |

Safety properties are to verify that unwanted situations never occur. In addition, model deadlock freedom property is verified.

Table 5. Safety properties

| **Property** | **Explanation** |
|---|---|
| `A[] not deadlock` | Deadlock will never happen |
| `A[] not (System.stopping and forall (id : specialistId_t) Specialist(id).giving_feedback)` | Situation where System is in location *stopping* but Specialists is in *giving_feedback* will never happen |
| `A[] not (System.starting and forall (id : specialistId_t) Specialist(id).giving_feedback)` | Situation where System is in location *starting* but Specialists is in *giving_feedback* will never happen |
| `A[] System.starting imply Creator.idle and forall (id: specialistId_t) Specialist(id).idle` | System can be at *starting* location only when Creator and Specialist are in *idle* |
| `A[] System.starting imply next_specialist == 1` | When system is starting, *next_specialist* value will equal to one |

With liveness properties it is verified that System, Creator and Specialist will eventually be at location *idle*.

Table 6. Liveness properties

| **Property** | **Explanation** |
|---|---|
| `A<> System.idle` | Eventually System will be at location *idle* |
| `A<> Creator.idle` | Eventually Creator will be at location *idle* |
| `A<> forall (id: specialistId_t) Specialist(id).idle` | Eventually Specialists will be at location *idle* |

Result of verification is available in Appendix 1 – Results of Model Verification.

## 3.4 Test generation

As UPPAAL tool family uses online testing methods test cases will be generated during the execution and test generation step does not need any further action since all behaviours in the model are also tried by test execution tool. When the set of behaviours needs to be constrained, necessary updates can be made directly in the model.

## 3.5 Test deployment

Before the test execution, the extra time component introduced by test deployment on the test configuration must be added in the model and the extended model re-verified. UPPAAL TRON, which DTRON is built on, does not support channel arrays because it uses former version of UPPAAL modelling language [3]. In Paragraph 3.2 specialists are modelled in one model as they have identical behaviour. This simplifies verification of the models [3]. Synchronisation channel *notify* between SUT and specialists is same for all specialists but has different identifiers which makes the same effect as using the array of channels. To avoid channel arrays all participants must be modelled separately with individual channels as *notify1*, *notify2* etc.

The templates modelled using individual channels are available in Appendix 2 – Models with individual channels.

For test deployment DTRON adapters are written in Java. As the models use individual channels, adapters must cover every individual channel as well. As concluded in [3], for a large systems this may be too complicated and time consuming. In adapters DTRON channels are defined as shown in Figure 14.

```
IDtronChannel ch_start = new DtronChannel("start");
IDtronChannel ch_approve = new DtronChannel("approve");
IDtronChannel ch_disapprove = new DtronChannel("disapprove");
```

Figure 14. Creating channels in adapter

In addition, listeners for each channel must be written as well as their behaviour when message is received from channel. For example, listener for channel *ch_start* is shown in Figure 15. Full adapter code is available in Appendix 3 – Adapter.

```
getMBTDtron(0).addDtronListener(new DtronListenerExt(ch_start) {

	@Override
	public void messageReceived(IDtronChannelValued v) {
		boolean started = false;
		try {
			started = system.start(proceedingId, token);
		} catch (IOException e) {
			e.printStackTrace();
		}

		if(started) {
			String response = “notify1”;
			IDtronChannel reply = new DtronChannel(response);
			IDtronChannelValued valued =
			reply.constructValued((Map<String, Integer>) null);

			send(valued);
		}
	}
}
```

Figure 15. Channel listener

When a message is received in listener for channel *ch_start* method *start* is called in class *system*. In class *system* all logic is written to communicate with SUT. Code for method *start* in Java class *system* is presented in Figure 16. Full code for communicating with SUT is available in Appendix 3 – Adapter.

```
public boolean start (int proceedingId, String token) throws
IOException {
       int responseCode = 0;
       CloseableHttpClient httpclient = HttpClients.custom().build();

       try {
              HttpPatch request = new HttpPatch("https://teele-
dev.netgroupdigital.com/api/proceedings/" + proceedingId);
              StringEntity requestEntity =new
StringEntity("{\"statusCode\":\"INPROGRESS\"}");
              request.addHeader("content-type", "application/json");
              request.addHeader("Authorization", token);
              request.setEntity(requestEntity);
              HttpResponse response = httpclient.execute(request);

              responseCode = response.getCode();
              httpclient.close();
       } catch (Exception e) {
              System.err.print(e.getMessage() );
       }

       if (responseCode == 200) {
              return true;
       } else {
       return false;
       }
}
```

Figure 16. Communication between adapter and SUT

## 3.6 Test execution

To run the tests Spread Toolkit is used. Spread Toolkit is open source messaging service and it can be used for distributed test execution [21]. For execution Java main class is supplemented in the same Java project as adapters. In the executable code, shown in Figure 17, connection with DTRON and Spread Network is created. DTRON process is started giving Spread Network and UPPAAL models as input. Also, timeout in time units, time unit in milliseconds and verbosity are defined.

```
public static void main(String[] args) {
	MBTApplication app = new SampleAdapter();
	app.launchApp();
}

@Override
public boolean start() {
	boolean proceed = false;
	String testModel = "model/models.xml";
	SpreadNetwork sn = SpreadNetwork.create();

	if(sn != null && sn.start()) {
		List<SpreadNetwork> snl = new ArrayList<SpreadNetwork>();
		snl.add(sn);
		SampleTest vt = new SampleTest(snl);
		if(vt.start()) {
			DtronProcess dp = DtronProcess.create(sn, 
testModel, 100000, 600, VERBOSITY);
			if (dp != null) {
				dp.start();
				proceed = true;
			}
		} else {
			DtronAdapterHelper.Logger().warning("Adapter failed 
to start");
		}
	}
	return proceed;
}
```

Figure 17. Adapter execution code

Executing the Java project will give DTRON test result. Full result is available in Appendix 4 – DTRON test result.

# 4 Results and conclusions

In following sections test results are analysed and the usability of distributed model-based testing is evaluated.

## 4.1 Analysis of test results

Before introducing distributed model-based testing methodologies in project Teele, its testing was done manually. There were no test cases written because testers used exploratory approach. Exploratory testing is widely used in agile software projects where requirement documents are not complete. Testers have full control and their goal is to investigate the system, think and find bugs. Testing results depend on testers knowledge, intuition and understanding of the system [22]. Using exploratory testing approach, it is not easy to evaluate the exact test coverage. Although it is possible to analyse human effort and bugs found.

Human effort for manual testing in project Teele is high and it increases with every test run. Positive aspect is that manual testing does not need any preparation in MBT. When using automated tests preparation takes majority of time because tester must create a model, specify test cases and encode them either in the form of decidable TCTL constraints or model injected constraints before the test generation/execution can be run. Human effort is high at the beginning of the project, but afterwards testers only need to maintain and execute generated scripts or test model itself. Human effort spent on testing during the projects is stated in Figure 18. Graph shows how manual testing needs more human effort when the number of test iterations is high and, therefore, it is more suitable for short-time projects. Automated tests require more work in the beginning but in the long term they pay off. The point X is called breaking point as from that point forward automated tests have paid off [23]. During the current research project Teele did not reach the breaking point.

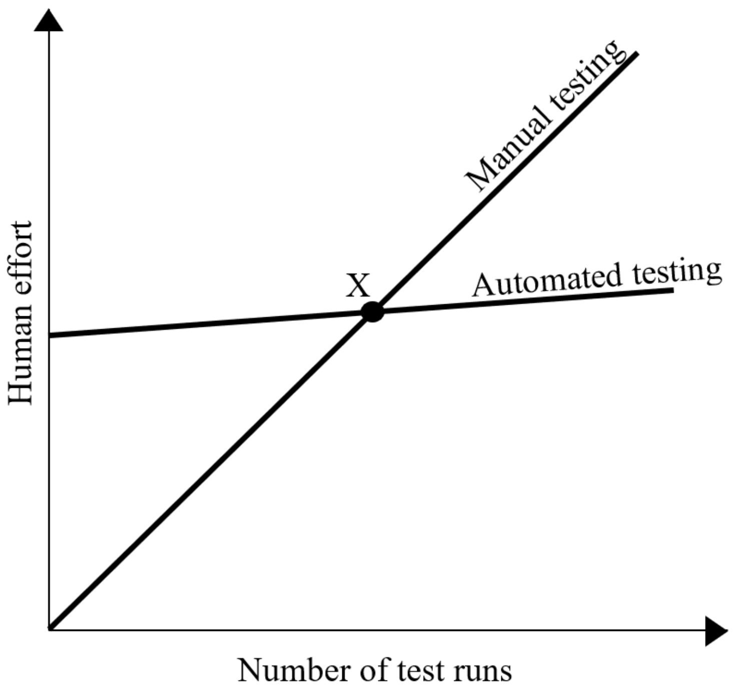


Figure 18. Human effort for manual testing and automated testing

Distributed model-based tests did not find any bugs from proceeding module during the research. One reason might have been that the code had been tested manually before MBT was applied to that. Manual exploratory testing found two of them:

- Proceeding step will not change color after specialist have given approval
- Specialists avatars are not loaded on the first load but on reload

Manually found bugs are rather about the design of user interface that automated tests are not expected to find. Comparing manually and automatically found bugs is discussed in article [24] written by James Bach. He states that bugs found by automated tests and manual tests are different and are not comparable. These two approaches have totally different processes as they reveal different kind of bugs. To achieve excellent testing results and high test coverage software testing strategy should include both of the methodologies [24].

Test coverage when using manual exploratory testing can vary. In project Teele testers try to cover all requirements and main scenarios but as manual testing is not documented it is not possible to evaluate and compare the real test coverage. As the goal of testers is to find bugs it may happen that knowing the system testers can come up with scenarios to reach bugs and achieving the goal, they leave other scenarios behind. Using distributed model-based testing all scenarios possible to reach in models are covered, test coverage is stable and high.

## 4.2 Usability evaluation

Usability is attribute of quality as is utility which asks the question “Whether the software provides the features users need?”. Usability itself defines how pleasant these features are to use. Usability can be evaluated by five components: learnability, efficiency, memorability, errors and satisfaction [14]. To evaluate usability of distributed model-based testing methodology and tools used in current study all usability components are analysed separately.

When taking into use new testing process and tools it is natural that in the beginning more time goes to learning. Usability learning component focuses on how easy it is for new users to use basic functionality for the first time [14]. Based on the current study it can be said that learning to use distributed model-based testing process is not complicated. Process consists of sequential logical steps that are easy to understand and there are supporting materials available. It is rather difficult to take into use UPPAAL and DTRON as the tools are not very commonly used and there are very few materials and examples to support the learning process. The most complex part of current study was to learn to model SUT with UPPAAL tool. UPPAAL is built with diverse functionalities that are hard to understand when not having exposure to modelling before. During the modelling tester also must bear in mind all necessary aspects for distributed execution later for example synchronisation clocks. Another rather complex part was executing the tests with DTRON. As DTRON uses Spread Toolkit and adapters to run it is difficult to learn how all the parties would communicate.

Efficiency can be evaluated after the learning process as it answers the question “How quickly can users perform tasks when they have completed the learning process?” [14]. Using distributed model-based testing tools later in the project is more efficient. Maintaining models in UPPAAL and creating new ones is easier and can be done quicker than in learning phase. Maintaining adapters and executing DTRON is also straightforward when there is previous experience. Distributed model-based testing process for project Teele is rather not efficient, not because of the testing process itself but its suitability into the development process. As the project documentation is not complete it is not possible to have use cases written before a part of development needs to be tested. Using model-based testing process testers have to write use cases based on developed features not documentation. Following the model-based testing process takes excessive part of testers time and the testing results are available later than expected.

Memorability of the process and tools has not been revealed as a problem because Teele is still partially in development phase and testers have to use the tools frequently to perform maintenance. Already completed project setup also contributes memorability which can be affected when starting new project from scratch.

Main error that happened in current research was when executing DTRON with models modelled with new version of UPPAAL modelling tool. DTRON is built on UPPAAL TRON which uses older version on UPPAAL and that caused lot of misunderstandings and extra effort to make models suitable for UPPAAL TRON. Errors in modelling may also happen when using UPPAAL itself but UPPAAL verifier probably assists to find the right solution.

The satisfaction of distributed model-based testing process and tools in web application development project Teele is rather low. As the creation of new model-based tests takes more time than manual testing and the tools used have insufficient or non-existent user interface it decreases the satisfaction. Testers find most bugs with manual exploratory testing and mainly these are the bugs that automated tests will never find like problems with user interface design. Distributed model-based testing would be suitable methodology to use in addition to manual testing. To avoid that preparation for model-based testing takes too long time and test results are available later than expected it would be necessary to have software documentation available before testing. Using correct development process this should not be problem. To achieve higher usability and especially shorter learning curve of tools used it may be essential to have more documentation and examples available for UPPAAL and DTRON. Also, the possibility to use DTRON with models modelled with newer version of UPPAAL would have positive impact to satisfaction.

# Summary

The goal of the current thesis was to validate and evaluate distributed model-based testing methodology in Tallinn City information system project Teele. When using model-based testing the test cases are generated automatically based on models and this can lead to high test coverage and low manual effort. The goal was to find out whether same results are possible to achieve in project Teele and are the tools UPPAAL, UPPAAL TRON and DTRON are mature enough for it.

First part of the thesis introduces model-based testing process and tools. Process starts with modelling the SUT or part of it with UPPAAL modelling tool. Models can be verified with UPPAAL verifier using TCTL language. Based on verified models it is possible to generate test cases but as execution tools UPPAAL TRON and DTRON are for online testing the test cases are generated during execution. Before execution test adapters must be written to map model inputs to SUT and SUT outputs back to models. Also Spread Toolkit must be configured for distributed execution.

Second part of the thesis uses previously described process and tools in Teele development project. Previously Teele was tested only manually using exploratory approach because documentation was not available and test results were needed fast after functionality was developed. Distributed model-based testing was taken into use for proceeding module of Teele where different specialists from city structure are giving their approval or disapproval to new legal acts before they are put into force.

As a result, Teele's project team was rather not satisfied with the new model-based testing process mainly because the test results are becoming available later than expected. Using model-based testing testers can start preparing the models and adapters after the functionality is developed because of unavailable requirements specification documentation. As a conclusion it can be said that for project Teele manual testing turned out to be more successful. Although this may not be the case with other projects where documentation is available and correct development process is followed.

Another usability aspect to appeal is learnability of the tools used. UPPAAL is complex tool with diverse functionalities that may be complex to understand, especially when there are not enough supporting materials and examples available. The same goes for execution tool DTRON which has to have connection with Spread Toolkit, adapters and models to communicate. To make initial setup for distributed model-based test execution a lot of time was committed. Also, testers have to keep in mind that UPPAAL TRON and DTRON are not supporting new version of UPPAAL modelling language, this makes modelling SUT more difficult and time consuming. Although in case where testers have already experience with distributed model-based testing and the tools the testing would take less effort and be more effective.

For future work distributed model-based testing can be taken into use in project where documentation is complete and correct development processes are followed so that new testing methodology will not bring any delays into development process. Also, part of SUT can be extended to cover more functionality and to analyse complexity of modelling when dimension of SUT is more significant.

# References

[1] M. Kääramees, “A Symbolic Approach to Model-based Online Testing,” Tallinn University of Technology, Tallinn, 2012.

[2] R. V. Binder, B. Legeard and A. Kramer, “Model-Based Testing: Where Does It Stand?,” *ACM Queue,* vol. 13, no. 1, 2015.

[3] A. Kruusamägi, “Model-based Testing of Distributed Systems: Tallinn Streetlight System Case-Study,” Tallinn University of Technology, Tallinn, 2016.

[4] X. Liu, Y.-J. Hsieh, R. Chen and S.-M. Yuan, “Distributed Testing System for Web Service Based on Crowdsourcing,” *Hindawi,* vol. 2018, p. 15, 2018.

[5] K. Saarna, “Aspect-Oriented Model-Based Testing,” Tallinn University of Technology, Tallinn, 2018.

[6] “UPPAAL,” [Online]. Available: http://www.uppaal.org. [Accessed 28 03 2020].

[7] “UPPAAL TRON,” [Online]. Available: https://people.cs.aau.dk/~marius/tron/. [Accessed 28 03 2020].

[8] “DTRON,” [Online]. Available: https://cs.ttu.ee/dtron. [Accessed 28 03 2020].

[9] M. Utting and B. Legeard, “Practical Model-Based Testing,” Morgan Kraufmann, 2007.

[10] “Model Based Testing,” ProfessionalQA, 02 03 2020. [Online]. Available: https://www.professionalqa.com/model-based-testing-tools. [Accessed 28 03 2020].

[11] G. Behrmann, A. David and K. G. Larsen, “A Tutorial on UPPAAL 4.0,” Aalborg University, Denmark, 2006.

[12] J. Vain, G. Kanter and S. Srinivasan, “Model Based Testing of Distributed Time Critical Systems,” in *6th International Conference on Reliability,* India, 2017.

[13] A. Anier, J. Vain and L. Tsiopoulos, “DTRON: A Tool for Distributed Model-Based Testing of Time Critical Applications,” *Proceedings of Estonian Academy of Sciences,* vol. 1, no. 66, pp. 75-88, 2017.

[14] J. Nielsen, “Usability 101: Introduction to Usability,” Nielsen Norman Group, 03 01 2012. [Online]. Available: https://www.nngroup.com/articles/usability-101-introduction-to-usability/. [Accessed 28 03 2020].

[15] J. Zander, I. Schieferdecker and P. J. Mosterman, Model-Based Testing for Embedded Systems, CRC Press, 2017.

[16] M. Felderer and A. Beer, “Estimating the Return on Investment of Defect Taxonomy Supported System Testing in Industrial Projects,” in *38th Euromicro Conference on Software Engineering and Advanced Applications*, 2012.

[17] E. Halling, J. Vain, A. Boyarchuk and O. Illiashenko, “Test Scenario Specification Language for Model-Based Testing,” *International Journal of Computing,* vol. 4, no. 18, pp. 408-421, 2019.

[18] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson and A. Skou, “Testing Real-Time Systems Using UPPAAL,” Lecture Notes in Computer Science 4949, 2008.

[19] A. Anier, “DTRON Tutorial,” [Online]. Available: https://cs.ttu.ee/dtron/dtronTutorial.pdf. [Accessed 28 03 2020].

[20] A. Anier, “Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems,” Tallinn University of Technology, Tallinn, 2016.

[21] “Spread Toolkit,” [Online]. Available: http://www.spread.org/index.html. [Accessed 28 03 2020].

[22] “What is Exploratory Testing?,” Guru99, [Online]. Available: https://www.guru99.com/exploratory-testing.html. [Accessed 15 04 2020].

[23] R. Ramler and K. Wolfmaier, “Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost,” Software Competence Center Hagenberg GmbH, Austria, 2006.

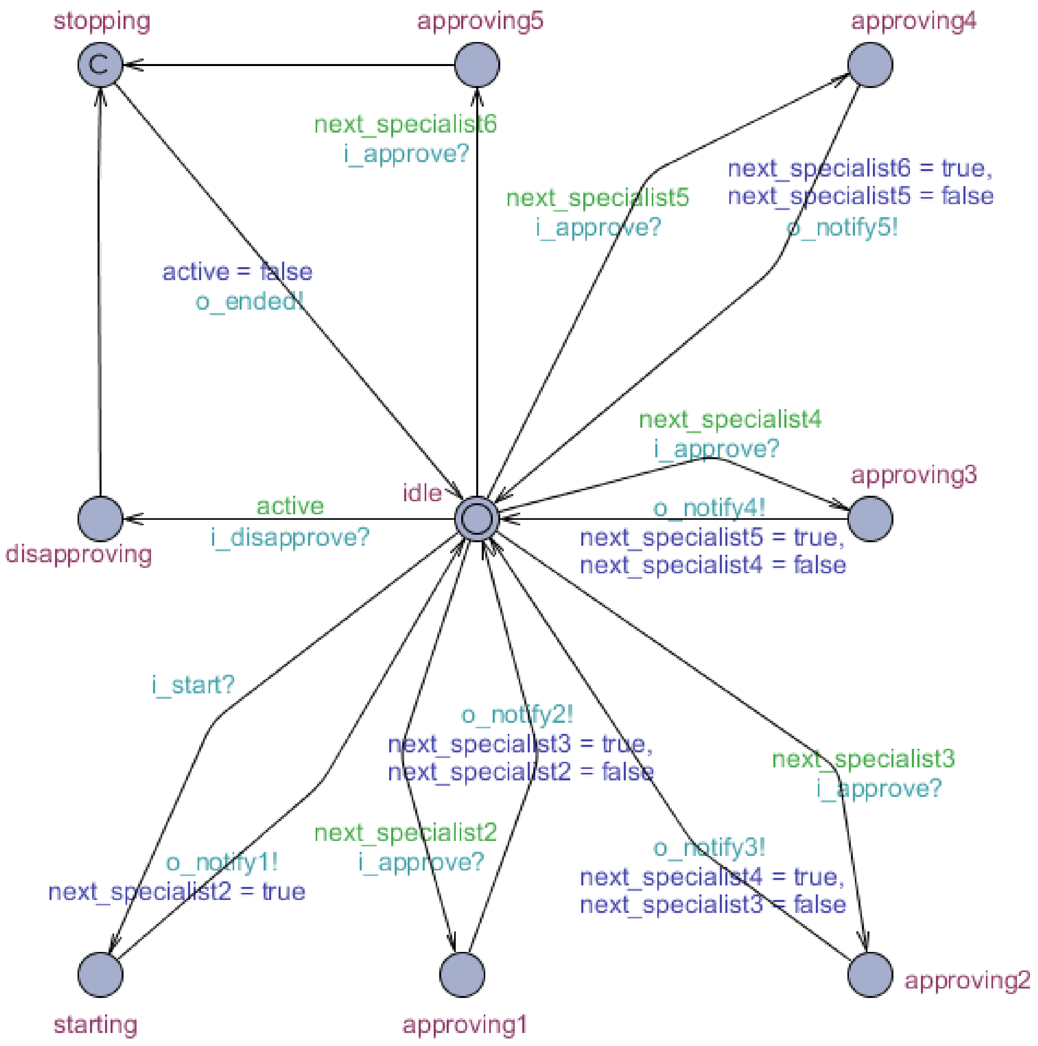
[24] J. Bach, “Test Automation Snake Oil,” 1999.

# Appendix 1 – Results of Model Verification

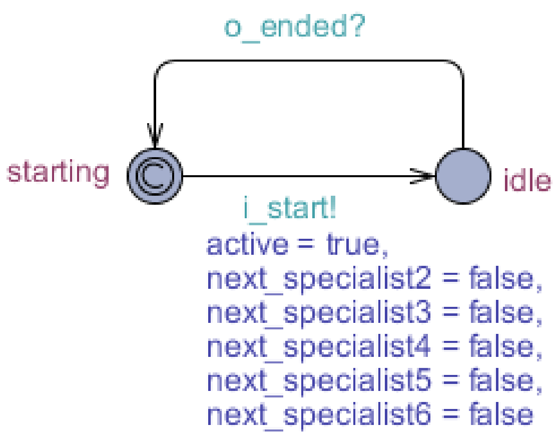
```
forall(i: specialistId_t) System.approving && next_specialist == i -->
Specialist(i+1).giving_feedback
Property is satisfied.
System.disapproving --> Creator.idle and forall (id: specialistId_t)
Specialist(id).idle
Property is satisfied.
A[] System.starting imply next_specialist == 1
Property is satisfied.
A[] not deadlock
Property is satisfied.
A[] not (System.starting and forall (id : specialistId_t)
Specialist(id).giving_feedback)
Property is satisfied.
A[] not (System.stopping and forall (id : specialistId_t)
Specialist(id).giving_feedback)
Property is satisfied.
A[] System.starting imply Creator.idle and forall (id: specialistId_t)
Specialist(id).idle
Property is satisfied.
A<> forall (id : specialistId_t) Specialist(id).idle
Property is satisfied.
A<> Creator.idle
Property is satisfied.
A<> System.idle
Property is satisfied.
```

# Appendix 2 – Models with individual channels

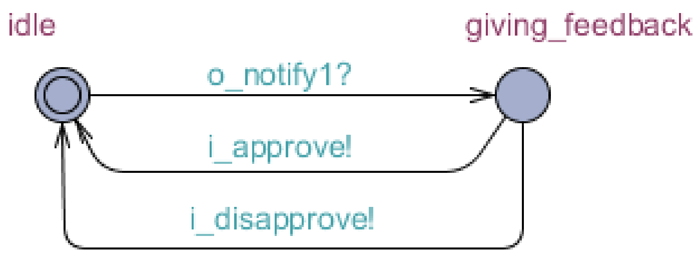
Model of system:



Model of creator:


o_ended?
starting
idle
i_start!
active = true,
next_specialist2 = false,
next_specialist3 = false,
next_specialist4 = false,
next_specialist5 = false,
next_specialist6 = false

Model of specialist1:



# Appendix 3 – Adapter

```
public class SampleTest extends DtronAdapter {
	public SampleTest(List<SpreadNetwork> snlist) {
		super(snlist);
	}

	private int nextSpecialist;
	private int proceedingId = 148 //insert next proceedingId here;
	private int stepId = 312 //insert next stepId here;
	private String token = "" //insert token here;

	public void addListeners() throws SpreadException {
		Sut system = new Sut();

		IDtronChannel ch_start = new DtronChannel("start");
		IDtronChannel ch_approve = new DtronChannel("approve");
		IDtronChannel ch_disapprove = new DtronChannel("disapprove");

		getMBTDtron(0).addDtronListener(new DtronListenerExt(ch_start) {
			@Override
			public void messageReceived(IDtronChannelValued v) {
				boolean started = false;
				try {
					started = system.start(proceedingId, token);
				} catch (IOException e) {
					e.printStackTrace();
				}
				if(started) {
					String response = “notify1”;
					IDtronChannel reply = new 
DtronChannel(response);
					IDtronChannelValued valued = 
						reply.constructValued((Map<String, 
Integer>) null);
					send(valued);
				}
			}
		});

		getMBTDtron(0).addDtronListener(new 
DtronListenerExt(ch_approve){
			@Override
			public void messageReceived(IDtronChannelValued v) {
				boolean approved;
				if (getNextSpecialist() == 5) {
```

```
                                    approved = system.disapprove(proceedingId,
stepId, token);
                              } else {
                                    approved = system.approve(proceedingId,
stepId, token);
                              }
                              if(approved) {
                                    String response = null;
                                    int s = getNextSpecialist();

                                    if (s == 0) {
                                          response = "notify2";
                                          setNextSpecialist(2);
                                          setStepId(getStepId() + 1);
                                    } else if (s == 2) {
                                          response = "notify3";
                                          setNextSpecialist(3);
                                          setStepId(getStepId() + 1);
                                    } else if (s == 3) {
                                          response = "notify4";
                                          setNextSpecialist(4);
                                          setStepId(getStepId() + 1);
                                    } else if (s == 4) {
                                          response = "notify5";
                                          setNextSpecialist(5);
                                          setStepId(getStepId() + 1);
                                    } else {
                                          response = "ended";
                                          setNextSpecialist(0);
                                          setProceedingId(getProceedingId()+ 1);
                                          setStepId(getStepId() + 1);
                                    }
                                    IDtronChannel reply = new
DtronChannel(response);
                                    IDtronChannelValued valued =
                                          reply.constructValued((Map<String,
Integer>) null);
                                    send(valued);
                              }
                        }
                  });

                  getMBTDtron(0).addDtronListener(new
DtronListenerExt(ch_disapprove) {
                        @Override
                        public void messageReceived(IDtronChannelValued v) {
                              boolean disapproved =
system.disapprove(proceedingId, stepId, token);
                              if(disapproved) {
                                    String response = "ended";
                                    if (getNextSpecialist() == 2) {
                                          setStepId(getStepId() + 4);
                                    } else if (getNextSpecialist() == 3) {
                                          setStepId(getStepId() + 3);
                                    } else if (getNextSpecialist() == 4) {
                                          setStepId(getStepId() + 2);
                                    } else if (getNextSpecialist() == 5) {
                                          setStepId(getStepId() + 1);
                                    } else if (getNextSpecialist() == 0){
                                          setStepId(getStepId() + 5);
                                    }
                                    setNextSpecialist(0);
                                    setProceedingId(getProceedingId() + 1);
                                    IDtronChannel reply = new
DtronChannel(resp);
                                    IDtronChannelValued valued =
                                          reply.constructValued((Map<String,
Integer>) null);
                                    send(valued);
                              }
                        }
                  });
            }

            @Override
            protected int getDtronID(SpreadNetwork sn) {
                  return 0;
            }

            @Override
            protected void cleanUpAdapter() {

            }

            public int getNextSpecialist() {
                  return nextSpecialist;
            }

            public void setNextSpecialist(int nextSpecialist) {
                  this.nextSpecialist = nextSpecialist;
            }

            public int getProceedingId() {
                  return proceedingId;
            }

            public void setProceedingId(int proceedingId) {
                  this.proceedingId = proceedingId;
            }

            public int getStepId() {
                  return stepId;
```

```
	}


	public void setStepId(int stepId) {
		this.stepId = stepId;
	}
}

public class Sut {

	public boolean start(int proceedingId, String token) throws
IOException {
		int responseCode = 0;
		CloseableHttpClient httpclient = HttpClients.custom().build();
		try {
		HttpPatch request = new HttpPatch("https://teele-
dev.netgroupdigital.com/api/proceedings/" + proceedingId);
		StringEntity reqEntity =new
StringEntity("{\"statusCode\":\"INPROGRESS\"}");
		request.addHeader("content-type", "application/json");
		request.addHeader("Authorization", token);
		request.setEntity(reqEntity);
		HttpResponse response = httpclient.execute(request);
		responseCode = response.getCode();
		httpclient.close();
		} catch (Exception e) {
		System.err.print(e.getMessage() );
		}
		if (responseCode == 200) {
		return true;
	} else {
		return false;
	}
	}

	public boolean approve(int proceedingId, int stepId, String token) {
		int responseCode = 0;
		CloseableHttpClient httpclient = HttpClients.custom().build();
		try {
		HttpPatch request = new HttpPatch("https://teele-
dev.netgroupdigital.com/api/proceedings/" +proceedingId+ "/steps/" +stepId+
"/responses/pending");
		StringEntity reqEntity =new
StringEntity("{\"status\":\"ACCEPTED\", \"comment\":\"text\"}");
		request.addHeader("content-type", "application/json");
		request.addHeader("Authorization", token);
		request.setEntity(reqEntity);
		HttpResponse response = httpclient.execute(request);
		responseCode = response.getCode();
		httpclient.close();
		} catch (Exception e) {
		System.err.print(e.getMessage() );
```

```
            }
            if (responseCode == 200) {
            return true;
      } else {
            return false;
      }
      }

      public boolean disapprove(int proceedingId, int stepId, String token){
            int responseCode = 0;
            CloseableHttpClient httpclient = HttpClients.custom().build();
            try {
            HttpPatch request = new HttpPatch("https://teele-
dev.netgroupdigital.com/api/proceedings/" +proceedingId+ "/steps/" +stepId+
"/responses/pending");
            StringEntity reqEntity =new
StringEntity("{\"status\":\"REJECTED\", \"comment\":\"text\"}");
            request.addHeader("content-type", "application/json");
            request.addHeader("Authorization", token);
            request.setEntity(reqEntity);
            HttpResponse response = httpclient.execute(request);

                  HttpPatch request2 = new HttpPatch("https://teele-
            dev.netgroupdigital.com/api/proceedings/" +proceedingId+
            "/restart");
            request2.addHeader("content-type", "application/json");
            request2.addHeader("Authorization", token);
            HttpResponse response2 = httpclient.execute(request2);
            responseCode = response2.getCode();
            httpclient.close();
            } catch (Exception e) {
            System.err.print(e.getMessage() );
            }
            if (responseCode == 200) {
            return true;
      } else {
            return false;
      }
      }
}
```

# Appendix 4 – DTRON test result

```
==========================Dtron-3899 output:START===========================
INFO e.t.c.d.t.TronInstaller: Checking for TRON from environment variable -
TRON_HOME
INFO e.t.c.d.t.TronInstaller: Running on Windows, checking for "tron.exe"
INFO e.t.c.d.t.TronInstaller: Found tron:
C:\DistributedModelBasedTesting\tron\tron.exe
INFO e.t.c.d.a.s.DtronUpta:
tronexe=C:\DistributedModelBasedTesting\tron\tron.exe
WARN b.c: Constructing listeners with c
INFO e.t.c.d.c.a: Configured with timeout 600 and timeunit 100000
INFO o.a.c.v.i.StandardFileSystemManager: Using
"C:\Users\KAIRIT~1.SIM\AppData\Local\Temp\vfs_cache" as temporary files
store.
INFO e.t.c.a.AntlrXta: Found channel - i_start
INFO e.t.c.a.AntlrXta: Found channel - i_approve
INFO e.t.c.a.AntlrXta: Found channel - i_disapprove
INFO e.t.c.a.AntlrXta: Found channel - i_stop
INFO e.t.c.a.AntlrXta: Found channel - o_notify1
INFO e.t.c.a.AntlrXta: Found channel - o_notify2
INFO e.t.c.a.AntlrXta: Found channel - o_notify3
INFO e.t.c.a.AntlrXta: Found channel - o_notify4
INFO e.t.c.a.AntlrXta: Found channel - o_notify5
INFO e.t.c.a.AntlrXta: Found channel - o_ended
INFO e.t.c.d.c.d: Found incoming channel: i_start
INFO e.t.c.d.c.d: Found incoming channel: i_approve
INFO e.t.c.d.c.d: Found incoming channel: i_disapprove
INFO e.t.c.d.c.d: Found incoming channel: i_stop
INFO e.t.c.d.c.d: Found outgoing channel: o_notify1
INFO e.t.c.d.c.d: Found outgoing channel: o_notify2
INFO e.t.c.d.c.d: Found outgoing channel: o_notify3
INFO e.t.c.d.c.d: Found outgoing channel: o_notify4
INFO e.t.c.d.c.d: Found outgoing channel: o_notify5
INFO e.t.c.d.c.d: Found outgoing channel: o_ended
INFO e.t.c.d.a.s.Dtron: Connecting to Spread at localhost:3899 (AXFqxR7V)
WARN e.t.c.d.a.s.Dtron: Connected to Spread at localhost:3899
INFO e.t.c.d.a.s.Dtron: Don't forget to clean up and disconnect()!
INFO e.t.c.d.a.s.DtronUpta: Going to execute -
[C:\DistributedModelBasedTesting\tron\tron.exe, -P, eager, -v, 9, -I,
SocketAdapter, C:\DistributedModelBasedTesting\model\models1.xml, --,
localhost, 6236]
UPPAAL TRON 1.5 using UPPAAL 4.1.2 (rev. 4351), June 2009
Compiled with i586-mingw32msvc-g++ -Wall -DLIBXML_STATIC -DNDEBUG -O2 -
ffloat-store -march=pentiumpro -march=pentium4 -march=prescott -
march=pentium-m -DTIGA_MERGE_STATES -DBOOST_DISABLE_THREADS
Copyright (c) 1995 - 2009, Uppsala University and Aalborg University.
```

```
All rights reserved.
Options for UPPAAL TRON:
 Search order is breadth first
 Using no space optimisation
 State space representation uses minimal constraint systems
 Observation uncertainties: 0, 0, 0, 0 (microseconds).
 Scheduling latency: 0 microseconds
 Future precomputation: closure(0 mtu).
 Input delay extended by: 0
 OS scheduler: non-real-time.
INFO b.a: Setting timeout to 600, timeunit to 100000
INFO e.t.c.d.a.s.Dtron: Joining group notify1
INFO e.t.c.d.a.s.Dtron: Joining group notify2
INFO e.t.c.d.a.s.Dtron: Joining group notify3
INFO e.t.c.d.a.s.Dtron: Joining group notify4
INFO e.t.c.d.a.s.Dtron: Joining group notify5
INFO e.t.c.d.a.s.Dtron: Joining group ended
 Emulation invariants: Creator, Specialist1, Specialist2, Specialist3,
Specialist4, Specialist5.
 Timeunit: 100000us
 Timeout: 600mtu
 Inputs: i_start(), i_approve(), i_disapprove(), i_stop()
 Outputs: o_notify1(), o_notify2(), o_notify3(), o_notify4(), o_notify5(),
o_ended()
TEST in progress | 0%INFO e.t.c.d.a.s.Dtron: Spreading message: name=start,
variables=null
TEST in progress / 1%
TEST in progress - 3%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress \ 4%INFO b.e: Received - name=notify2, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress | 5%INFO b.e: Received - name=notify3, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress / 7%INFO b.e: Received - name=notify4, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
INFO b.e: Received - name=notify5, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress - 8%INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve,
variables=null
TEST in progress \ 11%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
```

```
TEST in progress | 12%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress - 15%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress \ 15%INFO e.t.c.d.a.s.Dtron: Spreading message: name=start,
variables=null
TEST in progress | 17%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress | 20%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress / 21%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress - 22%INFO e.t.c.d.a.s.Dtron: Spreading message:
name=approve, variables=null
INFO b.e: Received - name=notify2, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress \ 23%INFO b.e: Received - name=notify3, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress | 25%INFO b.e: Received - name=notify4, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress \ 27%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress | 28%
TEST in progress / 29%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress \ 32%INFO b.e: Received - name=ended, variables=null,
timestamp=0
TEST in progress | 32%INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress / 33%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress - 35%INFO b.e: Received - name=notify2, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress / 36%DRIVER: 1586634257.357105s has passed, now it's
1586634257.358102s
TEST in progress / 38%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress - 39%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress \ 40%INFO e.t.c.d.a.s.Dtron: Spreading message:
name=approve, variables=null
INFO b.e: Received - name=notify2, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress \ 43%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress | 44%
TEST in progress / 46%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
INFO b.e: Received - name=notify2, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress - 47%INFO b.e: Received - name=notify3, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress \ 48%INFO b.e: Received - name=notify4, variables=null,
timestamp=0
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
INFO b.e: Reported to UPTA!
TEST in progress - 51%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress \ 52%
TEST in progress | 53%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
INFO b.e: Received - name=notify2, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress / 54%INFO e.t.c.d.a.s.Dtron: Spreading message:
name=approve, variables=null
INFO b.e: Received - name=notify3, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
```

```
TEST in progress \ 58%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress | 58%INFO e.t.c.d.a.s.Dtron: Spreading message: name=start,
variables=null
TEST in progress / 59%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress - 61%INFO b.e: Received - name=notify2, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress / 63%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress - 63%
TEST in progress \ 65%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress | 66%INFO b.e: Received - name=notify2, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
INFO b.e: Received - name=notify3, variables=null, timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress / 67%INFO e.t.c.d.a.s.Dtron: Spreading message:
name=disapprove, variables=null
TEST in progress | 70%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress / 71%
TEST in progress - 72%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress | 75%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress / 75%
TEST in progress - 77%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress \ 80%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress | 80%INFO e.t.c.d.a.s.Dtron: Spreading message: name=start,
variables=null
```

```
TEST in progress / 81%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress - 82%INFO b.e: Received - name=notify2, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress \ 86%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress | 87%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress / 89%INFO b.e: Received - name=notify2, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=approve, variables=null
TEST in progress - 90%INFO b.e: Received - name=notify3, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress | 93%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=start, variables=null
TEST in progress / 93%
TEST in progress - 94%INFO b.e: Received - name=notify1, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
INFO e.t.c.d.a.s.Dtron: Spreading message: name=disapprove, variables=null
TEST in progress | 98%INFO b.e: Received - name=ended, variables=null,
timestamp=0
INFO b.e: Reported to UPTA!
TEST in progress / 98%INFO e.t.c.d.a.s.Dtron: Spreading message: name=start,
variables=null
TEST in progress - 99%
TEST PASSED: Time out for testing
TR.Receiver: java.io.EOFException
reporter self-shutdown! (tester timed out and disconnected?)
CONN::readchannel: A blocking operation was interrupted by a call to
WSACancelBlockingCall.
WARN e.t.c.d.a.s.DtronUpta: Reporter server-socket close()
INFO e.t.c.d.a.s.Dtron: Leaving 6 groups ...
INFO e.t.c.d.a.s.Dtron: Un-registering 6 listeners ...
LISTENER: told to exit so returning
INFO e.t.c.d.a.s.Dtron: Disconnecting from Spread at localhost:3899
WARN e.t.c.d.a.s.Dtron: Disconnected from Spread at localhost:3899
===========================Dtron-3899 output: END===========================
```