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

Henri Jakobson 193321IAIB

# OPTIMISATION AND IMPLEMENTATION OF DISTRIBUTED TESTER GENERATION ALGORITHM

Bachelor's Thesis

**Supervisor**
Jüri Vain
PhD

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Tarkvarateaduse instituut

Henri Jakobson 193321IAIB

# HAJUSTESTRI GENEREERIMISALGORITMI OPTIMEERIMINE JA IMPLEMENTEERIMINE

Bakalaureusetöö

**Juhendaja**

Jüri Vain

PhD

Tallinn 2022

# 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: | Henri Jakobson | ..................................... |
| | | (signature) |
| Date: | 29.05.2022 | |

# Annotatsioon

Hajutatud reaalaja süsteemidel on sageli ranged ajastuspiirangud, mis nõuavad süsteemide testimisel ajastusega seotud probleemide lahendamiseks skaleeruvaid lähenemisviise. Kasutades täielikult hajutatud testrit lokaalsete testrikomponentidega, mis on ühendatud otse testitava süsteemi portidega, suudame ületada hilistumisprobleemid, mis on seotud testri ja testitava süsteemi vahelise sideviivitusega. Varem loodud algoritm, mille eesmärk on genereerida täielikult hajutatud tester tsentraliseeritud kaugtestrist, loob lokaaltestrite omavahelise sünkroniseerituse tagamiseks liiasusi, mida on võimalik vältida kommunikatsioonimahu optimeerimisega. Struktuursete ja kommunikatsioonis leiduvate liiasuste tagajärjena on testijadad pikemad, testimise aeg pikem ning avaldatakse suuremat koormust kommunikatsioonikanalitele.

Antud lõputöö peamine eesmärk on hajustestri genereerimise algoritmi optimeerimine, mille tulemuseks on optimeeritud sünkroniseerimist vajavate hajustestri lokaalkomponentide hulk. Optimeerimine viiakse läbi testimudeli põhjuslike liiasuste tuvastamise ja vähendamisega, mis põhineb tsentraliseeritud testris esinevate kommunikatsioonistruktuuride vaatluste analüüsil. Käesolevas töös loodud algoritmi õigsuse verifitseerimiseks kontrollisime bisimulatsiooni ekvivalentsi seost testiportide sündmuste suhtes tsentraliseeritud testri ja optimeeritud hajutatud testri vahel.

Optimeerimise tulemused on esitatud näite põhiselt. Optimeerimise tulemused näitasid üle 28% testimudeli struktuurset vähenemist, üle 23% testjälje pikkuse vähenemist, üle 26% keskmise ajakulu vähenemist ja üle 38% sünkroonimissõnumite arvu vähenemist.

Autorile teadaolevalt ei ole antud testri hajutusalgoritmi implementeeritud tarkvaraliselt. Seetõttu on töös tarkvaraliselt realiseeritud algoritmid meetodi üldkontseptsiooni ja praktilise rakendatavuse tõestuseks. Optimeeritud hajustestri genereerimise algoritm ja algne algoritm on implementeeritud ja integreeritud TalTech veebipõhisesse modelleerimis- ja testimiskeskkonda "BugBroom".

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 49 leheküljel, 7 peatükki, 33 joonist, 1 tabelit.

# Abstract

Real-time distributed systems often have strict timing constraints that require scalable approaches to overcome timing-related issues in testing. By using a fully distributed tester with local testers attached to the ports of the system under test, we are able to overcome delay issues related to communication delays between the tester and the system under test. The previously proposed algorithm to generate a fully distributed tester from a centralised remote tester creates overhead to ensure the synchronisation of local testers, which can be avoided by optimising the communication overhead. Structural and communication overhead results in longer test sequences, longer test time and more workload on the communication channels.

The main goal of this thesis is to optimise the distributed tester generation algorithm to reduce the communication overhead. The optimisation is conducted by detecting and reducing causal redundancies in the test model based on the analysis of observations of communication structures present in the centralised tester. To verify the correctness of the optimised algorithm, we verified the bisimulation equivalence relation between the centralised tester and optimised distributed tester with respect to the observable input/output actions on the system under test ports.

The optimisation results are presented based on an example. The optimisation results showed over $28\%$ of decrease in structural complexity of the test model, over $23\%$ of decrease in trace length, over $26\%$ of decrease in average time spent, and over $38\%$ of decrease in total synchronisation message count.

There are no implementations of the given tester distribution algorithm to the best of the author's knowledge. Therefore, the goals are to implement the algorithms as a proof of concept and integrate the implementation into the web-based modelling and testing environment "BugBroom" in TalTech.

This thesis is written in English and is 49 pages long, including 7 chapters, 33 figures, 1 table.

# List of abbreviations and terms

| | |
|---|---|
| API | Application Programming Interface |
| ATC | Abstract Test Case |
| DT | Distributed Tester |
| DTRON | Distributed TRON |
| ECDAR | Environment for Compositional Design and Analysis of Real-Time Systems |
| HTTP | Hyper Text Transfer Protocol |
| IO | Input-Output, can be denoted as i/o or I/O as well |
| IUT | Implementation Under Test |
| JSON | JavaScript Object Notation |
| LT | Local Tester |
| MBT | Model-Based Testing |
| NTA | Network of Timed Automata |
| OS | Operating System |
| POD | Per cent of Decrease |
| REST | Representational State Transfer |
| SUT | System Under Test |
| TA | Timed Automata |
| TCTL | Timed Computation Tree Logic |
| TRON | Testing Real-time Systems Online |
| TalTech | Tallinn University of Technology |
| UTA | Uppaal Timed Automata |
| WSGI | Web Server Gateway Interface |
| XML | Extensible Markup Language |

# Table of Contents

# List of Figures

# List of Tables

# 1. Introduction

Testing is an essential part of the development process, generally taking 30 to 60 per cent of gross development effort. Testing aims to show that the system's actual behaviour conforms with the expected behaviour. [1]

Model-based testing (MBT) is generally understood as black-box conformance testing, where a model of the expected behaviour of the system under test (SUT) is created. The SUT model is created based on the requirements of the behaviour, which usually include a subset of the entire behaviour. The tests are automatically generated from the model, opposite to manual testing, where tests are manually designed. MBT is said to reduce the test design effort and increase the testing coverage of possible behaviours of a system. [1]

The SUT observable ports can be spatially distributed in distributed systems, and the communication between components is conducted by passing messages. Two approaches are usually meant in distributed testing: remote testing with a centralised remote tester and a fully distributed set of local testers. In remote testing with a centralised remote tester, the test configuration is composed of SUT and remote tester. In remote testing with a fully distributed set of local testers, the test configuration is composed of SUT and local testers, each local tester assigned to one group of ports based on the spatial distribution of ports. With distributed real-time systems, strict timing constraints often lead to delay and timing issues. [2] In the paper [3], a parameter $\Delta$ is proposed, which sets the upper bound for the delay in order to ensure that the testing result is correct for remote testing. Distributed testing with local testers has the advantage to have $\Delta$-controllability. Whereas with centralised remote tester, we have $2\Delta$-controllability. Hence, the centralised remote tester cannot be used with real-time systems of timing constraints requirements less than $2\Delta$. [2] However, synchronising the distributed tester components creates communication overhead that could influence the testing results and should be kept as low as possible. Though this observation has been mostly ignored in testing distributed soft real-time systems, the reduction of communication overhead provides considerable improvement in test reaction time that, in turn, may be critical in testing hard real-time applications. This motivation has forced looking for better solutions, especially in the MBT domain of cyber-physical systems.

MBT testing is generally divided into five steps. The five steps depicted in Figure 1 are SUT modeling, test purpose specification, test generation, test deployment, and test execution. The first step is to model the SUT based on the requirements. In the second step, we specify the testing purpose by choosing a subset of the model's behaviour. The next step is to format the test purpose to abstract test cases (ATCs) used to generate executable test cases in step four. [4] Between steps three and four, the tester distribution algorithm can be applied to models to achieve a fully distributed tester with local testers. Finally, the tests are executed on the SUT, and the results can be analysed [4].

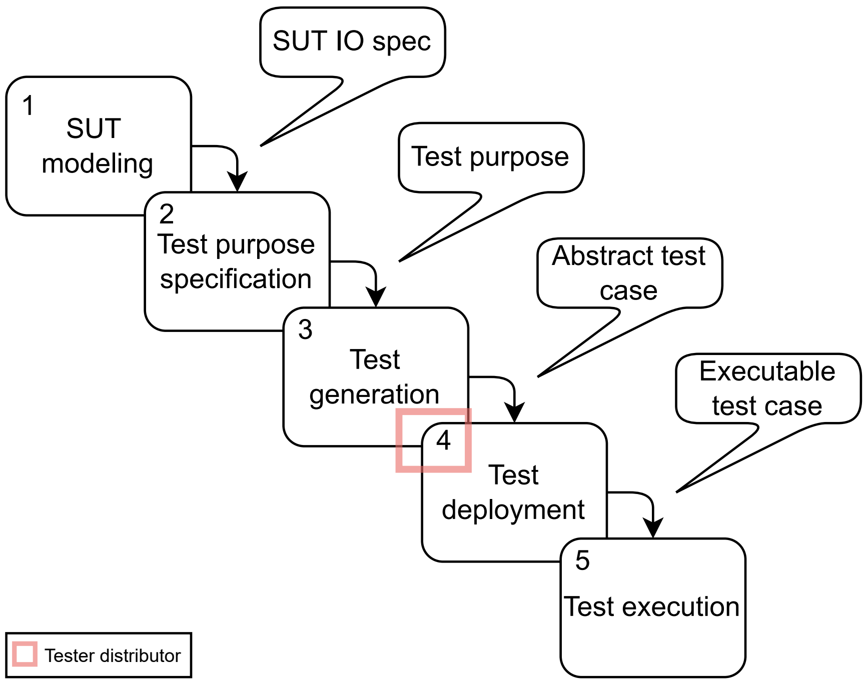


Figure 1. *Model-based testing workflow.*

The tester distribution algorithm (herein Algorithm 1) proposed in [5] assumes a centralised tester model has been generated using a method proposed in [6] and copies the remote tester to a local tester for each spatially distributed group of ports. This distribution causes structural overhead, which leads to communication overhead. Structural and communication overhead results in longer test traces, longer test time and more workload on communication channels.

We extend Algorithm 1 to optimise the local testers structurally to lower the structural overhead. In order to optimise the local testers, we explore one possible optimisation approach. The optimisation is composed of two steps. We first detect the causal redundancies and then remove the model elements deemed redundant. Detection is carried out by analysing observations of structures in the centralised tester and applying a general search algorithm to find the causal redundancies. Then, the reduction is carried out by utilising the detected causal redundancies by removing redundant model elements and adding necessary synchronisation to assure the preservation of global behaviour at SUT test ports. The optimisation is further motivated by the fact that Algorithm 1 has not been optimised yet to the author's best knowledge.

Algorithm 1 is conceptual, and has been applied manually in earlier works. However, manually applying algorithms can be time-consuming and prone to human error. Further, validating the result would be additionally time-consuming and likewise prone to human error. Therefore, offering a way to distribute and validate the result automatically is one of the goals of this thesis. This goal is further motivated by the fact that as of writing this thesis, as to the best of author's knowledge, there is no known implementation given to Algorithm 1.

The validation and comparison of the algorithms are carried out by using Uppaal[1] model checker tools as the tester distribution algorithm's input requirement is to use Uppaal timed automata (UTA) formalism. Uppaal toolbox offers an extensive set of tools that can be used to model real-time systems, verify models and simulate models. In order to validate the algorithms, the bisimulation relation is checked between the centralised tester and distributed tester with respect to the observable input/output ($i/o$) actions on the ports of the SUT. Consequently, Algorithm 1 and its optimised version Algorithm 2 are to be fully implemented with automated validation included. The resulting distributed testers generated by the algorithms are compared by structural, trace length, time cost, and total synchronisation message count difference. We further integrate the implementation to a testing tool in TalTech called BugBroom, which is a tool to provide web-based model checking and testing toolset to TalTech researchers and as a future perspective to be included in the education work.

Therefore, the **goals** of this thesis are as follows:

1. Extending Algorithm 1 by providing one possible optimisation approach to optimise the distributed tester model structural complexity.
2. Validating the algorithms by checking the bisimulation relation between the centralised tester and distributed tester with respect to the observable $i/o$ events on the SUT ports.
3. Implementing Algorithm 1 and the optimisation of Algorithm 1 (Algorithm 2) with automated validation of the output of the algorithms in the implementation.
4. Integrating the implementation to MBT toolset BugBroom.

[1]https://uppaal.org

The theoretical background is covered in section 2. Section 3 covers the optimisation process and presents the optimised tester distribution algorithm. Section 4 covers the process of validation. Section 5 presents the optimisation results. Section 6 covers the implementation of algorithms and integration to the testing environment BugBroom.

# 2. Preliminaries

This section covers the concepts to support further discussions and analysis. We first begin with describing model-based testing and bringing out the general workflow. Subsequently, we bring in real-time testing requirements, and in distributed testing, we cover two approaches for distributed testing. Next, we present Uppaal timed automata and the general notion necessary to support discussion on algorithms. Subsequently, the bisimulation section covers the prerequisite knowledge to support the validation of algorithms. Finally, we present Algorithm 1 to be optimised.

## 2.1 Model-based testing

In model-based testing (MBT), the intended behaviour of a system under test (SUT) is specified by models. Given SUT and behaviour models, the output of the pairs of input and output of the model is regarded as the expected output of the SUT. [4]

MBT is generally understood as black-box conformance testing, where the test cases are derived from the system requirements model. In black-box testing, the internal behaviour of the SUT is disregarded, and the tester can only control the input and observe the output. The derived test cases are executed during MBT, and the result is emitted as a test verdict. The test verdict can be a pass, fail or inconclusive and denotes the conformance between the requirements and the actual implementation. [7] The MBT process can be divided into five main steps, which are illustrated in Figure 1. The steps proceed as follows.

***SUT modeling***. The first step is to build a model of the SUT based on the requirements or specifications that describe the intended behaviour of the SUT. In this step, the model should be abstracted to be more simple than the SUT. Only the parts we wish to test should be included. [4] In this step, one may use the Uppaal toolbox to create a model of the SUT.

***Test purpose specification***. The second step is to choose the test selection criteria. Test selection criteria define the aim of the testing. The aim can be to test the model's structural properties, such as test coverage of transitions and states. The aim can also be focused on the given functionalities of a system. [4]

***Test generation***. The third step is to transform the test selection criteria to test case specifications. Test case specifications render the test selection criteria to a format used in the fourth step to generate test cases. [4] This step is to generate abstract test cases (ATCs). Abstract test cases are generated from the SUT model according to the test selection criteria. [1] The tester distribution algorithm is the intermediate step between the third and fourth steps. We decompose the centralised tester to a fully distributed tester by applying the tester distribution algorithm. In this step, one may use Uppaal models to represent the expected behaviour.

***Test deployment***. The fourth step is to transform ATCs into executable tests. Here the ATCs are implemented considering the low-level SUT details not included in the abstract model. [1] In this step, one may create executable tests from models by using a tool such as TRON[1].

***Test execution***. The fifth step is to execute the tests against SUT and assign verdicts. Here the execution difference is evident in whether the testing is online or offline. In online testing, the tests are executed as they are produced, i.e. steps 2 through 4 are generally merged. In offline testing, these steps usually remain separated. [1] In this step, we can execute the tests, and for distributed tester, one may use a tool such as DTRON[2].

After the fifth step, we are able to analyse the results. Given a test verdict of failure, the fault can be determined, and corrective actions can be made [1].

## 2.2 Real-time testing requirements

In real-time systems, timing constraints compliance plays a vital role in the system's behaviour. Given a real-time system, the system's behaviour is dependent on the input and the timing of the input. Further, the timing of the output upon input has timing constraints. [8]

Therefore, in real-time system testing, the input given by the tester and the output received upon input have timing constraints. Hence, the timing constraints proceed to determine the correctness of the behaviour, i.e. correct value is emitted at a specific time point. [8]

[1]https://uppaal.org/features/#tron
[2]https://cs.ttu.ee/dtron/dtronTutorial.pdf

## 2.3 Distributed testing

Under distributed testing, two alternative approaches are usually meant: remote testing with a centralised remote tester and a fully distributed set of local testers. In remote testing with a centralised remote tester, a single centralised tester generates all the test inputs. The input is generated for a specific port by a tester, and before another test input can be generated, the result must be received in some output port. This process continues until a test verdict can be emitted. Therefore, in non-negligible signal propagation time systems, the tester may not satisfy the timing constraints. [9]

The communication latency or signal propagation time between the tester and the SUT can lead to the interleaving of inputs and outputs, which affects the generation of inputs and the observation of outputs, possibly resulting in a wrong test verdict [3]. In the paper [3], $\Delta$-testability criterion is proposed. Parameter $\Delta$ is defined as the upper bound for communication latency or signal propagation time between the tester and the SUT to ensure the input and output interleaving never occurs, and the emitted test verdict is correct [3].

In remote testing, the communication is bidirectional between SUT and the tester. As a result, it is assumed that the time it takes to send the test input and receive the result is not less than $2\Delta$. Hence, the remote tester is not suitable for real-time systems with spatially distributed ports and timing constraints requiring response upon input sooner than $2\Delta$. [7]

The timing constraints incompliance can be overcome with a distributed testing approach extending the $\Delta$-testing by decomposing the centralised tester into multiple local testers. The bidirectional communication between the tester and SUT is replaced with unidirectional communication between the tester's local components. As the local testers are directly attached to the ports of the SUT, the communication delay with local ports can be ignored. The unidirectional communication occurs between the local testers for synchronisation purposes to propagate the local output and input to other local testers. The local testers are generated so that the correctness of the testers is preserved so that if the centralised tester is $2\Delta$-controllable, the distributed tester is $\Delta$-controllable. [2]

## 2.4 Uppaal timed automata

Uppsala University and Aalborg University developed the Uppaal toolbox. The purpose of the toolbox is to verify real-time systems. The tool's design focuses on systems that can be modelled as networks of timed automata (NTA). Uppaal modelling language takes the theory of TA introduced in [10] as a base and extends it with additional features such as committed locations, broadcast channels, and constants. [11] TA can be intuitively understood as a state-transition system with timing constraints. Timing constraints can be described through the notion of finitely many real-valued clocks. The usage of clocks allows modelling the behaviour of real-time systems over time. [10] NTA is composed of several TA in parallel, which defines a system. States of a system are defined through discrete variables values, clock values and all the locations of an NTA. [11] Uppaal provides an extensive set of features which are covered in [11]. Some of the offered features to support the discussion on algorithms are as follows.

Templates are used to represent parameterised automata. The nodes of an automaton are referred to as locations, and directed edges are referred to as transitions. A transition has a source location and a target location. Variables, channels, constants, and clocks can be declared locally in a template or globally in NTA. [11]

Transitions may have labels such as guard, assignment (update), and synchronisation. Guards set constraints on transitions that must be satisfied for the transition to be enabled, e.g. $x > 3$. The constraints are expressions that evaluate to a boolean value and may refer to clocks, integer variables and constants. An assignment is a comma-separated list of expressions that allows updating variables, e.g. $x := 3$. An assignment may refer to integer variables, constants, and clocks. The synchronisation between templates can be done through channels. Channel declared as $chan$ "$c$" forms a synchronisation pair "$c$!" and "$c$?", where send-action is denoted by channel name followed by "!" and receive-action "?". In order to send synchronisation as $chan$, the receiver must be ready. Channel declared as $broadcast\ chan$ can execute "$c$!" even if there are no receivers. For $broadcast\ chan$, the number of receivers can be arbitrary, and any receiver that can synchronise must do so. [11]

Location may be specified as $committed\ location$, $urgent\ location$, $initial\ location$, and may have constraints in form of $invariant$. Invariants set constraints on locations to allow visiting only the locations that satisfy the constraints. Invariants are expressions that may refer to clocks, integer variables or constants, e.g. $x < 5$. The initial location is to specify the initial state. Urgent location, denoted by "$U$", sets constraints on time in terms of entering the location and leaving the location. The constraint is not to let time pass when in an urgent location, i.e. it must be left without letting time pass. However, interleaving with other automata is allowed. A committed location, denoted by "$C$", sets further constraints on execution. Without delay, the committed location must be left by the first outgoing transition. [11]

## 2.5 Bisimulation

Bisimulation is an equivalence relation that allows us to distinguish between systems (agents) or to show the equivalence between systems' behaviour. The actions of one system have to be matched by the actions of the other system. [12] In order to define bisimulation, we bring in a formal definition of TA and preliminary properties from paper [13].

The necessary **notion** used to define TA are as follows: "Assume a finite set of real-valued variables $\mathcal{C}$ ranged over by $x$, $y$ etc.standing for clocks and a finite alphabet $\Sigma$ ranged over by $a$, $b$ etc.standing for actions." [13] Further, the **clock constraints** are defined as follows: "A clock constraint is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}, \sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. Clock constraints will be used as guards for timed automata. We use $\mathcal{B}(\mathcal{C})$ to denote the set of clock constraints, ranged over by $g$ and also by D later." [13]

**Timed automaton** is defined as follows: "A timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, E, I \rangle$ where

– $N$ is a finite set of locations (or nodes),
– $l_0 \in N$ is the initial location,
– $E \in N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges and
– $I : N \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations" [13]

The necessary **notion** to define operational semantics is defined as follows: "The semantics of a timed automaton is defined as a transition system where a state or configuration consists of the current location and the current values of clocks. There are two types of transitions between states. The automaton may either delay for some time (a delay transition), or follow an enabled edge (an action transition). To keep track of the changes of clock values, we use functions known as *clock assignments* mapping $\mathcal{C}$ to the non-negative reals $\mathbb{R}_+$. Let $u$, $v$ denote such functions, and use $u \in g$ to mean that the clock values denoted by $u$ satisfy the guard $g$. For $d \in \mathbb{R}_+$, let $u + d$ denote the clock assignment that maps all $x \in \mathcal{C}$ to $u(x) + d$, and for $r \subseteq \mathcal{C}$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in $r$ to 0 and agree with $u$ for the other clocks in $C \setminus r$. " [13]

**Operational semantics** is defined as follows: "The semantics of a timed automaton is a transitions system (also known as a timed transition system) where states are pairs $\langle l, u \rangle$, and transitions are defined by the rules:

– $\langle l, u \rangle \xrightarrow{d} \langle l, u + d \rangle$ if $u \in I(l)$ and $(u + d) \in I(l)$ for non-negative real $d \in \mathbb{R}_+$

– $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$ if $l \xrightarrow{g,a,r} l',\ u \in g,\ u' = [r \mapsto 0]u$ and $u' \in I(l')$" [13]

The necessary **notion** to define bisimulation is as follows: "A *timed action* is a pair $(t, a)$, where $a \in \Sigma$ is an action taken by an automaton $\mathcal{A}$ after $t \in \mathbb{R}_+$ time units since $\mathcal{A}$ has been started." [13] Timed action is denoted by $\sigma$. The states denoted by $s_1, s_2, s_1', s_2'$ are pairs in the form $\langle l, u\rangle$. [13]

The **definition of bisimulation** is as follows: "A bisimulation $R$ over the states of time transition systems and the alphabet $\Sigma \cup \mathbb{R}_+$, is a symmetrical binary relation satisfying the following condition: for all $(s_1, s_2) \in R$, if $s_1 \xrightarrow{\sigma} s_1'$ for some $\sigma \in \Sigma \cup \mathbb{R}_+$ and $s_1'$, then $s_2 \xrightarrow{\sigma} s_2'$ and $(s_1', s_2') \in R$ for some $s_2'$. Two automata are timed bisimilar iff there is a bisimulation containing the initial states of the automata." [13]

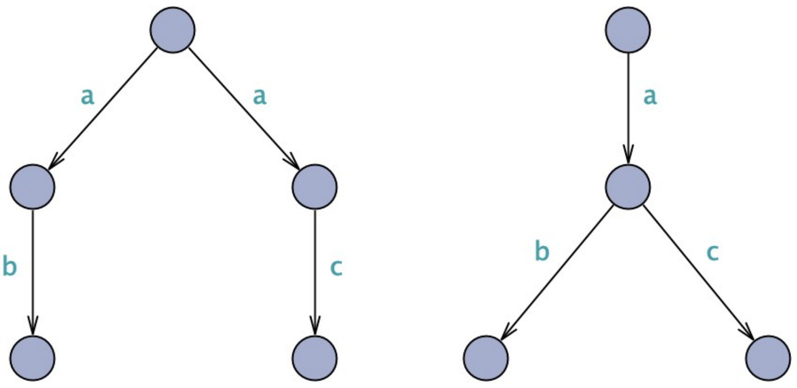


Figure 2. *Example of transition systems that are not bisimilar.*

An example of transition systems where the bisimulation relation does not hold is depicted in Figure 2. To define bisimulation on the observable $i/o$ actions on the ports of a SUT, we replace $\Sigma$ with $\Sigma' \subseteq \Sigma$, where $\sigma' \in \Sigma'$ are the $i/o$ actions on the observable $i/o$ ports of the SUT. Bisimulation with respect to $\Sigma'$ is used as the basis for the validation of the algorithms in section 4. This allows us to show the correctness of the algorithms by checking the bisimulation equivalence relation between a centralised tester and a fully distributed set of local testers with respect to the observable $i/o$ actions on the ports of the SUT.

## 2.6 Tester distribution algorithm: Algorithm 1

One of the goals of this thesis is to optimise and implement the tester distribution algorithm (herein Algorithm 1) proposed in the paper [5]. By applying Algorithm 1, the centralised (monolithic) remote tester is transformed into a set of communicating distributed local testers. As a result, the test is $\Delta$-controllable instead of $2\Delta$-controllable, where the communication is unidirectional between the local testers [5]. Algorithm 1 is described as follows:

"Let $M^{MT}$ denote a monolithic remote tester model generated by applying the reactive planning online-tester synthesis method [6]. $Loc(IUT)$ denotes a set of geographically different port locations of $IUT$[3]. The number of locations can be from 1 to $n$, where $n \in \mathbb{N}$ i.e. $Loc(IUT) = \{l_n \mid n \in \mathbb{N}\}$. Let $P_{l_n}$ denotes a set of ports accessible in the location $l_n$.

1. For each $l$, $l \in Loc(IUT)$ we copy $M^{MT}$ to $M^l$ to be transformed to a location specific local tester instance.
2. For each $M^l$ we go through all the edges in $M^l$. If the edge has a synchronizing channel and the channel does not belong to the set of ports $P_{l_n}$, we do the following:
   – if the channel's action is $send$, we replace it with the co-action $receive$.
   – if the channel's action is $receive$, we do nothing.
3. For each $M^l$ we add one more automaton that duplicates the input signals from $M^l$ to $IUT$, attached to the set of ports $P_{l_n}$ and broadcasts the duplicates to other local testers to synchronize the test runs at their local ports." [5] The local output event observations are similarly broadcast to other local testers for the same purposes [5].

All in all, the locations (geographical) are chosen based on the spatial distribution of the SUT test ports. Only the local actions remain as $send$. The synchronisation mechanism used to synchronise the local test runs for local testers is described in step 3. Finally, the correctness of Algorithm 1 is outlined in [5] by checking the bisimulation equivalence relation between the model of the initial centralised tester and the resulting distributed tester.

[3]System under test (SUT) is also referred to as implementation under test (IUT) in the literature.

# 3. Optimisation of Algorithm 1

The tester distribution Algorithm 1, introduced in section 2.6, copies the remote tester to each geographical location according to the groups of ports. In order to synchronise test runs for local testers at their local ports, the local testers communicate by sending synchronisation messages about all locally observed $i/o$ events. However, a local tester needs to only know about the $i/o$ events that directly influence its behaviour. Therefore, local testers should selectively send synchronisation messages to a set of local testers by considering if the event directly influences their subsequent behaviour. Further, the local testers should only receive synchronisation messages about the non-local $i/o$ events directly influencing its behaviour. Consequently, the copying of remote tester to each geographical location results in structural redundancies, leading to communication overhead. The **requirements** for the optimised algorithm are as follows:

(a) Input is a centralised tester in UTA formalism as an NTA composed of two automata templates: remote tester and SUT. The $i/o$ events are specified on the ports of the SUT. A unique set of ports is assigned to each geographical location. Further, the remote tester is assumed to be deterministic.
(b) Output is a distributed tester in UTA formalism composed of a SUT and a group of local testers.
(c) Distributed tester reaction time upon SUT outputs does not exceed the distributed tester message propagation greatest delay $\Delta$. ($\Delta$-max message propagation time between local testers). In other words, we must ensure $\Delta$-controllability.
(d) Given a *deadlock free* centralised tester, the resulting distributed tester must remain *deadlock free* in Uppaal testing tool.
(e) The parallel composition of local testers has to ensure equivalent (bisimulation relation) behaviour on the observable test ports with the centralised tester.

In the sub-sections 3.1 to 3.3, one possible optimisation approach is explored to reduce the structural and consequently communication overhead. In section 3.1 the requisite augmentations of Algorithm 1 are presented before optimisation is conducted. Section 3.2 covers the process of detecting causal redundancies with examples and pseudocode. Section 3.3 uses the results of section 3.2 and presents pseudocode to remove elements deemed redundant and to add synchronisation.

## 3.1 Algorithm 1 augmentations

Prior to optimisation, Algorithm 1 is augmented in order to satisfy requirement (d). Further, we specify the synchronisation sending mechanism used. All of the decisions are related to the fact that Uppaal uses *interleaving semantics*[1]. More specifically, we focus on the *interleaving semantics* property that at any point in execution, one transition is picked non-deterministically. The **augmentations** are as follows:

(I) Synchronising non-synchronisation transitions between local testers. In order to meet the requirement (d), synchronisation is added between transitions that appear in at least two local testers and have no $i/o$ synchronisation action. Otherwise, Uppaal is able to generate test sequences where *deadlock* occurs as it is using *interleaving semantics*. In Figure 3, an example is depicted, where $Local\ tester$ 1 transitions have been non-deterministically chosen until a committed location to send synchronisation to $Local\ tester$ 2 is reached. As the $Local\ tester$ 2 is not ready to receive, the test run results in *deadlock*. The described issue is resolved by including the described augmentation to local testers. How this was technically achieved is covered in the implementation section 6.2.1.

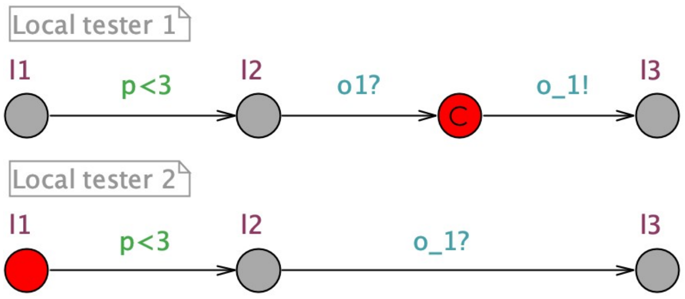


Figure 3. *Segments of an NTA showing transitions with no synchronisation.*

(II) Using $chan$ instead of $broadcast$. To meet the requirement (d), as an addition to augmentation (I), synchronisation channels are declared as $chan$. Broadcast is a *one-way-handshake*, synchronisation can be sent if there is no one receiving. Hence, by using $chan$, we enforce that the receiver must be ready to send synchronisation. Further, transitioning to $chan$ also results in unique synchronisation (one sender and one receiver), allowing us to send synchronisation messages selectively. The synchronisation messages format is implementation-specific and is covered in 6.2.1.

(III) The synchronisation sending mechanism is added to the local tester's automaton template. Accordingly, instead of using additional automaton templates for adapters to broadcast local event occurrences, the synchronisation is added to the local

[1]https://courses.engr.illinois.edu/cs477/sp2013/lectures/23-promela-2x3.pdf

testers by using intermediate committed locations. The augmented synchronisation mechanism ensures that the local testers send all selected synchronisation messages before continuing the local test run. Further, as intermediate committed locations are used to add the synchronisation, the state space does not increase. Additionally, as committed locations do not let time pass, we maintain the timing properties of local testers. How this was technically achieved is covered in the implementation section 6.2.1.

## 3.2 Detection of causal redundancies

***Structures***. Consider segments of an NTA depicted in Figure 4. The NTA is composed of three geographical locations (henceforth referred to as $loci$). For each $locus$ a segment of local tester is presented where all are initially at location $l1$. The segments are sequences of transitions and each $locus$ sends synchronisation messages regarding locally observed events in order to keep local test runs synchronised, e.g. given $locus$ 1 input event $i\_1!$, edges $i\_1\_2!$ and $i\_1\_3!$ are fired, which are received by $locus$ 2 and $locus$ 3, respectively.

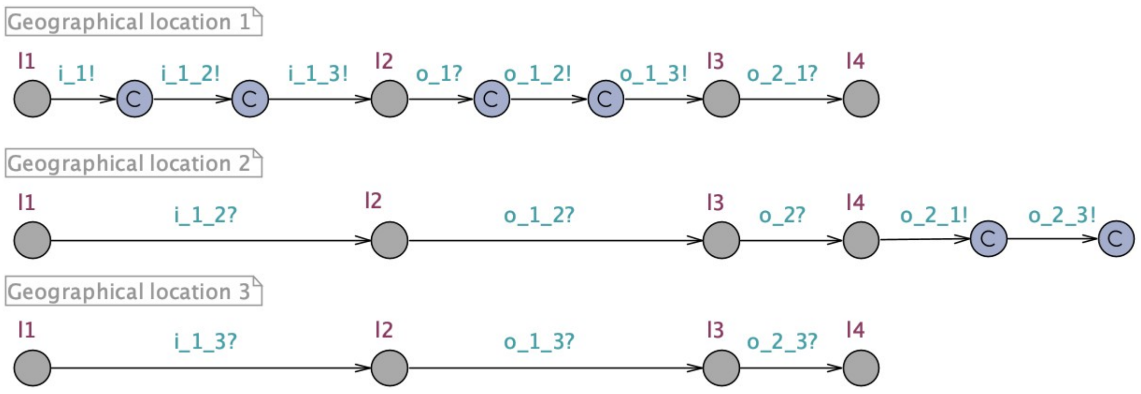


Figure 4. *Segments of an NTA.*

***Key idea***. However, by considering the discussion regarding Algorithm 1 overhead in the first paragraph of section 3, one might notice that $locus$ 3 does not have local events and $locus$ 2 has two non-local events in a row. The key idea in reducing redundancies in this thesis is to contract sequences in a local tester with more than one non-local event in a row. Further, only the last non-local event in described sequences is necessary to ensure synchronised test runs. Local events are considered to be local $i/o$ and assignments. Assignments are required to keep the local variable values synchronised with other local testers to provide correct valuations in terms of guards and invariants. Therefore, the causal dependencies are local $i/o$, assignments, and last non-local events in the sequences of transitions. In Figure 5, the general idea behind the redundancy reduction is depicted.

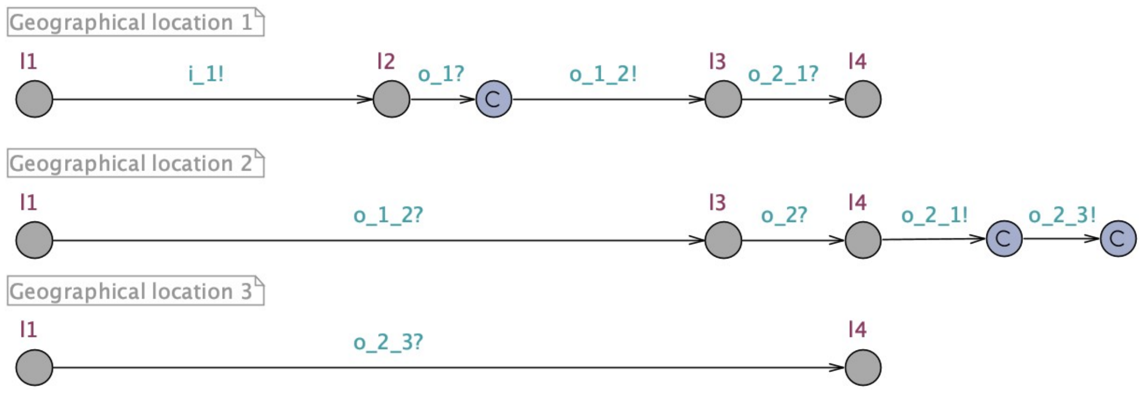


Figure 5. *Segments of an NTA with reduced redundancies.*

In order to detect the redundancies, the necessary model elements must be collected to determine where to connect the last non-local event synchronisation and what can be removed. As the search is to cover every possible path, a breadth-first search is chosen to be suitable. Hence, to conduct the structural analysis on a remote tester, a breadth-first search is performed, layer-by-layer, from local events for each locus by going back along the event causality chain.

In Figure 6, pseudocode is presented to perform the optimisation. The optimised tester distribution algorithm (henceforth referred to as Algorithm 2) extends Algorithm 1 by reducing structural and communication overhead. Algorithm 2 is divided into two main steps. The first step is causal redundancy detection, which is covered in this sub-section, and the second step is redundancy reduction, which is covered in the next sub-section. The first step contains the use of two search algorithms, depicted in Figure 7 and Figure 8. The second step is divided into synchronisation of local testers and model elements removal, depicted in Figure 9 and Figure 10, respectively.

***Algorithm description***. The input for Algorithm 2 is a centralised tester model as described in requirement (a). The output of the Algorithm 2 is a fully distributed tester model as described in requirement (b). The *for*-loop covering lines 3 to 23 is done over the set of loci. The set of loci is defined as $loci = \{locus_i \mid i \in [1, ..., n]\}$, where $n \in \mathbb{N}$. In line $10$, we access the set of ports accessible from one geographical location $locus_i \in loci$ is denoted by $locus\_ports_i = \{p \mid accessible(p, locus_i)\}$, where $accessible(p, locus_i)$ is true *iff* port $p$ is accessible from $locus_i$.

In line 4, we make a copy of the remote tester. In lines $5$ to $9$, five empty sets are created: $h1$, $h2$, $keep$, $remove$, and $visited\_transitions$. The set $h1$ is a set of triples, $h2$ is a set of triples, $keep$ is a set of locations, $remove$ is a set of locations, and $visited\_transitions$ is a set of transitions. The set $h1$ is defined as $h1 = \{(l2, l1, t) \mid$

$l2$ and $l1$ are locations, $t$ is a transition$\}$. In the set $h1$ triples the location $l1$ is the target of the last non-local event, $l2$ is respectively the source and $t$ is the transition. The set $h2$ is defined as $h2 = \{(l2, l1, t) \mid l2$ and $l1$ are locations and $t$ is a transition$\}$. In the set $h2$ triples the transition $t$ is from $h1$ triple, location $l2$ is where the last non-local synchronisation is sourced post contraction of the sequence of transitions (possibly no contraction), and $l1$ is respectively the target location or in other words $l1$ from set $h1$. The set $keep$ is defined as $keep = \{k \mid k$ is a location$\}$, these are the locations that must remain to ensure connectivity in the local testers, hence cannot be removed. Hence, initially the set $keep$ contains all $t \in local\_transitions_i$ source and target locations. The set $remove$ is defined as $remove = \{r \mid r$ is a location$\}$, these are the intermediate locations in the causality chain that can be removed if $r \notin keep$.

```
1   INPUT:MT
2   OUTPUT:DT
3   for locus in loci:
4       local_tester = remote_tester.copy()
5       h1 = {}
6       h2 = {}
7       keep = {}
8       remove = {}
9       visited_transitions = {}
10      locus_ports = get_locus_ports(locus)
11      all_transitions = local_tester.get_transitions()
12      local_transitions = get_local_transitions(locus)
13      for t in local_transitions:
14          h1_, keep_, h2_ = bfs_1(t, locus_ports, local_transitions,
                all_transitions)
15          h2 = h2_.union(h2)
16          h1 = h1_.union(h1)
17          keep = keep_.union(keep)
18      for triple in h1:
19          h2_, remove_ = bfs_2(triple, locus_ports, all_transitions,
                local_transitions)
20          h2 = h2_.union(h2)
21          remove = remove_.union(remove)
22      add_sync(h2, local_tester)
23      remove_from_model(h2, remove, keep, local_tester)
```

Figure 6. *Algorithm* 2.

In line 12, a set of transitions $local\_transitions$ is defined as $local\_transitions_i = \{t \mid t$ is a transition and $local(t)\}$. The predicate $local(t)$ is true *iff* $has\_update(t)$ *or* $local\_event\_port(t)$ *or* $has\_initial\_source(t)$. The predicate $has\_update(t)$ is true *iff* there is an update on transition $t$. The predicate $local\_event\_port(t)$ is true *iff* $t$ contains an $i/o$ event and the port $p \in locus\_ports_i$ The predicate $has\_initial\_source(t)$ is to maintain the initial locations of the models and is true *iff* transition $t$ source is the initial location.

In the *for*-loop covering lines 13 to 17 we apply the first search to each transition $t \in local\_transitions_i$. The search results in three sets $h1\_$, $h2\_$, and $keep\_$ which are unified with sets $h1$, $h2$, and $keep$, respectively. Similarly, in the *for*-loop covering lines 18 to 21, result in sets $h2\_$ and $remove\_$ which are unified with sets $h2$ and $remove$, respectively.

***Finding last non-local events***. The details of the first search, depicted in Figure 7, are as follows. In line 6, the predicate $initial\_or\_non\_local\_update(t)$ is true *iff* $has\_initial\_source(t)$ or $has\_sync(t)$ and $has\_update(t)$ and not $local\_port\_sync(t)$. If the predicate is satisfied, we add the transition $t$ with $source$ and $target$ to the set $h2\_$. First, this is to ensure the initial location remains. Secondly, as the updates are local, we must account for the case when the transition also includes a non-local $i/o$ event and therefore add the necessary synchronisation. In line 10, we add the source of the local transition to the set $keep\_$ as this is where the synchronisation is connected. We use queue containing locations to conduct the search until the queue is empty, where the first location is the source location of local transition, as can be seen in line 11.

```
1  INPUT: t
2  OUTPUT: h1_, keep_, h2_
3  h1_ = {}
4  keep_ = {}
5  h2_ = {}
6  if initial_or_non_local_update(t)
7      keep_.add(t.target)
8      h2_.add((t.source, t.target, t))
9  queue = Queue()
10 keep_.add(t.source)
11 queue.put(t.source)
12 while not queue.empty():
13     l1 = queue.get()
14     backward = get_backward_transitions(l1)
15     for b in backward:
16         l2 = b.source
17         if b in local_transitions:
18             pass
19         elif halt_condition_1(b):
20             visited_transitions.add(b)
21             h1_.add((l2, l1, b))
22         else:
23             visited_transitions.add(b)
24             keep_.add(l2)
25             queue.put(l2)
```

Figure 7. *Algorithm* 2 *finding last non-local events.*

The *while*-loop covering lines 12 to 25 contains the following steps. For each iteration we get a location $l1$ from the queue and find all transitions $backward$ with given location as the target as can be seen in lines 13 and 14. Secondly, for each transition $b$ in $backward$, we check for three conditions:

1. If the transition $b \in local\_transitions_i$, we skip the transition and select next from the queue as the search is applied for each transition in $local\_transitions_i$ and hence is covered by some other transition.
2. If the transition satisfied $halt\_condition\_1(b)$, we add the transition $b$ to the set $visited\_transitions$ and $(l2, l1, b)$ to the set $h1\_$. The location $l2$ is the source of the transitions $b$ and $l1$ is the current queue element. The predicate $halt\_condition\_1(b)$ is true *iff* $has\_sync(b)$ and not $local\_port\_sync(b)$. This allows us to detect the subsequently last non-local $i/o$ event in the sequences of transitions.
3. If neither of the above applied, we continue the search by adding the locations to the queue, transition to the set $visited\_transitions$, and the set $keep\_$ as the transition will remain local to ensure connectivity of the local tester.

***Finding where to connect last non-local events***. The details of the second search, depicted in Figure 8, are as follows. We use a queue containing locations to conduct the search until the queue is empty. The search begins from the location $l2$ of the triple from $h1$, as can be seen in line 6. The *while*-loop covering lines 7 to 19 consists of the following steps. In line 9, we check the predicate $is\_local\_source\_location(v1, t)$ which is true *iff* $\exists$ transition $t2$ such that $t2.source = v1$ and $is\_local\_transition(t2)$ and $t2 \neq t$.

```
 1  INPUT: (l2,l1,t)
 2  OUTPUT: h2_, remove_
 3  h2_ = {}
 4  remove_ = {}
 5  queue = Queue()
 6  queue.put(l2)
 7  while not queue.empty():
 8      v1 = queue.get()
 9      if is_local_source_location(v1, t):
10          h2_.add((v1, l1, t))
11      else:
12          backward = get_backward_transitions(v1)
13          for b in backward:
14              v2 = b.source
15              if halt_condition_2(b):
16                  h2_.add((v1, l1, t))
17              else:
18                  remove_.add(v1)
19                  queue.put(v2)
```

Figure 8. *Algorithm 2 finding where to connect last non-local events.*

In other words, if there is another local transition with the source location $v1$, we must halt as the location cannot be removed. If the predicate is satisfied, we add $(v1, l1, t)$ to the set $h2\_$, where $v1$ is the synchronisation source post contraction, $l1$ is the target, and $t$ is the transition between them, respectively. If the predicate was not satisfied, we find all transitions $backward$ with $v1$ location as the target in line 12. Next, we traverse all the $b \in backward$ transitions and check for the following conditions:

1. If $halt\_condition\_2(b)$ is satisfied, we add $(v1, l1, t)$ to the set $h2\_$. The predicate $halt\_condition\_2(b)$ is true *iff* $b \in local\_transitions_i$ or $b.target \in keep$ and $b \in visited\_transitions$. If the transition is in local transitions, then the search is conducted from there already and the current search stops. If the transition target is in set $keep$ and the transition in $visited\_transitions$, then the search is conducted from there already and we can stop the current search.
2. If the predicate $halt\_condition\_2(b)$ was not satisfied, we add the $v1$ location to the set $remove$ and $v2$ location to the queue. Location $v1$ is added to the $remove$ set because this is the intermediate location in the causality chain that can possibly be removed.

***Resulting sets***. Consequently, the searches resulted in four sets: $h1$, $h2$, $keep$, and $remove$, which are used in the redundancy reduction step.

## 3.3 Reduction of causal redundancies

Algorithm 2 redundancy detection step resulted in four sets: $h1$, $h2$, $keep$, and $remove$. The next step is to contract the sequences of transitions by removing the intermediate elements in the causality chain and keeping the last non-local event as the synchronisation event.

***Adding synchronisation***. In Figure 9, the general idea of using the resulting sets to add synchronisation is depicted. In the *for*-loop covering lines 4 to 14, we traverse over the triples $(l2, l1, t) \in h2$. In line 5, we make a copy $tc$ of the transition $t$ to modify it. In line 6, we access the last non-local transition with target location $l1$ and change the source to $l2$. If the condition in line 7 is satisfied, we do the following steps. In line 9, we access the transition $tc$ synchronisation components used in lines 10 and 12 to create a synchronisation pair of receiver and sender. In line 14, we add the transition to the local tester. The transitions are added to the local tester if the line 7 condition was not satisfied as well as we are keeping initial locations that may not have $i/o$ event transitions connected to it. A more technical description of the synchronisation mechanism is covered in the implementation section 6.2.1

```
INPUT: h2, local_tester
OUTPUT: local_tester
id = 0
for (l2, l1, t) in h2:
    tc = t.copy()
    tc.source = l2
    if has_sync(tc) and not is_local_port_sync(tc):
        original_sync = tc.synchronisation
        io, non_local_locus, non_local_port = get_sync_components(original_sync)
        receive_sync = create_receive_sync(original_sync, io, non_local_locus, non_local_port, id)
        t.synchronisation = receive_sync
        add_send_sync(original_sync, io, non_local_locus, non_local_port, id)
        id += 1
    local_tester.transitions.append(tc)
```

Figure 9. *Algorithm* 2 *adding synchronisation.*

***Removing elements***. In Figure 10, the pseudocode for removing redundant elements is shown. For each locus, we need the sets $keep$, $remove$ and $h2$ to reduce redundancies. In line $3$, all the transitions are collected from the local tester. In the *for*-loop covering lines $4$ to $6$, we first remove all the transitions replaced by transitions with modified synchronisations, i.e. remove duplicates. In line $7$, we take the set difference $locations = remove \setminus keep$ in order to keep only the locations that can be removed. In lines $8$ to $15$, we detect the transitions and locations to be removed and perform redundancy reduction by removing them.

```
1  INPUT: h2, remove, keep, local_tester
2  OUTPUT: local_tester
3  transitions = local_tester.get_transitions()
4  for (l2,l1,t) in h2:
5      if t in transitions:
6          transitions.remove(t)
7  locations = remove.difference(keep)
8  for location in locations:
9      transitions_remove = {}
10     for transition in local_tester.transitions:
11         if transition.target == location or transition.source ==
               location:
12              transitions_remove.add(transition)
13     for transitions in transitions_remove:
14         local_tester.transitions.remove(transition)
15     local_tester.locations.remove(location)
16 remove_non_local_transitions(local_tester)
```

Figure 10. *Algorithm* 2 *removing elements.*

In line $16$, we remove the remaining non-local transitions as the intermediate non-local transitions can have a source and target not deemed redundant. These non-local transitions are the redundant intermediate transitions before the source for the last non-local transition is found, hence can be removed.

# 4. Validation of the algorithms

Section 3 presented the Algorithm 2 and section 3.1 described the requisite augmentations for Algorithm 1. The requirement (a) is satisfied since the input for Algorithm 2 is defined as a centralised tester with described properties. The requirement (b) is satisfied since the output is defined as distributed tester composed of SUT and local testers with described properties. The requirement (c) is satisfied since we maintained the Algorithm 1 property of unidirectional communication between local testers instead of bidirectional communication between remote tester and SUT. The requirement (d) was satisfied by providing augmentations (I) and (II). In this section, the validation of the algorithms is presented to satisfy requirement (e). In section 4.1, the method of validation is presented to confirm the correctness of the algorithms by checking the bisimulation equivalence relation between the centralised tester and distributed tester with respect to observable $i/o$ actions on the SUT ports. In section 4.2, the resulting traces are being compared.

## 4.1 Method of validation

To verify the correctness of the algorithms, we use model checking to verify the bisimulation equivalence relation between the centralised tester and distributed tester with respect to observable $i/o$ actions on the SUT ports. The validation process consists of two steps. The first step is to compose the models by parallel composition and carry out the following adjustments:

1. Adding synchronisation between the remote tester and set of local testers. In the remote tester, for all observable $i/o$ actions on the ports of the SUT, we add a preceding synchronisation action receive "?" from a local tester by using an intermediate committed location. The format for synchronisation is $m\_[i]?$ where $i \in \mathbb{N}$. The synchronisation action send $m\_[i]!$ is added to a local tester as a subsequent action following the equivalent local $i/o$ action using an intermediate committed location. The described configuration allows us to bind the $i/o$ actions to detect bisimulation relation violations in the synchronous parallel composition.
2. Copying the SUT for the remote tester and adding the prefix $m\_$ for the remote tester, and the copied SUT $i/o$ actions to distinguish the communication from the distributed tester. The described configuration allows us to track the $i/o$ actions of the

centralised tester. It is assumed that there are no transitions with no synchronisation in the SUT.

3. Adding synchronisation to transitions with no synchronisation in the parallel composition of the remote tester and local testers as described in 3.1 augmentation (I). The described configuration does not affect the observable $i/o$ actions on the ports of the SUT.

The second step is to confirm no deadlocks in the parallel composition. For that purpose, *verifyta* provided by the Uppaal toolbox is used with the following model checking query: "$A[]\ not\ deadlock$". The described query states that there is no deadlock in the system for all model execution paths. The parallel composition with the adjustments and the described query allows us to detect the actions where the observable $i/o$ actions on the SUT ports differ between the centralised tester and distributed tester and, therefrom, the violation of the bisimulation relation.

## 4.2 Equivalence results

The validation is conducted on an example of a centralised tester in Figure 16, Algorithm 1 output in Figure 21 and Algorithm 2 output in Figure 24. The deadlock freedom condition was satisfied for each synchronous parallel composition. For completeness of the validation process, traces were generated to confirm the equivalence of $i/o$ behaviour on SUT ports. As Uppaal does not output trace if no deadlock was found in the system, we used the following test query to generate traces:

```
E<> local_tester_1.trap[1] == true and local_tester_2.trap[1] == true
    and local_tester_3.trap[1] == true and local_tester_1.l10 and
    local_tester_2.l10 and local_tester_3.l10 and remote_tester.l10 and
     remote_tester.trap[1] == true
```

This query checks whether the distributed tester local testers and centralised tester remote tester have traversed the edges labelled with trap variable updates and reached the following location. The traces were generated by *verifyta* using the following query:

```
./verifyta -t2  model.xml query.txt 2> output.txt
```

The *-t2* allows us to access the fastest trace in *stderr*[1] and *2>* directs it to a text file. The traces were filtered by only leaving the observable $i/o$ communication on the SUT ports,

[1]https://www.computerhope.com/jargon/s/stderr.htm

i.e. intermediate coordination messages were filtered out. As the whole trace files are too large to include as a whole, we present segments of the traces. In Figure 11 Algorithm 1 bisimulation validation trace is depicted.

```
 1 local_tester_1.l1->local_tester_1._id90 { 1, i_1_[1]!, 1 }
 2 SUT._id15->SUT._id16 { 1, i_1_[1]?, 1 }
 3 remote_tester._id44->remote_tester.l2 { 1, m_i_1_[1]!, 1 }
 4 sut_copy._id15->sut_copy._id16 { 1, m_i_1_[1]?, 1 }
 5 local_tester_1.l2->local_tester_1._id87 { 1, i_1_[2]!, 1 }
 6 SUT._id16->SUT._id17 { 1, i_1_[2]?, 1 }
 7 remote_tester._id42->remote_tester.l3 { 1, m_i_1_[2]!, 1 }
 8 sut_copy._id16->sut_copy._id17 { 1, m_i_1_[2]?, 1 }
 9 SUT._id17->SUT._id18 { 1, o_2_[3]!, 1 }
10 local_tester_2.l3->local_tester_2._id69 { 1, o_2_[3]?, 1 }
11 sut_copy._id17->sut_copy._id18 { 1, m_o_2_[3]!, 1 }
12 remote_tester._id31->remote_tester.l4 { 1, m_o_2_[3]?, 1 }
13 local_tester_2.l4->local_tester_2._id66 { p == 0, i_2_[4]!, 1 }
14 SUT._id18->SUT._id19 { p == 0, i_2_[4]?, 1 }
15 remote_tester._id29->remote_tester.l5 { p == 0, m_i_2_[4]!, 1 }
16 sut_copy._id18->sut_copy._id19 { p == 0, m_i_2_[4]?, 1 }
17 SUT._id19->SUT._id20 { 1, o_2_[3]!, 1 }
18 local_tester_2.l5->local_tester_2._id48 { 1, o_2_[3]?, 1 }
19 sut_copy._id19->sut_copy._id20 { 1, m_o_2_[3]!, 1 }
20 remote_tester._id37->remote_tester.l6 { 1, m_o_2_[3]?, 1 }
```

Figure 11. *Algorithm* 1 *bisimulation validation trace segment.*

In the validation trace, the centralised testers has prefix $m\_$ added so the $i/o$ can be distinguished from that of distributed tester. Each line represents a transition with $i/o$ event, e.g. Figure 11 line 2 represents SUT transitioning from location with $id15$ to location with $id16$ and receiving input $i\_1\_[1]$. In Figure 12, segment of Algorithm 2 validation trace is depicted, where alike results can be seen.

```
local_tester_1.l1->local_tester_1._id80 { 1, i_1_[1]!, 1 }
SUT._id15->SUT._id16 { 1, i_1_[1]?, 1 }
remote_tester._id42->remote_tester.l2 { 1, m_i_1_[1]!, 1 }
sut_copy._id15->sut_copy._id16 { 1, m_i_1_[1]?, 1 }
local_tester_1.l2->local_tester_1._id81 { 1, i_1_[2]!, 1 }
SUT._id16->SUT._id17 { 1, i_1_[2]?, 1 }
remote_tester._id44->remote_tester.l3 { 1, m_i_1_[2]!, 1 }
sut_copy._id16->sut_copy._id17 { 1, m_i_1_[2]?, 1 }
SUT._id17->SUT._id18 { 1, o_2_[3]!, 1 }
local_tester_2.l3->local_tester_2._id64 { 1, o_2_[3]?, 1 }
sut_copy._id17->sut_copy._id18 { 1, m_o_2_[3]!, 1 }
remote_tester._id33->remote_tester.l4 { 1, m_o_2_[3]?, 1 }
local_tester_2.l4->local_tester_2._id54 { p == 0, i_2_[4]!, 1 }
SUT._id18->SUT._id19 { p == 0, i_2_[4]?, 1 }
remote_tester._id36->remote_tester.l5 { p == 0, m_i_2_[4]!, 1 }
sut_copy._id18->sut_copy._id19 { p == 0, m_i_2_[4]?, 1 }
SUT._id19->SUT._id20 { 1, o_2_[3]!, 1 }
local_tester_2.l5->local_tester_2._id55 { 1, o_2_[3]?, 1 }
sut_copy._id19->sut_copy._id20 { 1, m_o_2_[3]!, 1 }
remote_tester._id34->remote_tester.l6 { 1, m_o_2_[3]?, 1 }
```

Figure 12. *Algorithm* 2 *bisimulation validation trace segment.*

For each distributed tester $i/o$ pair, the equivalent centralised tester $i/o$ pairs with prefix $m\_$ follow subsequently. As a result, the traces confirm that the $i/o$ on the observable ports remains the same after distribution. Consequently, based on the results, we conclude that the requirement (e) has been satisfied.

# 5. Optimisation results

In this section, we compare the results of Algorithm 1 and Algorithm 2. The comparison comprises four parts: test model structural reduction, trace length, time cost, and synchronisation message total count difference. To illustrate the optimisation efficiency, we use the NTA depicted in Figure 16. The distributed tester generated by Algorithm 1 is depicted in Figure 21, and the distributed tester generated by Algorithm 2 is depicted in Figure 24.

## 5.1 Structural difference results

In Table 1, the structural difference is depicted (LT stands for local tester and POD for per cent of decrease). Comparison is between transition, synchronisation, and location count. Values are calculated in the per cent of decreased columns by dividing the difference column value by Algorithm 1 column value.

Table 1. *Table of structural difference.*

| | Algorithm 1 | Algorithm 2 | Difference | POD |
|---|---|---|---|---|
| LT 1 transitions | 25 | 15 | 10 | 40.00% |
| LT 2 transitions | 41 | 32 | 9 | 21.95% |
| LT 3 transitions | 29 | 22 | 7 | 24.14% |
| LT 1 synchronisations | 25 | 15 | 10 | 40.00% |
| LT 2 synchronisations | 40 | 31 | 9 | 22.50% |
| LT 3 synchronisations | 28 | 21 | 7 | 25.00% |
| LT 1 locations | 21 | 12 | 9 | 42.86% |
| LT 2 locations | 37 | 28 | 9 | 24.32% |
| LT 3 locations | 25 | 18 | 7 | 28.00% |
| Transitions total | 95 | 69 | 26 | 27.37% |
| Synchronisations total | 93 | 67 | 26 | 27.96% |
| Locations total | 83 | 58 | 25 | 30.12% |
| Total count (transitions and locations) | 178 | 127 | 51 | 28.65% |
| Bisimilarity | SATISFIED | SATISFIED | | |
| No deadlock | SATISFIED | SATSIFIED | | |

The model element total count regarding transitions and locations has decreased by over $28\%$. The greatest per cent of decrease is over $42\%$. The total count of elements regarding transitions and locations removed is $51$. As a result, the NTA model element total count has decreased considerably, and we consider it a satisfactory result. The following sub-section compares the difference in the trace lengths.

## 5.2 Trace length difference results

In this sub-section, we use the examples generated and run a specific test query using *verifyta* provided by the Uppaal toolbox for both algorithms to compare the resulting trace length difference. We also use the Unix *time*[1] command to record the time spent for each query. Optimising test trace lengths while preserving the same test coverage allows reducing computational resources used in lengthy test campaigns. The test query checks whether the distributed tester local testers have traversed the edges labelled with trap variable updates and reached the following location. The last trap is collected if the variable $p$ value is $30000$. The variable $p$ counts each moment where we either transition from location $l10$ to $l1$ or from location $l6$ to $l4$. The test query is as follows:

```
E<> local_tester_1.trap[1] == true and local_tester_2.trap[1] == true
    and local_tester_3.trap[1] == true and local_tester_1.l10 and
    local_tester_2.l10 and local_tester_3.l10
```

The resulting command used to conduct the experiment:

```
time ./verifyta -t2 -f prefix modelname.xml queryfile.txt
```

Where -t2 means generate diagnostic with the fastest trace and -f prefix means to save the file in form prefix-1.xtr.

The described query was applied to the distributed testers. In order to take the average time taken, we ran the query ten times for each distributed tester. In Figure 13, the resulting trace lengths are depicted. Algorithm 1 resulted in $5400006$ lines and Algorithm 2 in $4140006$ lines with a difference of $1260000$ and per cent of decrease of about $23.33\%$. The results are consistent with the theoretical expectation that the trace length decreases if structural complexity decreases.

[1]https://man7.org/linux/man-pages/man1/time.1.html

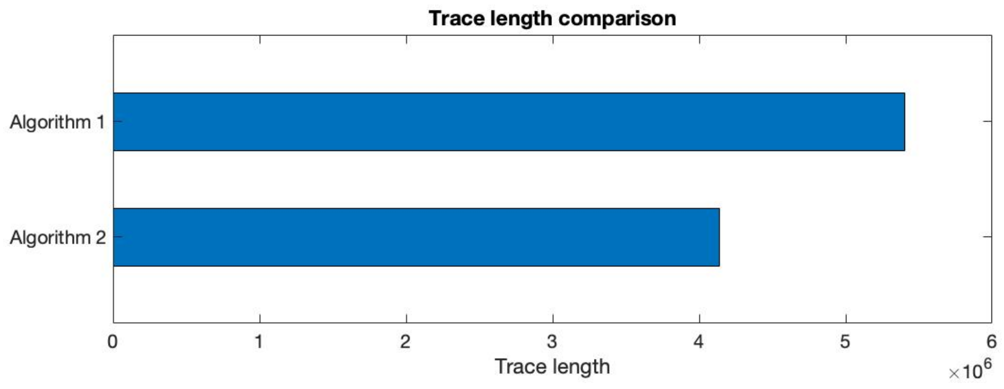


Figure 13. *Comparison of Algorithm* $1$ *and Algorithm* $2$ *trace lengths.*

In Figure 14, the time spent result is depicted. The experiment was conducted using a computer with the following specifications: processor 2.3 GHz Dual-Core Intel Core i5, memory 8 GB 2133 MHz LPDDR3, and graphics Intel Iris Plus Graphics 640 1536 MB. Algorithm 2 average time spent resulted in $2.485s$ and Algorithm 1 in $3.37s$ with a difference of $0.885s$ and per cent of decrease of about $26.26\%$. The result is consistent with the theoretical expectation that longer traces result in more time cost.

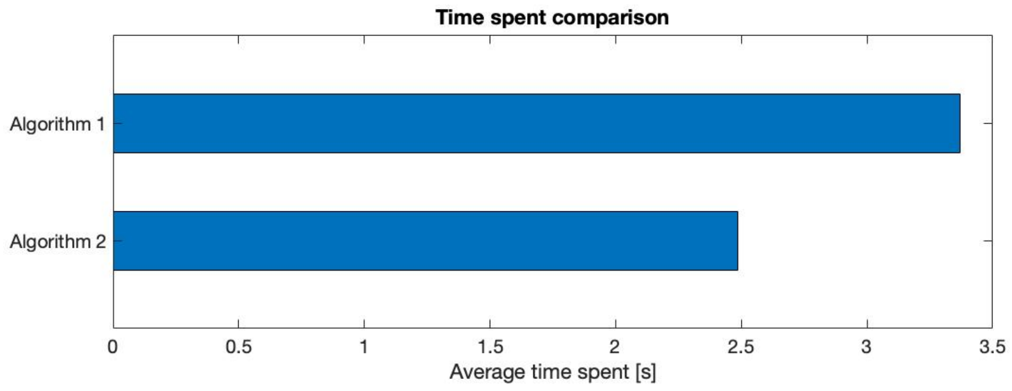


Figure 14. *Time spent comparison of Algorithm* $1$ *and Algorithm* $2$*.*

Consequently, the reduction of causal redundancies presented considerable gains in structural optimisation, trace length reduction, and test time reduction.

## 5.3 Communication overhead reduction results

The communication overhead reduction results are presented by counting the synchronisation messages sent between local testers regarding locally observed $i/o$ actions on the ports of the SUT. The NTA is configured to collect the last trap if the variable $p$ value is $1000$. In Figure 15 the results are depicted.

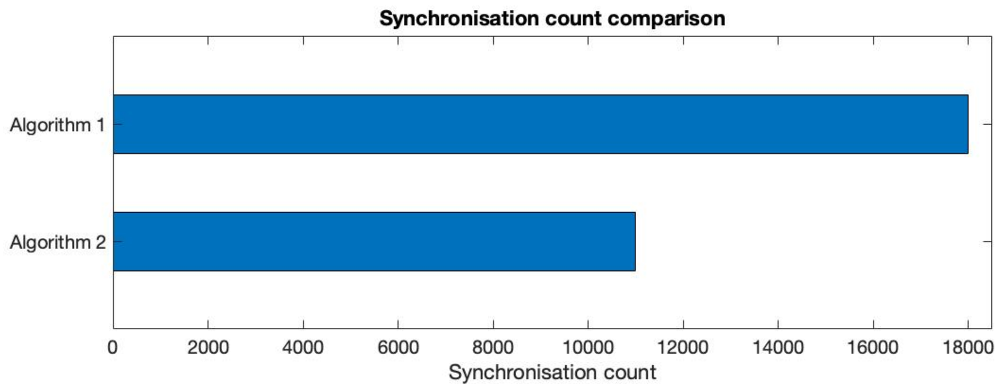


Figure 15. *Synchronisation count comparison of Algorithm* $1$ *and Algorithm* $2$.

Algorithm 1 resulted in $17998$ synchronisation messages and Algorithm 2 in $10998$. The difference is $7000$ with per cent of decrease of about $38.89\%$. As a result, the workload on the communication channels has reduced considerably.

# 6. Implementation and integration

In this section, we cover the implementation and integration. In section 6.1 we describe the technological decisions made regarding implementation and integration. In section 6.2 we cover the implementation of Algorithm 1, Algorithm 2, and automated validation. In section 6.3, we describe the algorithms implementation integration to the model-based testing environment BugBroom.

## 6.1 Back-end: technological decisions

***UTA XML parser***. The main component in implementing the algorithms is a parser for UTA XML files, which allows reading, modifying, and writing NTAs. For that purpose, PyUPPAAL[1] is used. PyUPPAAL is a Python library for manipulating Uppaal XML files and is maintained by Aalborg University academic personnel[2]. Further, compared to other parsers available, PyUPPAAL had been referenced in multiple articles related to model-based testing, which ensured the trustworthiness of the implementation.

***Programming language***. As PyUPPAAL is written in Python, the decision was to use Python to develop the algorithms. However, the library is written in Python 2. Hence, the library includes deprecated syntax as Python 2 is not maintained anymore. The deprecated syntax can cause issues in future perspectives. Therefore, we had to upgrade the deprecated parts to Python 3. Hence, the programming language used to implement the algorithms is Python 3[3].

***Model checker***. Uppaal verifier functionality can be accessed with *verifyta*[4] command-line utility. The tool *verifyta* allows us to automate the verification process of the models, e.g. confirming an NTA is deadlock-free by the "A[] not deadlock" query. This tool was chosen as we are using the Uppaal toolbox, which includes *verifyta* as one of the tools.

***Integration***. The integration is conducted by providing a *dockerised* REST (representational state transfer) API (application programming interface) in an Ubuntu server provided

[1]https://launchpad.net/pyuppaal
[2]https://homes.cs.aau.dk/ adavid/python/
[3]https://www.python.org
[4]https://docs.uppaal.org/toolsandapi/verifyta/

by TalTech. REST API is composed of Flask, Gunicorn, and Nginx. Flask[5] is a micro-framework used to develop web applications. As a micro-framework, Flask is lightweight with minimal dependencies, which gives developers more freedom in design choices. Flask comes with a development server that eases the development process but is not made for production. For production, we used Gunicorn[6] which is a popular Python WSGI HTTP server. In addition, Nginx[7], a high-performance web server, is used as a reverse proxy to handle incoming requests and direct them to the Gunicorn application server. By using Docker[8], an open-source containerisation platform, we can isolate the application from its environment, and Docker Compose[9] allows us to run multi-container Docker applications.

## 6.2 Back-end: implementation of the algorithms

***Input format***. In order to distinguish SUT $i/o$ communication between different loci and to read the remote tester and SUT, the following assumptions are made in terms of input. We give a standardised way to describe the input to automate the distributing process. Therefore, in addition to the requirements (a) and (b) presented in the section 3, we present more technical assumptions on the input.

1. Synchronisation is declared in the format $[i/o]\_[locus]\_[port]$.
2. In order to identify remote tester to be copied for each locus, the templates have specific names. Remote tester is declared as $"remote\_tester"$ and SUT is declared as $"SUT"$ or $"IUT"$. Both are case insensitive.
3. Synchronisation is declared as $chan$ globally. If not local then testers cannot communicate.
4. Variables are declared locally as the updates are local events.

An example of a centralised tester is depicted in Figure 16.

[5]https://flask.palletsprojects.com
[6]https://gunicorn.org
[7]https://www.nginx.com
[8]https://www.docker.com
[9]https://docs.docker.com/compose/

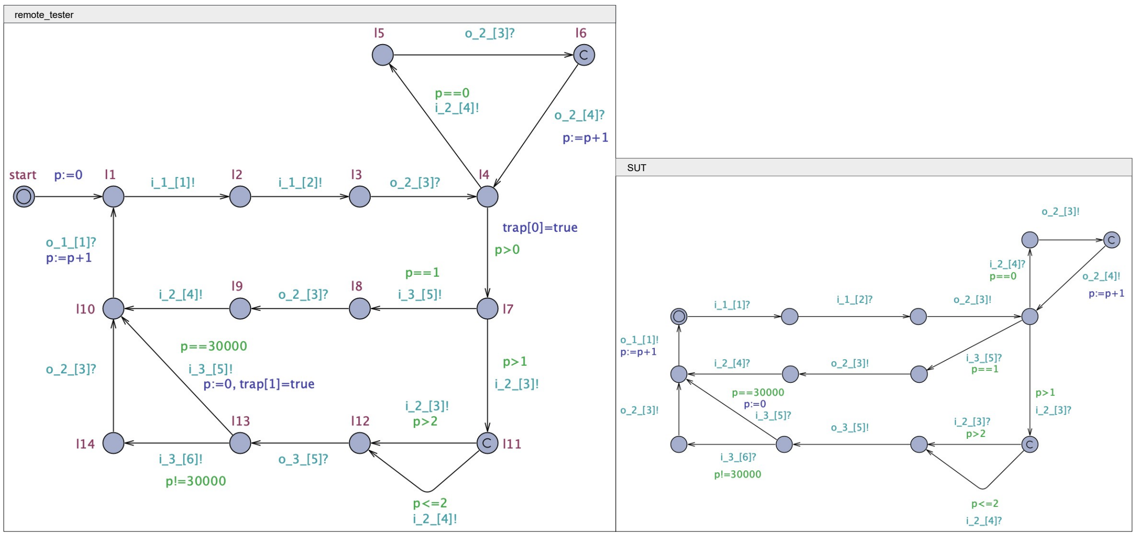


Figure 16. *Centralised tester.*

***Parsing XML to NTA***. Input is parsed by PyUPPAAL to one NTA. The core classes generated by the parser are depicted in Figure 17. Graphical elements such as nails and x and y coordinates are omitted as we are not focusing on the visual appeal of the models.

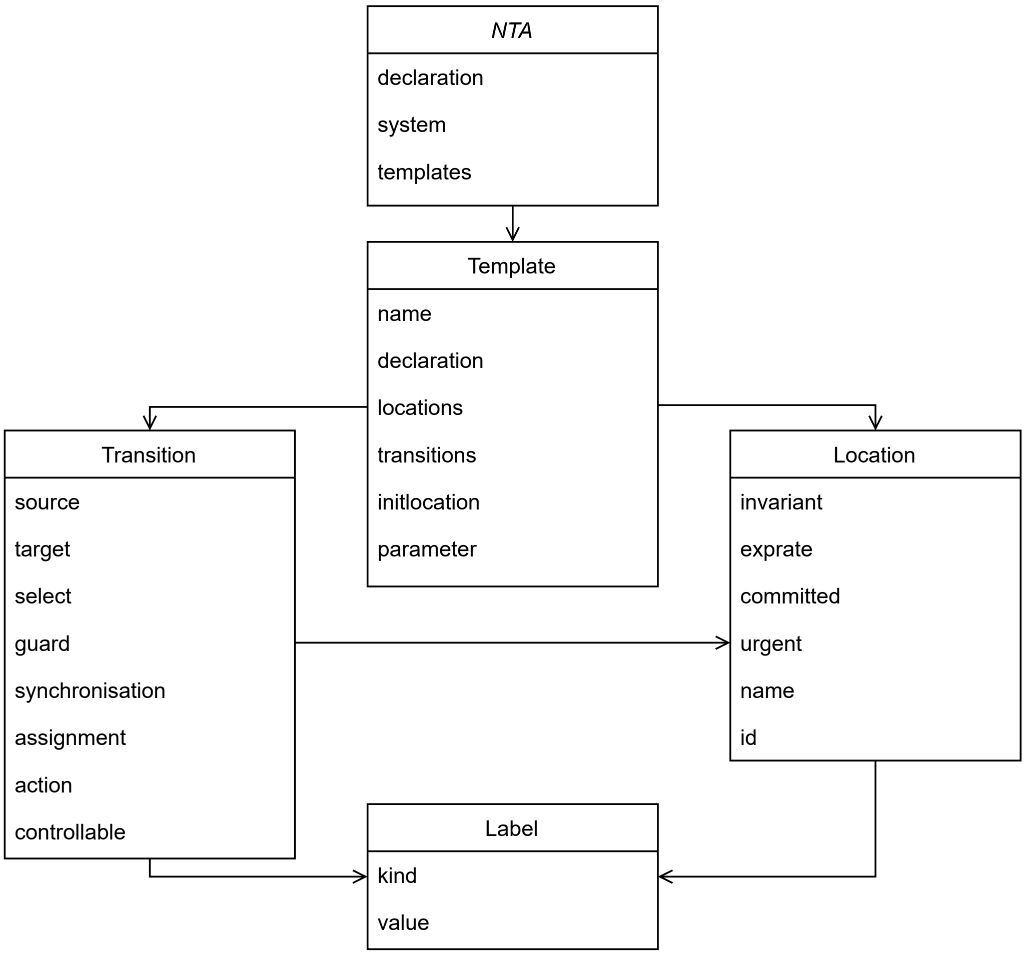


Figure 17. *PyUPPAAL NTA general architecture.*

***Tester distributor***. Tester distributor is implemented as a class $TesterDistributor$. Algorithm 1 can be accessed from $distribute\_tester\_alg\_1$ method and Algorithm 2 from $distribute\_tester\_alg\_2$ method. Both methods take in 5 arguments. The first two arguments define the path to the XML file and where to save the result. Two arguments specify if to generate the bisimulation validation specific models and what $verifyta$ version to use, which varies by OS. Lastly, a parameter is added to specify if to count model elements and write the result to $stdout$[10]. Tester distributor general workflow is depicted in Figure 18.

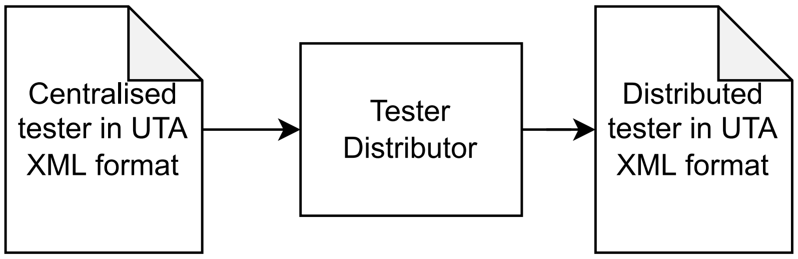

Centralised tester in UTA XML format
Tester Distributor
Distributed tester in UTA XML format

Figure 18. *Tester distributor workflow.*

In the sub-section 6.2.1 we describe the implementation of Algorithm 1 considering the augmentations and requirements described in section 3. In sub-section 6.2.2 we describe the implementation of Algorithm 2. Finally, in sub-section 6.2.3, we describe how automated validation was implemented.

### 6.2.1 Algorithm 1

Algorithm 1 implementation is composed of 5 main steps in total:

1. ***Reading NTA from XML***. Parsing input XML to class $NTA$, reading locus with corresponding ports, and copying remote tester template for each locus local tester.
2. ***Changing non-local synchronisation to receive***. Traversing over local tester transitions to detect all the non-local synchronisation values. All non-local synchronisation values are replaced with receive action.
3. ***Adding send synchronisation***. Using Step 2 collected synchronisation to be sent by locus to add synchronisation send actions to local testers.
4. ***Synchronising non-synchronisation transitions***. Detecting transitions that do not have synchronisation and synchronising these transitions between local testers.
5. ***Adding declarations and writing NTA to XML***. Adding declarations collected in the previous steps to NTA and writing NTA to XML.

[10]https://www.computerhope.com/jargon/s/stdout.htm

In step 1, PyUPPAAL parses the XML to NTA composed of two Templates: $remote\_tester$ and $SUT$. From $remote\_tester$ we are able to access all transitions and read the synchronisation values of the from $[i/o]\_[locus]\_[port]$. The described format allows us to collect each locus and corresponding ports. Next, template $remote\_tester$ is copied for each locus and renamed to $local\_tester\_[locus]$.

In step 2, detected non-local synchronisations values are reformatted to $c\_[i/o]\_[uid]\_[sender\ locus]\_[receiver\ locus]\_[port]$. Local tester synchronisation value is changed to receiving action "?" and for the $sender\ locus$ we add send action "!" to synchronisation to be sent from the corresponding transition. During step 2, we also collect the declarations to be added, as all of the synchronisations have to be declared. The synchronisation is made unique by incrementing the variable $uid$ value if found non-local synchronisation. The variable $uid$ value is initially $0$ and resets to $0$ for each locus. Adding uniqueness to synchronisations by using variable $uid$ is necessary for distributed testers where local testers are not copies of the remote tester. Hence, we describe it in the next sub-section under Algorithm 2 steps.

In step 3, the synchronisation to be sent is collected for each locus for each transition. The SUT $i/o$ remains as the first synchronisation action, and then the synchronisation is sent to other local testers by adding intermediate committed locations (augmentation (III) in section 3.1). Intermediate committed locations are created using class $Location$ with committed instance variable value declared as $True$. These locations are connected by creating class $Transition$ instances and changing the $source$ and $target$ locations accordingly. The general mechanism of sending synchronisation is depicted in Figure 19.

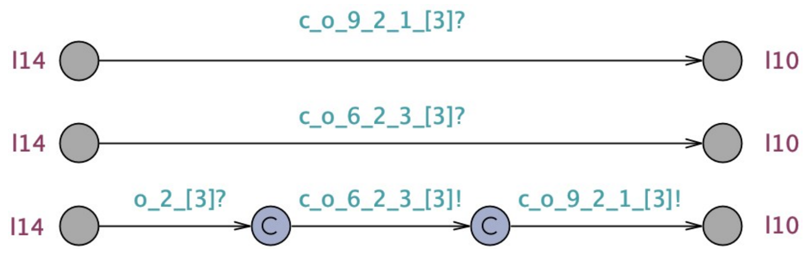


Figure 19. *General synchronisation sending mechanism.*

In step 4, we traverse over all the transitions for each locus and collect transitions that do not have synchronisation. Synchronisation on these transitions is added by selecting one local tester as the sender "!" and assigning the remaining local testers as receivers "?" (augmentation (I) in section 3.1). The format for the synchronisation is $c[uid]\_[sender\ locus]\_[receiver\ locus]$. The variable $uid$ is initially $0$ and incremented for each transition to make the transitions distinguishable. The synchronisation sending mechanism is identical to Step 3. An example of adding synchronisation between transitions with no synchronisation is depicted in Figure 20.

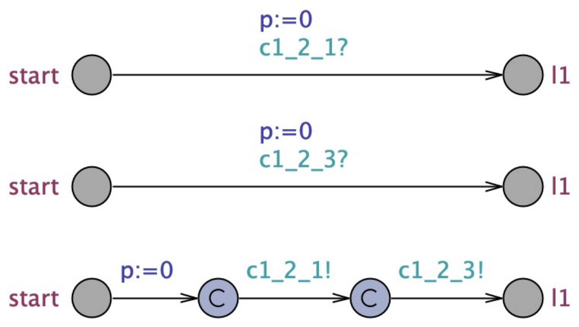


Figure 20. *Synchronisation sending mechanism for non-synchronisation transitions.*

In step 5, we declare local testers in the system declaration and add all synchronisation declarations globally as $chan$ (augmentation (II) in section 3.1). All the variables are locally declared for each local tester as we copied the $remote\_tester$ template for each locus. Finally, the distributed tester is written to XML by PyUPPAAL.

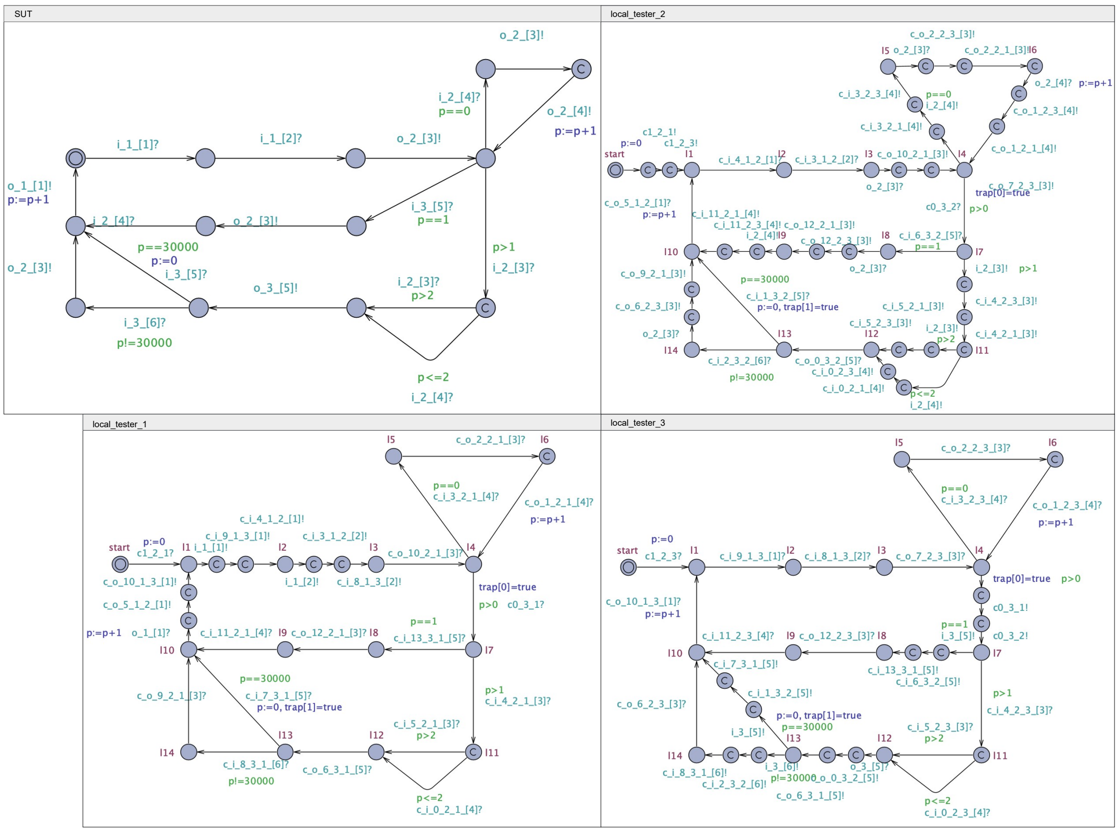


Figure 21. *Distributed tester generated by Algorithm* 1.

An example of the result of applying Algorithm 1 to NTA depicted in Figure 16 is shown in Figure 21.

### 6.2.2 Algorithm 2

As Algorithm 2 extends Algorithm 1, the steps of Algorithm 1 remain, but additional two steps are added: causal redundancy detection and causal redundancy reduction. Hence, Algorithm 2 is composed of 7 main steps in total:

1. ***Reading NTA from XML***. Parsing input XML to class $NTA$, reading locus with corresponding ports, and copying remote tester template for each locus local tester.
2. ***Redundancy detection***. Applying causal redundancy detection steps presented in section 3.2.
3. ***Adding receive synchronisation***. Traversing over the set $h2$ elements collected in Step 2 and adding synchronisation accordingly as presented in section 3.3.
4. ***Redundancy reduction by removing model elements***. Applying causal redundancy reduction steps presented in section 3.3.
5. ***Adding send synchronisation***. Using the synchronisation to be sent by locus collected in previous steps to add synchronisation send actions to local testers.
6. ***Synchronising non-synchronisation transitions***. Detecting transitions that do not have synchronisation and synchronising these transitions between local testers.
7. ***Adding declarations and writing NTA to xml***. Adding declarations collected in the previous steps to NTA and writing NTA to XML.

In step 2, redundancy detection from section 3.2 is directly transferred to the implementation. Hence, no further discussion is included here. In step 3, the synchronisation adding is presented in section 3.3. The synchronisation format and adding mechanism remain the same as in Algorithm 1 implementation. With Algorithm 2, we must consider that the local testers are no longer copies of the remote tester after redundancy reduction. The necessity of making synchronisation messages unique with variable $uid$ is depicted in Figure 22.

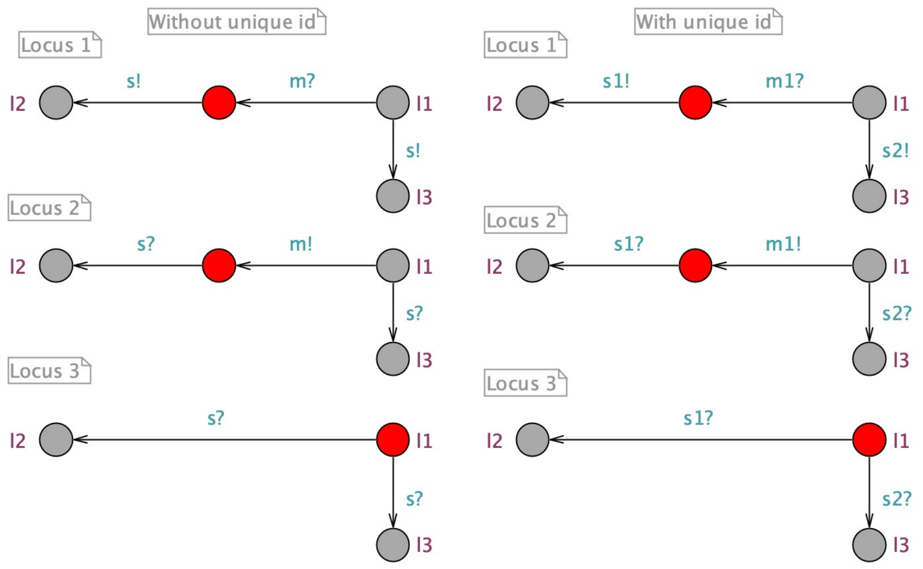


Figure 22. *Simplified NTA to show unique id importance.*

The segments of an NTA on the left-hand side are without uniqueness added to the synchronisations. For example, the $Locus\ 3$ can continue its test run by transitioning to location $l2$ or $l3$, whereas the correct location to be reached is $l2$. By adding uniqueness, which can be seen on the right-hand side, we ensure that the transitions are enabled correctly, i.e. the local test runs are synchronised correctly. In step 4, the redundancy reduction by removing elements presented in section 3.3 is directly transferred to the implementation. Hence, no further discussion is added here. In step 5, the synchronisation adding remains as for Algorithm 1 except if one locus appears more than once in synchronisation to be sent from transition. In this case, we add a transition between consecutive locations without adding a committed location. The described situation results from causal redundancy reduction where locations can be reached through different paths. In order to provide that the transitions are correctly enabled, we must ensure that all possible paths are considered. An example of this is depicted in Figure 23.

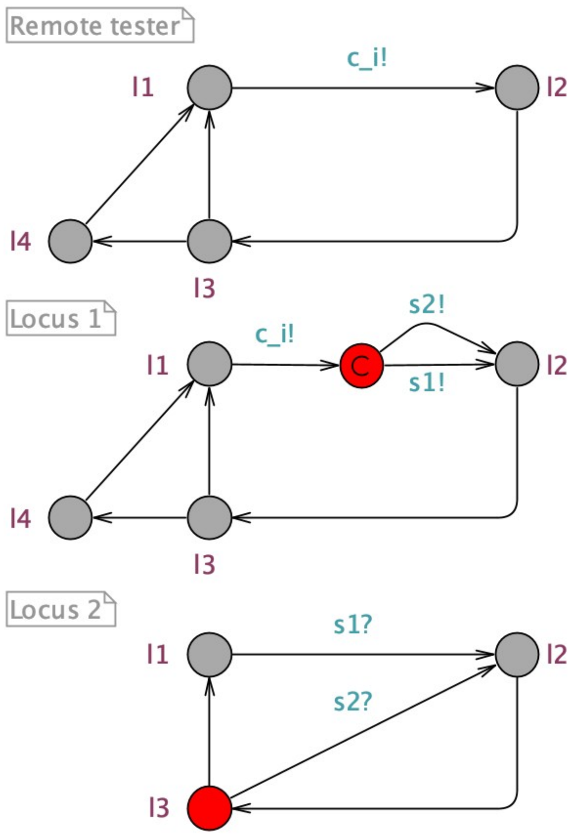


Figure 23. *Simplified NTA to show how multiple paths are handled.*

In this example, we can see that after redundancy reduction, $Locus$ 2 does not have location $l4$. The path from $l3$ through $l4$ to $l2$ is replaced with path from $l3$ to $l2$. In order to ensure that the transitions are correctly enabled, a synchronisation send-action $s2!$ is added to $Locus$ 1 by using a transition without an intermediate committed location. The described configuration allows us to account for different paths one location can be reached after redundancy reduction.

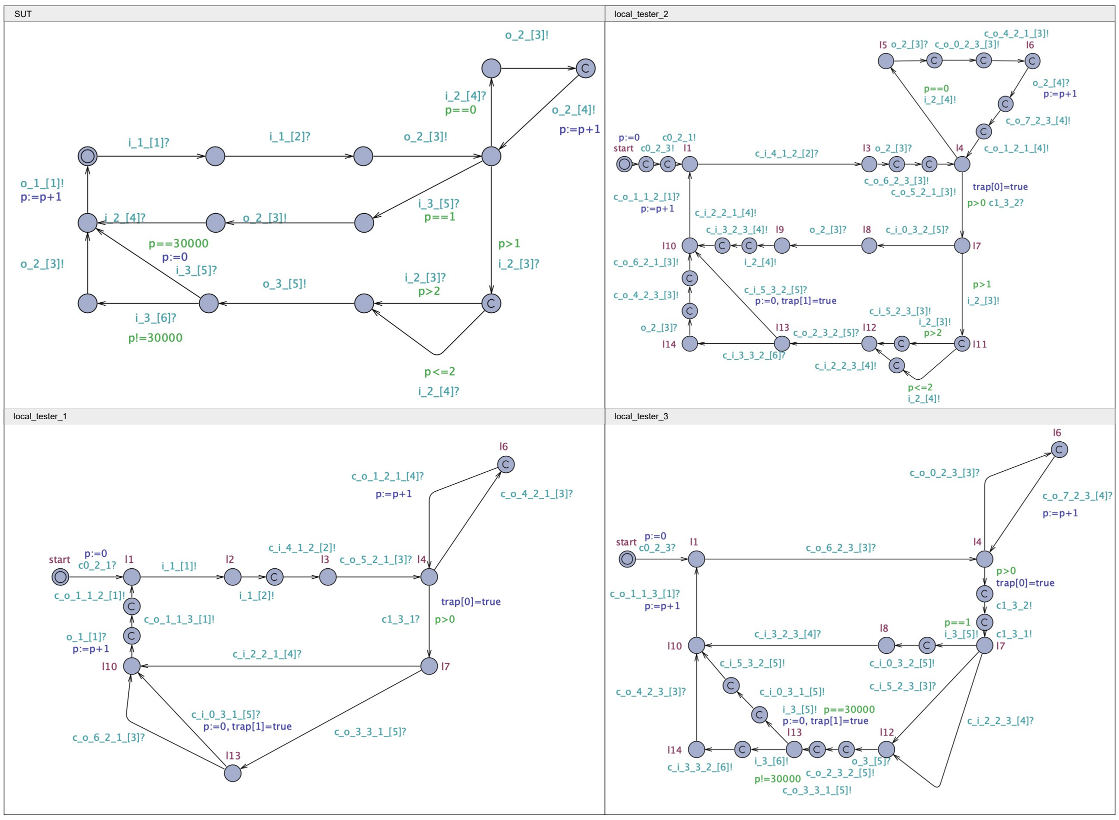


Figure 24. *Distributed tester generated by Algorithm* 2.

An example of the result of applying Algorithm 2 to NTA depicted in Figure 16 is shown in Figure 24.

### 6.2.3 Automated validation

In section 4, we presented the validation of the algorithms. We implemented the method used to check bisimulation between the centralised and distributed tester. An example of NTA to perform bisimulation relation validation between NTA in Figure 16 and Figure 24 is shown in Figure 25.

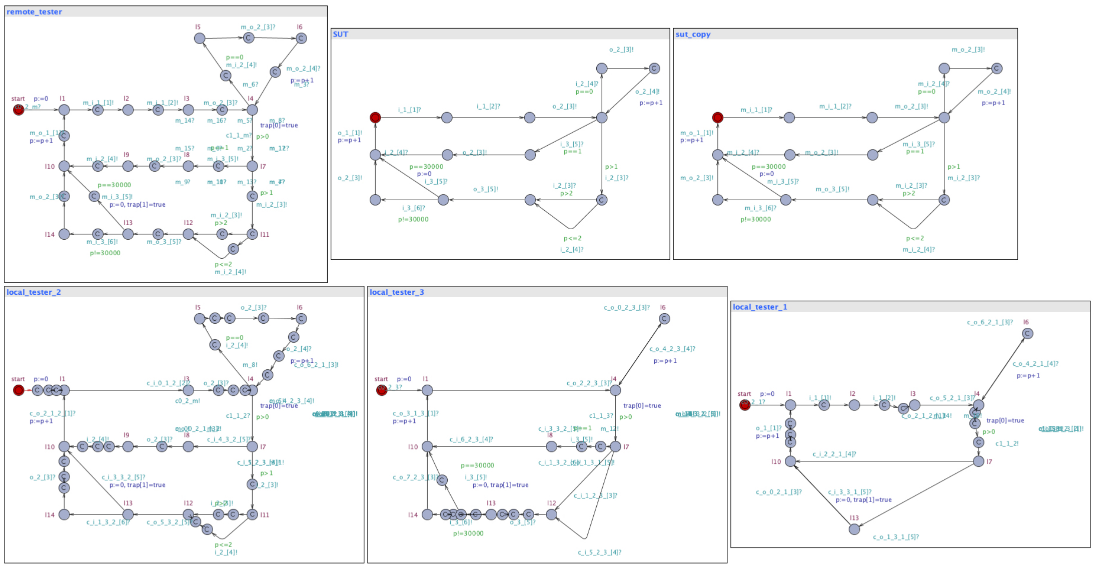


Figure 25. *Bisimulation validation NTA for Algorithm* 2.

The option to generate NTA for bisimulation check is added as a $boolean$ value for both algorithm methods. As we are using $verifyta$ which is OS-dependent, we also added a parameter to specify which $verifyta$ version to use: *"bin-Windows"*, *"bin-Darwin"*, and *"bin-Linux"*. Additionally, we added a deadlock freeness check for both algorithms in the workflow after the centralised tester is distributed, allowing for automated NTA verification.

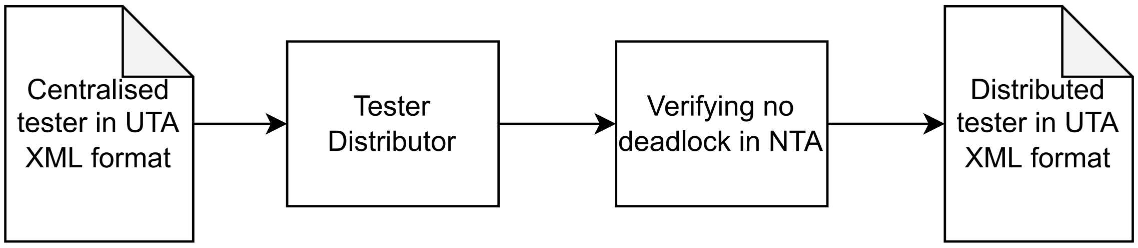

Centralised tester in UTA XML format
Tester Distributor
Verifying no deadlock in NTA
Distributed tester in UTA XML format

Figure 26. *Tester Distributor workflow with deadlock freeness check.*

The general workflow of tester distributor with deadlock freeness check is depicted in Figure 26.

## 6.3 BugBroom integration

BugBroom is a web-based modelling environment for timed automata. This application is intended to replace tools such as Uppaal and ECDAR[11]. Further, BugBroom is proposed

[11]https://www.ecdar.net

to have additional functionalities not provided by Uppaal nor ECDAR. BugBroom uses microservice-based architecture, i.e. all the external services run separately as web servers. As of writing this thesis, BugBroom is currently under development. The application's current architecture is depicted in Figure 27.

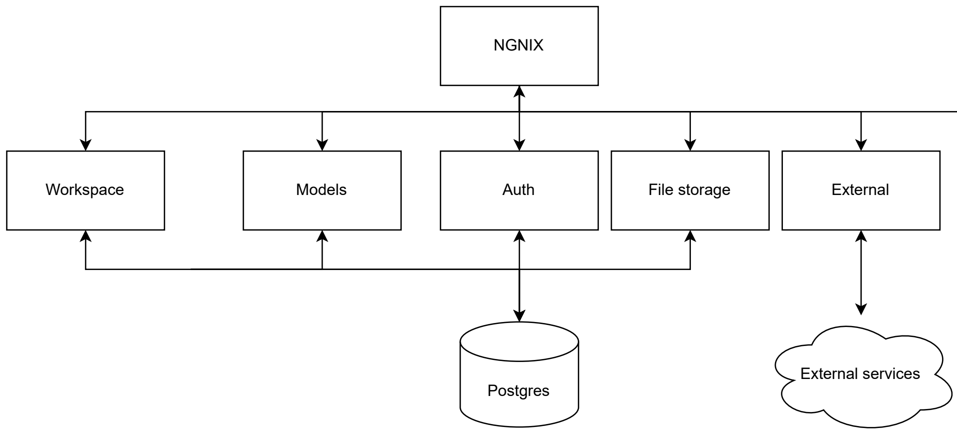
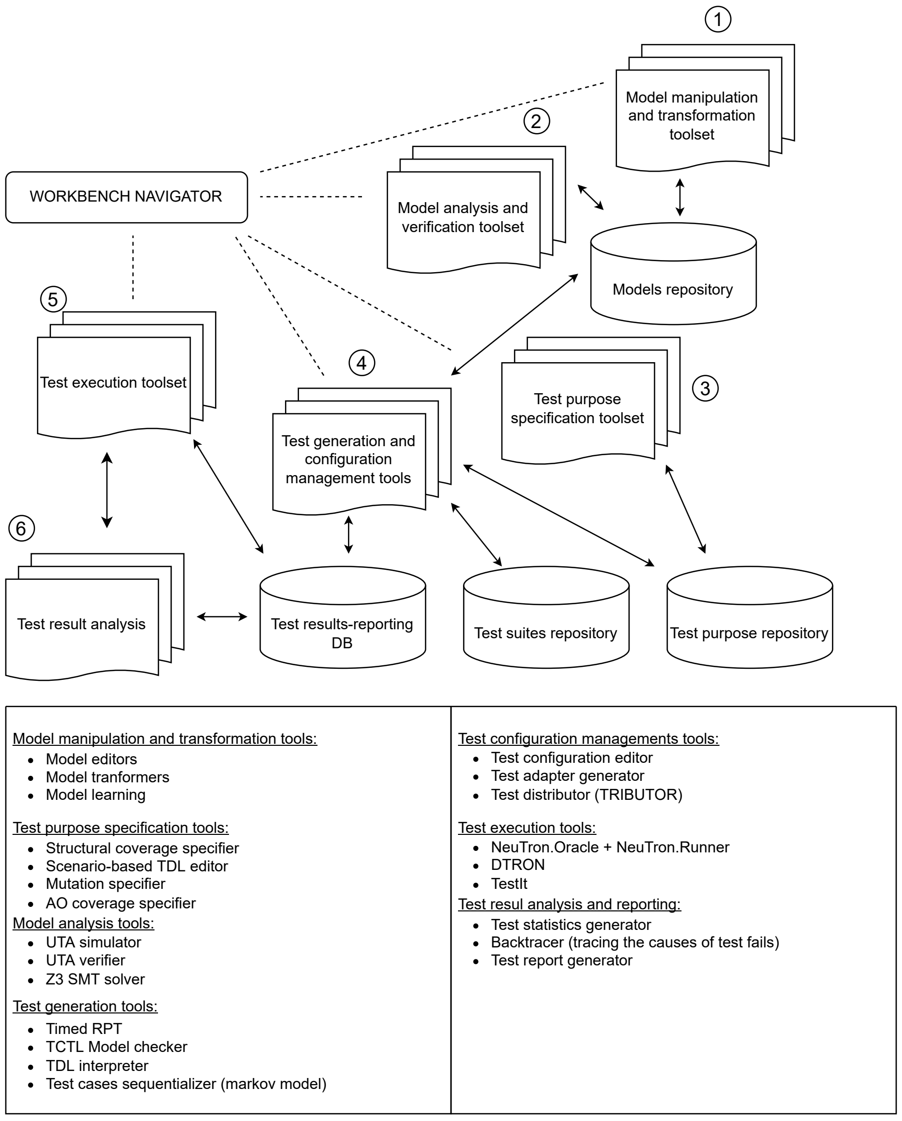


Figure 27. *Architecture of current BugBroom to show external services.*

The general architecture, depicted in Figure 28, was proposed by Jüri Vain in the year 2020. At this time, the environment was referred to as a workbench for model-based testing. The implementation to be integrated into BugBroom is the "Test distributor" component presented under general component 4 named "Test configuration management tools".


1
Model manipulation and transformation toolset
2
WORKBENCH NAVIGATOR
Model analysis and verification toolset
Models repository
5
Test execution toolset
4
Test generation and configuration management tools
Test purpose specification toolset
3
6
Test result analysis
Test results-reporting DB
Test suites repository
Test purpose repository

Model manipulation and transformation tools:
- Model editors
- Model tranformers
- Model learning

Test purpose specification tools:
- Structural coverage specifier
- Scenario-based TDL editor
- Mutation specifier
- AO coverage specifier

Model analysis tools:
- UTA simulator
- UTA verifier
- Z3 SMT solver

Test generation tools:
- Timed RPT
- TCTL Model checker
- TDL interpreter
- Test cases sequentializer (markov model)

Test configuration managements tools:
- Test configuration editor
- Test adapter generator
- Test distributor (TRIBUTOR)

Test execution tools:
- NeuTron.Oracle + NeuTron.Runner
- DTRON
- TestIt

Test resul analysis and reporting:
- Test statistics generator
- Backtracer (tracing the causes of test fails)
- Test report generator

Figure 28. *BugBroom workbench.*

In section 6.3.1, we cover how the tester distributor as an external service is added to the front-end. Finally, in section 6.3.2, we cover how the implementation of the algorithms was integrated into the system BugBroom as an external microservice.

## 6.3.1 Front-end

Front-end development consisted of two steps: creating views to interact with the tester distributor and adding routing for external service for back-end communication. As the goal is to integrate the implementation into the environment, we use the provided technologies. The development is conducted by using mainly TypeScript[12] and Vue.js[13] which are used

[12]https://www.typescriptlang.org
[13]https://vuejs.org

by the environment.

Tester distributor is added under *Testing tools* which can be accessed from the navigation bar (herein navbar). We created a component called distributor. The tester distributor view can be accessed by selecting *Distributor* from the *Testing tools* drop-down menu. The *Distributor* view displays the details of the current project selected in the *Projects* view. The details include *Project name*, *Description*, and systems as *Test configuration*. In order to distribute a tester, two buttons are added: *Distribute tester Algorithm* 1 and *Distribute tester Algorithm* 2. The result is depicted in Figure 29.

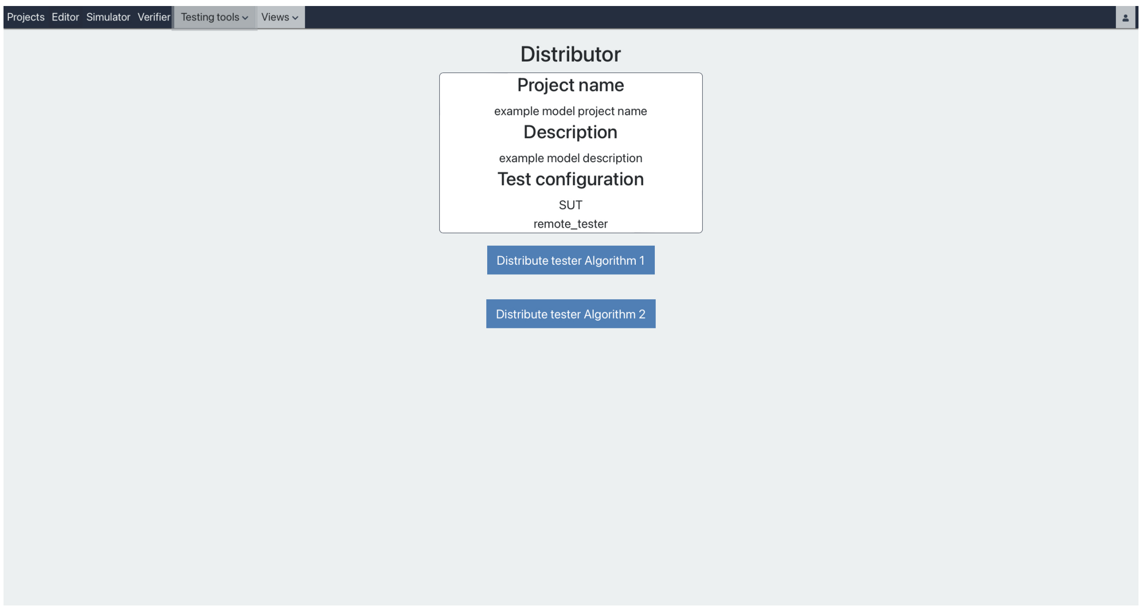
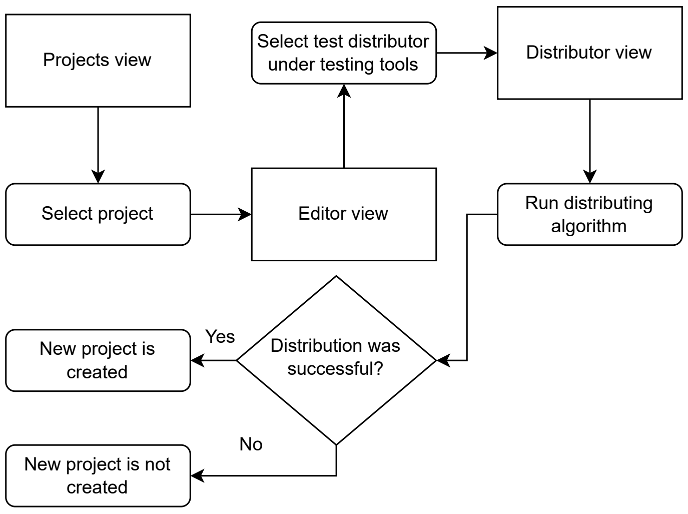


Figure 29. *Distributor view in BugBroom.*

Next step is to add external service workflow for distributing the tester. The workflow is depicted in the Figure 30.


Projects view
Select test distributor under testing tools
Distributor view
Select project
Editor view
Run distributing algorithm
Yes
New project is created
Distribution was successful?
No
New project is not created
Figure 30. *Front-end workflow.*

First, the user selects a project in the main view, navigates to the tester distributor and runs the selected distributing algorithm. If the distribution was successful, a message notifying the successful event is displayed, and a new project is created for the user. The response XML is written into JSON (JavaScript Object Notation) by *parserService* provided by the environment, and a new project is created by *projectsService* provided by the environment. If the distribution was unsuccessful, e.g. synchronisation format does not conform with the input assumptions, the user is notified by a message about the unsuccessful event, and we do not create a new project. The service to distribute a tester is added under external services.

### 6.3.2 Back-end and front-end communication

As BugBroom follows microservice-based architecture principles, the implementation is provided as an external microservice to the environment. The tester distributor microservice is a REST API running in docker containers installed in an Ubuntu server provided by TalTech. The general architecture is depicted in Figure 31.

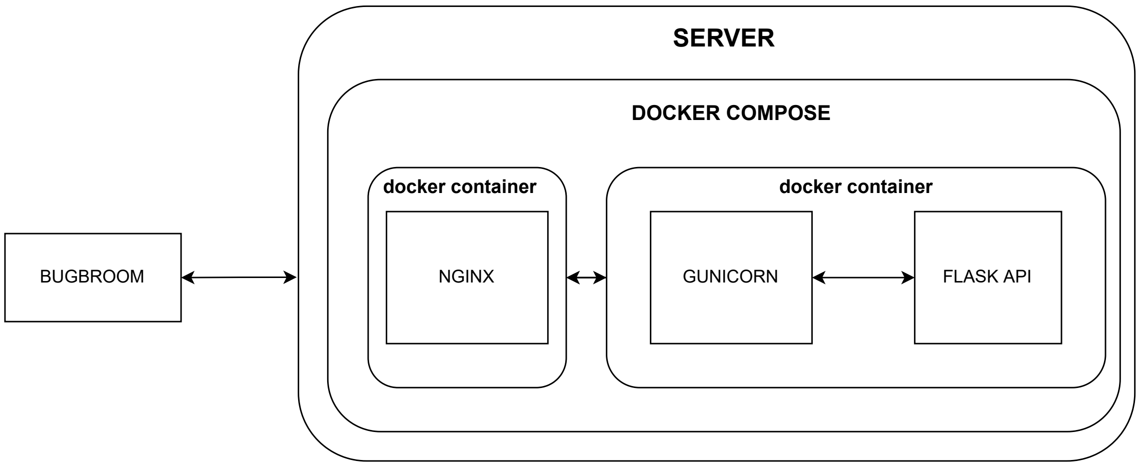


Figure 31. *Integration architecture overview.*

Nginx is set up as a reverse proxy to handle client requests on port 80 by redirecting them to localhost port 5000, where the Flask wrapped with Gunicorn REST API is running. The test distributor REST API comprises two endpoints: Algorithm 1 and Algorithm 2. Both endpoints follow the same workflow as depicted in Figure 32.

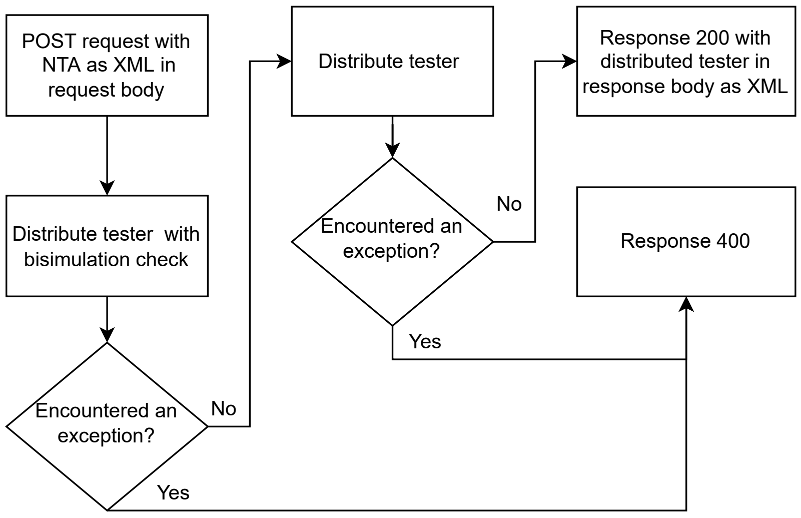

POST request with NTA as XML in request body
Distribute tester with bisimulation check
Encountered an exception?
No
Yes
Distribute tester
Encountered an exception?
No
Yes
Response 200 with distributed tester in response body as XML
Response 400

Figure 32. *General workflow of the API.*

Front-end sends a POST request with XML in the request body. The model is read from the body, and the tester is distributed with the bisimulation check to provide automated validation. As PyUPPAAL reads files from a path, we used Python standard library module *tempfile*[14] which allows creating temporary files that are automatically deleted after closing. Next, the tester is distributed and checked for deadlock freeness. Response 200 is sent to the front-end with XML in the response body if no exceptions were encountered. If an

[14]https://docs.python.org/3/library/tempfile.html

exception occurs, response 400 is sent with text containing possible exception reason, e.g. synchronisation is in an incorrect format.

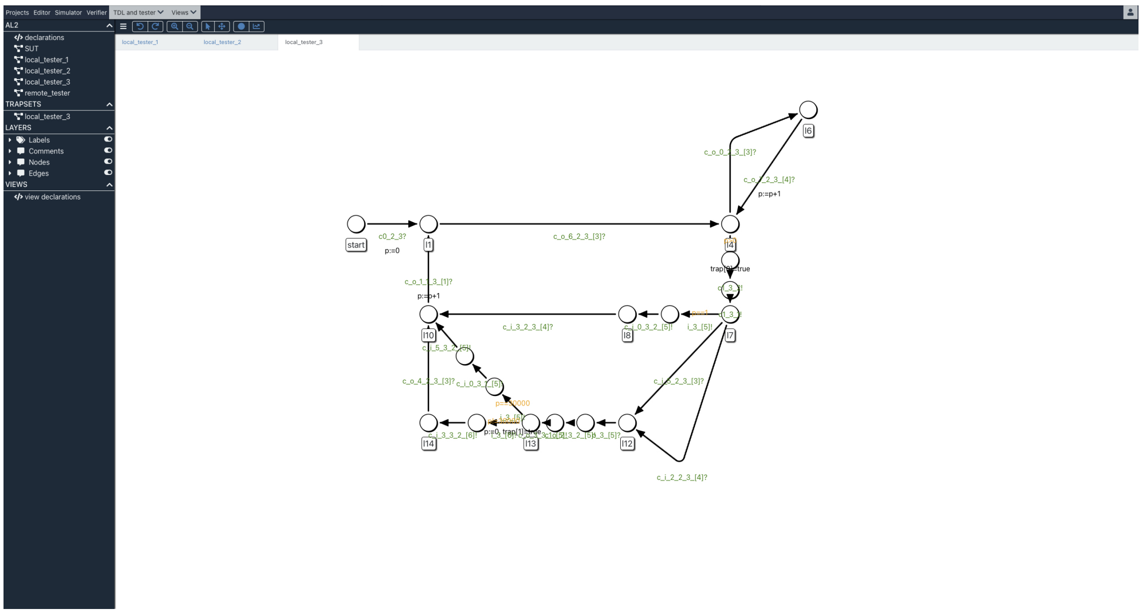


Figure 33. *Algorithm* 2 *distributed tester local tester in BugBroom.*

An example of a local tester of a distributed tester in BugBroom is depicted in Figure 33

# 7. Summary

In this thesis, we provided an approach to optimise distributed tester local components generated by a state-of-the art algorithm referred in this thesis as Algorithm 1 from paper [5]. Prior to optimisation, Algorithm 1 was augmented to fulfil the requirement (d) and to specify the synchronisation mechanism used. The correctness of the optimised algorithm was validated by checking the bisimulation relation between the centralised tester and fully distributed tester with local testers concerning the observable $i/o$ actions on the ports of the SUT. We further confirmed the validation result by comparing segments of test traces.

The optimisation resulted in over $28\%$ of decrease in structural complexity of the test model, over $23\%$ of decrease in trace length, over $26\%$ of decrease in average time spent, and over $38\%$ of decrease in total synchronisation message count based on the example used. The original distribution algorithm and its optimising extension were implemented as a proof of concept, with automated validation included, and made accessible in the BugBroom testing environment. Therefore, the goals of this thesis were achieved.

This thesis focused on one optimisation approach. As for future work, we propose an approach to extend the optimisation by doing calculations on variables selectively in local testers and sending the result to other testers that the value directly influences. For example, one local tester is doing the calculations, and other local testers omit the intermediate calculations by receiving the latest resulting value from a local tester selected to do the calculations. Another approach would be to extend the optimisation by keeping only the guards, updates, and invariants in the local tester if they directly influence its behaviour. For example, if the local tester does not contain any guards or invariants related to some variable, then the updates for this variable can be omitted. Furthermore, as stated in requirement (a), we limited this thesis to two template test models. Future work may extend this to more templates. An essential step in future work would be integrating the algorithms into a real testing environment such as DTRON. Furthermore, the visual presentation of the models was not the focus of this thesis. Future work would be to create an algorithm to adjust the visual layout of the models to be more human-readable. Lastly, the integration can be improved to include more advanced options for the tester distributor, display more specified error messages, and give information about the tool.

# References

[1] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[2] Jüri Vain, Evelin Halling, Gert Kanter, Aivo Anier, and Deepak Pal. "Model-based testing of real-time distributed systems". In: *International Baltic Conference on Databases and Information Systems*. Springer. 2016, pp. 272–286.

[3] Alexandre David, Kim G Larsen, Marius Mikučionis, Omer L Nguena Timo, and Antoine Rollet. "Remote testing of timed specifications". In: *IFIP International Conference on Testing Software and Systems*. Springer. 2013, pp. 65–81.

[4] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software testing, verification and reliability* 22.5 (2012), pp. 297–312.

[5] Jüri Vain, Evelin Halling, Gert Kanter, Aivo Anier, and Deepak Pal. "Automatic Distribution of Local Testers for Testing Distributed Systems." In: *DB&IS (Selected Papers)*. 2016, pp. 297–310.

[6] Jüri Vain, Marko Kääramees, and Maili Markvardt. "Online testing of nondeterministic systems with the reactive planning tester". In: *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*. IGI Global, 2012, pp. 113–150.

[7] Jüri Vain, Gert Kanter, and Seshadhri Srinivasan. "Model based testing of distributed time critical systems". In: *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2017, pp. 99–105. DOI: 10.1109/ICRITO.2017.8342406.

[8] Anders Hessel. "Model-based test case selection and generation for real-time systems". PhD thesis. Uppsala University, 2006.

[9] Aivo Anier, Jüri Vain, and Leonidas Tsiopoulos. "DTRON: a tool for distributed model-based testing of time critical applications." In: *Proceedings of the Estonian Academy of Sciences* 66.1 (2017).

[10] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.

[11] Gerd Behrmann, Alexandre David, and Kim G Larsen. “A tutorial on Uppaal 4.0”. In: *Department of computer science, Aalborg university* (2006).

[12] Robin Milner. “Communication and concurrency”. In: *PHI Series in computer science*. 1989.

[13] Johan Bengtsson and Wang Yi. “Timed automata: Semantics, algorithms and tools”. In: *Advanced Course on Petri Nets*. Springer. 2003, pp. 87–124.

# Appendices

## Appendix 1 - Non-exclusive licence for reproduction and publication of a graduation thesis[1]

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.