TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Naveed Ahmed Alizai 182468IVSM

BISIMULATION VERIFICATION OF UPPAAL MODELS

Master's Thesis

Supervisor: Jüri Vain

Ph.D.

Leonidas Tsiopoulos

Ph.D.

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Naveed Ahmed Alizai 182468IVSM

UPPAALI MUDELITE BISIMILAARSUSE VERIFITSEERIMINE

Magistritöö

Juhendaja: Jüri Vain

Ph.D.

Leonidas Tsiopoulos

Ph.D.

Tallinn 2020

Author's declaration of originality

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

Author: Naveed Ahmed Alizai

04.08.2020

Abstract

In this thesis, we aim to elaborate the bisimulation verification method for UPPAAL timed automata models. We explore bisimulation verification suggested [1] to build a parallel synchronous composition between timed automata models using channel semantics in the UPPAAL modeling tool and then verify the deadlock freeness of the composition by model checking. We also investigate other bisimulation verification methods and compare them with our approach.

According to the definition of bisimulation relation in computer science, if state-based transition systems simulate the behavior of each other, they are said to be bisimilar. Efficient checking of bisimulation between models plays key role in several branches of computer science, in model transformations, model checking and others. Therefore, bisimulation checking capability could be a valuable functionality of any model-based development and analysis tool. Though UPPAAL tool is widely used for research and industry projects, in its current version there is no bisimulation support implemented in the tool yet.

The goal of this thesis is to implement an algorithm that transforms the models to the form that enables verifying their bisimilarity using UPPAAL model checker without extending it.

This thesis is written in English and is 53 pages long, including 6 chapters, 24 figures and 1 appendix.

Annotatsioon

UPPAALi mudelite bisimilaarsuse verifitseerimine

Käesoelva magistritöö eesmärk on töötada välja UPPAALi ajaga automaatite bisimulatsiooni verifitseerimise meetod. Töös on lähtutud artiklis [1] publitseeritud bisimulatsiooni verifitseerimise ideest, mis seisneb võrreldavate mudelite sünkroonse paralleelkompositsiooni konstrueerimises. UPPAALi automaatide kanalite semantika tagab bisimilaarsete olekute ja siirete sünkroonse täitmise. See omakorda võimaldab tuvastada mudelkontrolliga tupikute tekkimise juhul, kui mudelid ei ole bisimilaarsed. Magistritöös uuritakse samuti teisi varem publitseeritud bisimulatsiooni verifitseerimise võtteid ja võrreldakse neid käesoleva töö meetodiga.

Bisimulatsiooni relatsiooni all mõistetakse arvutiteaduses seost olekumudelite vahel, kus kumbki mudel on simulatsiooni suhtes teise mudeliga. Bisimulatsiooni efektiivne verifitseerimine omab võtmetähtsust paljudes arvutiteaduse valdkondades nagu näiteks mudelite teisendused, mudelkontroll, mudeli-põhine testimine jt. Seetõttu osutub bisimulatsiooni automaatse verifitseerimise mudeli-põhise arendus- ja analüüsi tööriista juures oluliseks funktsionaalsuseks. Ehkki UPPAAL on laialdaselt kasutuses nii teadus-, kui ka äriprojektides, puudub UPPAALis mudelite bisimulatsiooni automaatse verifitseerimise tugi.

Käesoleva magistritöö eesmärk on realiseerida algoritm, mis teisendab võrreldavad mudelid sünkroonse paralleelkompositisiooni kujule, mis võimaldab bisimulatsiooni verifitseerida UPPPAALis juba olemasoleva mudelkontrolli vahendiga.

Magistritöö on kirjutatud inglise keeles, 53 leheküljel, sisaldab 6 peatükki, 24 joonist ja 1 lisa.

List of abbreviations and terms

LTS	Labelled Transition System
TA	Timed Automata
SUT	System Under Test
AOT	Aspect-Oriented Testing
HRM	Home Rehabilitation System
TCTL	Timed Computation Tree Logic
PTP	Parallel Timer Process
BDD	Binary Decision Diagram
GUI	Graphical User Interface
SCS	Safety Critical System
IDE	Integrate Development Environment
OS	Operating System
UI	User Interface
RAM	Random Access Memory

Table of contents

List of figures

1 Introduction

Formal methods play an essential role in the requirements engineering, design and development phases of complex systems. They are mostly used for mathematical proving and validation of the requirements' correctness of system design during the development cycle. Formal methods help in early identification of probable faults that may arise in later stages and facilitating reliability, dependability, performance and cost improvements of safety critical systems (SCS) [18] [19] [20].

Formal methods incorporate formal specification, synthesis and verification. Formal specifications are constructed using formal modelling languages or other mathematical notations which have rigorous formal semantics. Formal verification is applied through tools mainly categorised as equivalence checkers, model checkers and theorem provers to verify set of properties on the specification [22].

Formal specification languages for reactive systems have common semantic basis in the form of labelled-transition systems (LTS). LTS allow specifying the interaction of system under test (SUT) with environment or set of environment components under specified constraints. LTS is also used in the simulation to visualize the behavior of the system. In the simulation processes the model generates traces of possible execution paths of the system. These traces can be used to debug any existing deadlocks or other critical issues that the verifier reports by checking the properties specified during the specification and verification of the system [23].

Safety critical real-time systems such as traffic control systems for aeroplanes and trains, have reactive behaviour in contrast to office systems that are "transformational". Reactive systems constantly interact with the environment that may involve humans for decision making. Human and natural/technical environment involvement typically widens the range of predicted and unpredicted behavioural scenarios. Due to the hazardous nature of many reactive system applications and the need to prevent any potential faults, formal methods are applied in different forms throughout their development cycle. For instance, when modelling the constraints in requirements of the system, Constrain Diagrams (CD)

can be used and during the design the state-transition models can be used to verify design specification [21].

Depending on the system specification's complexity, its formal modeling can be challenging. To make sure that the model represents the system behavior accurately, or to compare if two models represent the same behavior various techniques can be applied. Bisimulation verification is one of the techniques used widely to identify the behavioral equivalence of models, to compare their performance and usability characteristics, but also to answer the questions related to addressing state space explosion problems [4].

The modeling, simulation, and verification of real-time systems with continuous time constraints can be done using the UPPAAL tool where timed automata models have semantics expressed in LTS. UPPAAL is a powerful tool that enables for the modeling, simulation, and verification of extended timed automata networks. A timed automaton is a finite LTS-based system supporting real-valued model clocks and is used in modeling real-time systems where these clocks represent time constraints. UPPAAL has a Java-based GUI that provides user interface to three main components: a model editor, simulator and model checker. An editor enables constructing and editing models supporting it with on-the-fly syntax check. A simulator allows to select next available transitions manually or run the model with selecting next executable transitions in random mode. It also visualizes the traces and model variable values on every step. A verifier allows specifying properties for model checking and generates diagnostic traces for the properties that are not satisfied. Another component added in the newer versions of UPPAAL (Version 4.1.24 onwards) called a concrete-simulator provides more advanced options for execution visualization. There are some popular extensions of UPPAAL namely Verifyta [5], TRON [12] [13], TIGA [14], PORT [15], and ECDAR [16 17] with enhanced capabilities for modeling, testing, and verification of TA models [5].

In the current stable version of UPPAAL (4.1.24), the feature to automatically synchronize two automata models for bisimulation verification is not yet available. In more realistic scenarios, the models that are supposed to be synchronized for bisimulation check operate independently in their separate environments. In UPPAAL, only one model can be opened at once. Thus, an algorithm is required to be developed either in form of tool extension or another tool that merges such two independent models as one system without affecting their original behavior in order to support bisimulation verification.

1.1 Research Goal

The main goals of this thesis are:

- implementing the general idea of bisimulation verification proposed in [1] by developing a constructive algorithm for transforming the comparable models in the form which is relevant for bisimulation checking using the existing UPPAAL model checker;

- implementing the algorithm and evaluating its complexity in terms of model structural units to be added for bisimulation checking and measuring experimentally the time and memory usage needed for model checking the construction.

- validating the approach by experimenting it on UPPAAL standard case studies such as train-gate and coffee-machine.

1.2 Unit of Study

In UPPAAL, the same specification can be modeled differently. Generally, it is desirable to have the ability to check behavioral equivalence between models when model structures are different. Another reason for checking behavioral equivalence is proving conservation of model properties after applying model transformations. To check the equivalence of two models, one option is to statically analyze the models. Another option is to execute the models and identify the simulation relation by comparing the enabledness of corresponding transitions dynamically. Currently, there is no publically available tool support for checking bisimilarity between two timed automata models designed in UPPAAL. Although it is possible to make model comparison manually but it is not practical because it is very time consuming and error prone process. This motivates the need for an algorithm implementation of the bisimulation checking that can be applied without introducing changes in the existing and very efficient UPPAAL verifier.

In most of the theorems, specifically related to state space minimization problems [4], bisimulation is applied on the state level in the models. Since the thesis topic has grown out from the practical need to compare models used in model-based testing we apply in the thesis the theory of bisimulation on two potentially bisimilar TA models. Since the

general context of bisimulation study in the thesis is model-based testing (MBT) we restrict ourselves with models that represent Closed World View. Therefore, the models to be compared have assumingly two partitions: System Under Test (SUT) and its environment. Both partitions in the model can contain multiple parameterized templates and their instances called UPPAAL TA processes. To apply the bisimulation verification approach proposed in [1] these models have to be composed into a single model and their respective components need to be synchronized based on their observable (at partitions interfaces) input-output actions. Moreover, due to the fact that UPPAAL model checker (used in this approach) performs dynamic forward reachability state space exploration, the synchronous composition of models should guarantee that non-deterministic choices in both models are synchronized as well. These considerations form the key requirements to be followed when elaborating the pattern for synchronous composition of comparable models.

1.3 Research Questions

1. What does it mean for two models to be bisimilar?
2. How to automate bisimulation verification check for UPPAAL models?
3. How to implement the approach suggested in [1]?
4. How to evaluate the applicability of our algorithm using UPPAAL standard case studies?

1.4 Organization of the Thesis

This thesis is structured in the following manner:

- **Related Work and Background:** The chapter highlights the work in the field of bisimulation in formal methods and the concept of bisimulation and its formal definition for UPPAAL timed automata. Moreover, a summary of UPPAAL tool is provided.
- **Model Transformation:** This chapter describes our approach of transforming models to apply UPPAAL model checker without modifying it for bisimulation checking. We also study the transformation complexity and related to it model checking complexity.
- **Case Studies:** This chapter contains the bisimulation checking results on the UPPAAL TA case-studies of train-gate and coffee-machine.
- **Analysis:** This chapter elaborates how we started constructing an algorithm and what challenges we faced and how we addressed those challenges.
- **Conclusion:** This chapter summarizes the outcomes of this thesis research and highlights possible future work.

2 Related Work and Background

2.1 Related Work

Automatic Distribution of Local Testers for Testing Distributed Systems [1]: This paper presents an approach for automated model-based testing in low-latency systems. It describes the criticality of time in large geographic cyber-physical systems where a single system is distributed over the network. Usually the most difficult part in the design of such systems is timing because a very small increase in the latency may have severe consequences.

Since different parts of these systems are deployed in different geographical locations, manual testing is not possible. And automated testing becomes more complex due to the timing constraints. This paper extends remote Δ-testing method and propose a distributed test architecture by deploying local testers. This approach exorbitantly reduces the overall reaction time.

The correctness of the algorithm has been verified by applying bisimulation equivalence between the centralized automated tester and the distributed tester. A parallel synchronous composition has been constructed using generator-acceptor automata and a deadlock property have been verified that if composed models are non-blocking then models are non-blocking separately as well.

We will use bisimulation equivalence introduced in this paper as a starting point for our thesis. And based on it construct a generic approach for the synchronization of two models for bisimulation verification.

Aspect-Oriented Model-Based Testing [24]: In chapter 5 “Analysis and Validation of AOT”, the bisimulation checking has been considered in the context of aspect-oriented modelling using UPPAAL TA. An approach from [1] on how to check the equivalence of aspect oriented and non-aspect oriented models in UPPAAL has been presented. It was shown that aspect-oriented and non-aspect oriented models can be compared based on weak bisimulation equivalence relation with respect to observable at system interfaces input/output actions. Main steps to perform bisimulation verification on the UPPAAL models was introduced.

The motivation behind bisimulation checking was to demonstrate that an aspect-oriented UPPAAL model has quantitative and qualitative advantages over bisimilar to it non-aspect-oriented model. The conclusions were based on the measurable effort spent on specifying test coverage criteria and the effort of test generation from models, also concerning the complexity results of the TCTL (Timed Computation Tree Logic) model checking.

While in [24] it has been shown how aspect-oriented and non-aspect oriented models can be compared for the bisimilarity between them, this approach can also be applied to perform the bisimulation verification on any other models that are built in UPPAAL verification tool for timed automata systems.

Lectures on Concurrency and Petri Nets [6]: The "Timed Automata: Semantics, Algorithms and Tools" chapter of this lecture notes provides a tutorial for semantics and algorithms of verification tools for time automata models. As the name describes it contains in-depth knowledge of semantical and algorithmic aspects of verification and the data structures verification tools like UPPAAL.

The main areas discussed in this chapter are the Introduction to Timed Automata, Symbolic Semantics and Verification, Algorithms and Data Structures, and finally modeling in UPPAAL. In the Timed Automata section, there is a brief explanation of the decidability of timed bisimulation and its definition. The symmetrical binary-relation satisfying condition helps to understand the concept of bisimilarity on the transition systems.

Better Verification through Symmetry [7]: In this article, the problem of state explosion of finite-state systems has been addressed and significant improvements are recorded. In finite-state systems, even simple systems produce a large number of states making it difficult for the algorithms to handle. A new datatype scalar-set is introduced which is used to detect symmetries easily and based on the implied symmetries by scalar-sets, a verifier generates a reduced state space, on-the-fly.

According to the article "The algorithm has been implemented and evaluated on several realistic high-level designs. Memory requirements were reduced by amounts ranging from 83% to over 99%, with speedups ranging from 65% to 98%". Furthermore, the article defines the symmetry and proof that symmetry-based state-space reduction does

not affect the results by providing proof to handle the challenges discussed in the Description Language section and also verifying this technique on different synchronization algorithms.

As per the practical results section of this article, their symmetry-based detection algorithm is the extension of a MurΦ verification system which supports non-determinism and concurrency through iterated guarded commands. MurΦ compiler generates a C++ program based on the description of a finite-state asynchronous concurrent system and the program checks the properties on the system by enumerating through all reachable states. In our case, we construct a similar algorithm purely based on UPPAAL semantics to synchronize two models for bisimulation check and being able to verify the properties using UPPAAL verifier.

Decidability of Bisimulation Equivalences for Parallel Timer Processes [8]: In this paper, an abstract model of parallel timer processes (PTPs), allowing specification of temporal quantitative constraints on the behavior of real-time systems is introduced. The parallel timer processes are defined in a dense time domain and can model both concurrent (with delay intervals overlapping on the time axis) and infinite behavior."

It is proved that both strong and weak bisimulation equivalence problems of parallel timer processes that are abstracted from internal actions are decidable. It is also proved that if the PTP model is additionally provided with memory cells for moving the timer value information along the time axis, the bisimulation equivalence problems become undecidable. The paper provides detailed theorems for parallel timer processes, and comprehensive definitions and theorems on bisimulation, as well as proofs on deciding of bisimulations.

Bisimulation and Model Checking [9]: In the effort to reduce the state explosion problem and reduce state-space before model checking in the transition systems, the bisimulation minimization technique is used to perform equivalence checks on the systems. What makes this research unique from the other researches based on bisimulation minimization techniques to reduce state-space is that it is in the context of verifying invariant properties.

This paper considers three bisimulation minimization algorithms. A conventional model checker based on backward reachability is compared to a model checker for invariant

properties which is produced on-the-fly for each of these algorithms. The bisimulation minimization algorithms used are the following:

1. *PT (Paige-Tarjan): Has the best provable worst-case running time of traditional bisimulation minimization algorithms (those that stabilize both reachable and unreachable blocks).*

2. *BFH (Bouajjani-Fernandez-Halbwachs): Improves on PT by choosing only reachable blocks to stabilize on each iteration; however, it may stabilize an unreachable block that was split off from the reachable block being stabilized in the current iteration.*

3. *LY (Lee-Yannak): Improves on BFH by never stabilizing an unreachable block.*

This paper concludes that while bisimulation has many important roles in the verification of transition systems and helps to collapse infinite-state to finite-state, reducing also state explosion problems, both theoretical and experimental comparisons of this research conclude that the bisimulation minimization is not viable in model checking in the context of verifying invariance properties.

Sigref – A Symbolic Bisimulation Tool Box [10]: Sigref is a tool that is used to compute the most popular bisimulation using a uniform signature-based approach. It is implemented symbolically using binary decision diagrams (BDDs) to handle large transition systems. The main goal of this framework is to handle state space efficiently for the systems that are too large for other tools that require an explicit state-space description.

The goal of this research paper was to handle the famous state-space explosion problems in state verifications in a large state-space system. In their approach, they have combined BDDs (Binary Decision Diagrams) in which extremely large state-space can be presented in a very compact symbolic representation with ~~b~~isimulation minimization.

The preliminaries section of this paper contains the basic notations and important types of bisimulation i.e. strong, weak, progressing, orthogonal, delay, safety, branching, and n-bisimulation. In the next sections of the paper, signatures of some of the bisimulation types are discussed, and, furthermore, the symbolic computations and representation of data and experimental results are provided.

In the process of constructing an algorithm for bisimulation verification of UPPAAL timed automata models, the illustrations in this paper helped in better understanding of the definition of bisimulation and its types conceptually.

Model Checking Timed Automata [11]: This chapter from "Modelling and Verification of Real-Time Systems" discusses the timed automata model and its main characteristics. One important characteristic of the timed automata model is real-valued variables called clocks and these clocks increase synchronously with the time. Using clock values, it can be decided when the transition can be performed by associating clocks to the guards and similarly update operations to be performed associated with the clocks.

Furthermore, this chapter focuses on other characteristics of timed automata like syntax and semantics, decision procedure for checking reachability, timed languages (branching-time and linear-time temporal logics). Some other extensions of timed automata are diagonal clock constraints, additive clock constraints, internal actions, updates of clocks, linear hybrid automata. Subclasses of timed automata and algorithms for timed verification are also discussed in the later sections. Lastly, a brief introduction to the UPPAAL model-checking tool for timed systems has been provided.

A Tutorial on UPPAAL [5]: This paper serves as a tutorial for the UPPAAL model-checking and verification tool. The paper provides an introduction to the implemented timed automata in the tool and also describes the user interface and instructions for using the tool. It also contains information about modeling patterns and reference examples.

UPPAAL is a toolbox for verification of real-time systems jointly developed by Uppsala University and Aalborg University. The model-checker of UPPAAL is based on timed automata and modeling language supporting bounded integer variables and urgency. The query language of UPPAAL is a subset of TCTL (Time Computation Tree Logic) which is used to specify properties to be checked.

Understanding of time in UPPAAL is explained in detail and an overview of the GUI is provided. The user interface of UPPAAL is composed of three main sections: the editor, the simulator, and the verifier. Each section is briefly explained and also two detailed examples are provided, i.e., Train-Gate example and Fischer's protocol. Lastly, comprehensive information about different modeling patterns including Intent, Motivation, Structure, and Sample is provided.

This paper is extremely helpful in understanding the UPPAAL tool, the timed automata, modeling, and querying specifications in UPPAAL.

To conclude the overview of related work, most of the approaches except that of [1] [8] and [24] rely on model execution based on-the-fly comparison of enabled state transitions. The method introduced in [1] and used in [24] is the only one that makes direct use of UPPAAL TA syntax and semantics by augmenting bisimilarity checks adding synchronization channels between observable i/o actions. This constraint guarantees that if the assumed bisimilar actions have different enabling conditions then it introduces deadlock conditions where one of the transitions has more limiting enabling condition than other. The deadlocks are detectable then by standard query of UPPAAL model checker.

2.2 Background and Relevant Tools

2.2.1 Theoretical Background

In the study of concurrent processes in computer science, the contribution of bisimulation is of utmost importance. The bisimulation equality or bisimilarity is a broadly studied form of behavioral equality of concurrent processes. In the proving of behavioral and trace equivalences of the processes, state-space minimization, and converting infinite-state space systems to finite, bisimulation is considered the optimum approach that can be implied. Bisimulation is also widely used in the theory of verification, compiler construction, program analysis, functional languages, object-oriented languages, type theory, databases, and many more [2].

In the sequel, we provide the definition of bisimulation relation using Labelled Transition System (LTS) notation. LTSs are commonly used for defining operational semantics in automata theory.

LTS is a tuple $\left(W, Act, \left\{\overset{a}{\rightarrow} : a \in Act\right\}\right)$ where W is set of states or processes, Act set of labels, and for each label a, a transition relation $\overset{a}{\rightarrow}$ on W is defined. Considering s,t to range over W and μ to range over the labels in Act we can write $s \overset{\mu}{\rightarrow} t$ when $(s,t) \in \overset{\mu}{\rightarrow}$; and call t a μ-derivative of s. With this information, bisimulation is a binary relation R on the states of an LTS whenever $s_1 \, R \, s_2$:

- *For all* s_1' *with* $s_1 \xrightarrow{\mu} s_1'$, *there is* s_2' *such that* $s_2 \xrightarrow{\mu} s_2'$ *and* $s_1' R s_2'$

- *The converse on the transitions is emanating from* s_2

Bisimilarity, written $\sim$, is the union of all bisimulations; thus $s \sim t$ holds if there is a bisimulation R with $s\ R\ t$ [2].

This definition of bisimilarity indicates that it is the finest equivalence on symmetrical systems and the bisimulation is very strong. And to be able to use bisimulation on a broader range of theories, different forms of bisimulation have been introduced [2, 3]. One prominent form is weak bisimulation which states that the systems imitate the behavior of each other but one system may have internal silent actions that make the converse rule false.

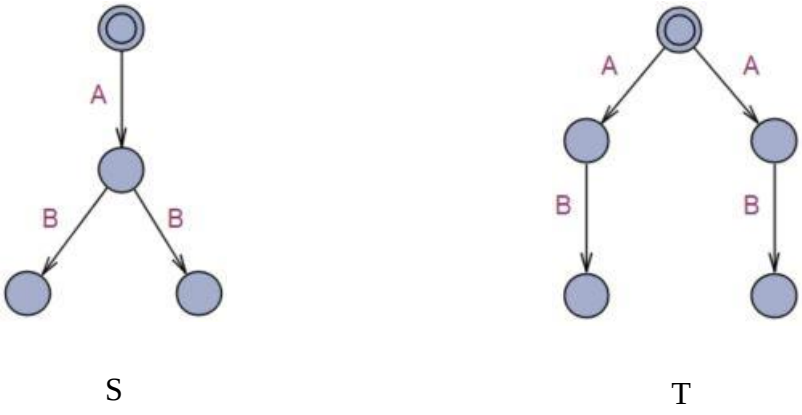


Figure 1. Bisimilar Models

In the above model S, when the transition A is taken then the choice is A(B+B) and in model T, the choice is taken at the initial state i.e. AB + AB (See Figure 1). Since both decisions lead to the same transition A(B+B) = AB + AB then these two models are considered behavioral and trace equivalent bisimilar.

Y Z

Figure 2. Non-bisimilar Models

By applying the same logic as above to compare model Y and model Z in Figure 2, at first glance the models look bisimilar because the behavior of both models is the same i.e. A(B+C) in Y and AB + AC in Z where A(B+C) = AB + AC. But if we closely observe the traces, they are not the same because in model Y even after taking transition A we still have a choice to either choose B or C but this is not true in the case of model Z where we can only proceed with B or C. Hence Y and Z are not considered bisimilar.

2.2.2 Relevant Tools

2.2.2.1 UPPAAL Modeling and Verification Tool

UPPAAL allows designing the system and interpret its operational semantics in terms of LTS. The modeling language provides numerous features like templates, declarations (C++ like code support) and expressions. Also, UPPAAL contains State formulae, Safety properties, and Liveness properties and provides a strong verification engine that supports writing and verifying properties on the system. We will use UPPAAL tool to validate the existing approach for bisimilarity check and construct our generic pattern using different models while verifying the correctness and measuring the time and space complexities of it.

2.2.2.1.1 UPPAAL Modeling Preliminaries

Templates: Templates act as processes and help to model system in smaller parts. Templates can have their local declarations and allows passing arguments to them.

Constants: Constants can only be integers in UPPAAL.

Bounded Integers: Bounded integers have a range from Min to Max value.

Binary Synchronizations: Channels in UPPAAL are declared with keyword type *chan*. Channels are used to communicate between templates/processes. When a channel is sending signal *"!"* is used with channel name and to receive a signal *"?"* is used. Another type of channel is broadcast channels in which all the receiving ends will receive the signal at once.

Clock: Clocks are real-valued integers and they are measured in time units. Clocks are used in invariants and guard conditions. Clock values can only be reset.

Invariants: Invariants are conditions or conjunction of conditions on locations. Invariants allow only integers variables, constants and clocks in the conditions.

Locations: Locations are the states of the system. Locations can be of three main types i. Initial location: From this location the process starts, ii. Urgent location: With urgent property, time is not allowed to pass when the system is in this state, iii. Committed location: This is the most restricted location type. When the location is committed, there must be at least one outgoing transition enabled and the transition is taken immediately and at this time, no other inter-leavings are allowed in the system.

Transitions: Transitions are the edges between the locations/states of the system. Transitions allow expressions of four different types i. Select: Allows to select a value from a range non-deterministically, ii: Guard: Guard is a condition or disjunction of conditions if added on the transition then it must be satisfied to proceed with the transition, iii. Synchronization: Channels are used as labels in synchronization to send or receive signals, iv. Update: Update allows initializations and update of variables and function calls on the transition.

Arrays: Arrays are allowed only for few types (i.e. integers, clocks, channels) in UPPAAL and they help in parameterizing data types.

Initializers: Initializers are used to initialize variables in declarations and on template transitions.

2.2.2.1.2 UPPAAL Query Language Preliminaries

Assume *s* and *t* state formulas, UPPAAL verifier supports the following queries:

- E< > *s*: there exists a path where *s* eventually holds.
- A[ ] *s*: for all paths *s* always holds.
- E[ ] *s*: there exists a path where *s* always holds.
- A< > *s*: for all paths *s* will eventually hold.
- *s*--> *t*: whenever s holds *t* will eventually hold.

we will be using the following property *"*A[ ] *not deadlock"* to verify our models. Which states that for all paths there is no deadlock in the system.

3 Model Transformation

3.1 Approach

The baseline for our approach is originally illustrated in [1] “Automatic Distribution of Local Testers for Testing Distributed Systems” and used in Ph.D. thesis [24] “Aspect-Oriented Model-Based Testing” chapter 5 “Analysis and Validation of AOT Method” section 5.2.1 “Bisimulation Verification”. This approach suggests for weak bisimulation checking to synchronize the observable moves of two models say M and M’ by identifying the input and output actions between environment and the controllers and add auxiliary channels to-the original IO actions in a way that the controller in M synchronize with the controller in M’ and both interacting with common environment. To construct this parallel synchronous composition in UPPAAL without affecting the original behavior of the models, a committed location is added after the original transition with an auxiliary channel on the transition after the committed location. With the committed location, the atomicity of the transition is preserved and an auxiliary channel on the transition after the committed location will act as a synchronizing link.

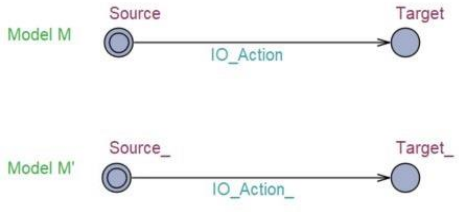


Figure 3. Original Models

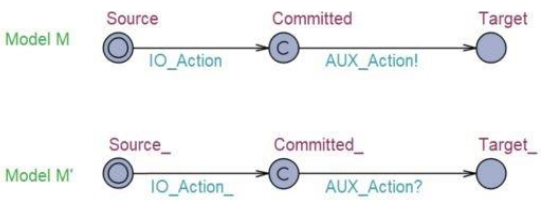


Figure 4. Synchronized Models

According to this theory, models are synchronized using a pattern based on input/output actions of the system. This pattern is applied and proven working for two possibly bisimilar controllers with a single common input environment and deterministic models. But we are considering separate, possibly bisimilar, environment for each controller. In more realistic scenarios, individual systems contain separate input environment or set of environments of their own. Considering the above theory from Figures 3 and 4, to synchronize models having separate environment for each controller, if we split the I/O action and add committed location with auxiliary channel after on both sending and receiving sides of the action then due to the committed location, there will be a deadlock (See Figure 5) because one model will send the original I/O signal and immediately try

to send the synchronization signal using auxiliary channel. Due to the restricted behavior of committed location, it expects that the receiving end of the signal is ready to receive and synchronize but the other model is yet to execute its I/O action.

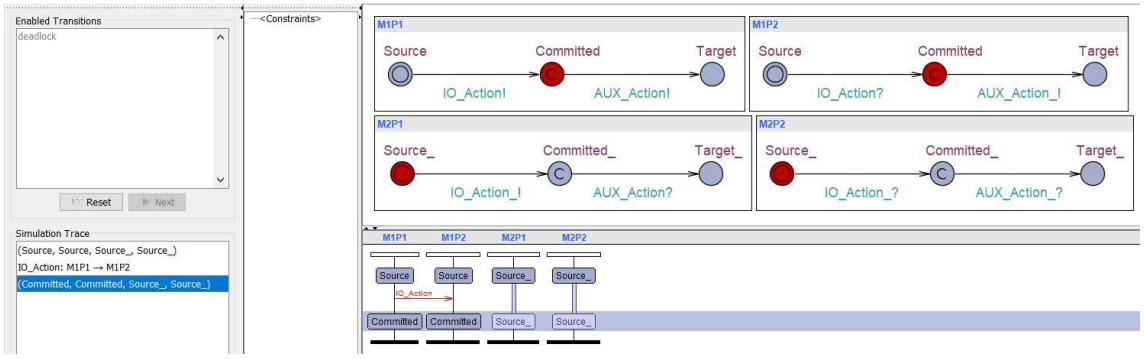


Figure 5. Split edge and committed location addition pattern deadlock

Therefore, there is a need of a pattern that can be applied to synchronize models based on their input/output actions and to preserve the original behavior and non-determinism in the system with separate input environment or set of environment components for each controller.

3.2 UPPAAL Timed Automata Bisimulation Pattern

3.2.1 Assumptions

The algorithm described in 3.3.2 relies on the following assumptions:

- The original models are working and have no errors and deadlocks (this assumption comes from the fact that reactive systems are generally supposed to be deadlock free).
- Models contain environment or set of environment components separately for each controller.
- Models contain input and output channels for communication between controller (SUT) and environment (any template that generates input signals to the controller).
- If two separate models are provided, then they must contain the same input output action labels (channel names) augmented with prefix “i_” for input and “o_” for output. e.g. i_input, o_output.

- If the models are already merged or one model is provided that contains both controllers and their set of environment components that needs to be synchronized for bisimulation verification, then the respective original channels must be distinguished by a postfix "_" with the same names. e.g. i_input, o_output of one set of controller and environments and i_input_, o_output_ for the other set.

3.2.2 Construction of the Pattern and Algorithm

We propose a pattern that complements the comparable models and makes it possible to verify weak bisimulation of system-environment type models by TCTL model checking. The pattern needs to be applied 1) on edges that model observable interactions between system and environment processes; 2) at non-deterministic choices to avoid side effects of non-deterministic constructs in the models, that for both timing and control flow non-determinism. In the following the pattern is described in detail.

The pattern has two parts for the synchronization. The first part of the pattern is applied on one instance and the second part is applied on the other instance which is supposedly bisimilar. To keep the pattern generic and applicable on different UPPAAL models, both parts of the pattern are synchronized using combination of integer variables, a Boolean flag and internal synchronization channels. If the model contains multiple instances of the environment components, then integer variables and internal synchronization channels are parameterized. The flag variable is used on environments synchronization only and purpose of it is to not allow inter-leavings when multiple instances are communicating with the controller.

Let us consider two bisimilar models Mi and Mj both containing two processes MiPi, MiPj, MjPi and MjPj. The process Pj in each model acts as a controller and Pi as its environment. A parallel synchronous composition is constructed between the models by synchronizing MiPi with MjPi and MiPj with MjPj using the pattern. The pattern proceeds as follows:

In the first part of the pattern:

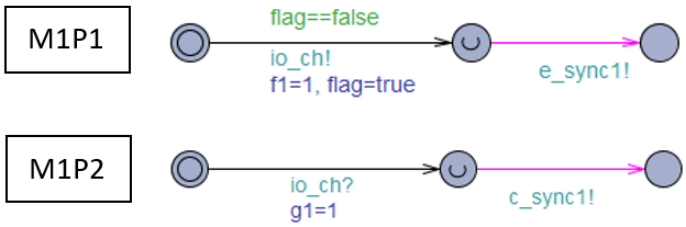


Figure 6. Pattern first part

1. A guard condition is added to original transition with a Boolean variable "flag" with initial value as "false".
2. If the guard value is false, then the original transition with the input/output channel takes place and sets the flag value to true.
3. On the same transition an integer variable f(X) for environment and g(X) for controller with initial value as "0" is set to "1".
4. The original transition is followed by an urgent location. The urgent location allows to wait for the second part of the pattern to reach the state where both instances can synchronize.
5. On the next transition after splitting the original transition with urgent location, an internal synchronization channel is added with prefix "e_syncX" for environment and "c_syncX" for controller to send signal.

In the second part of the pattern:

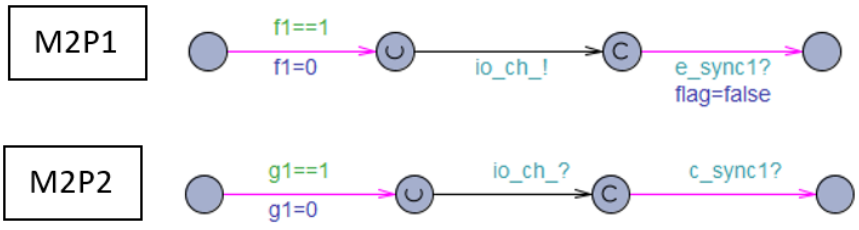


Figure 7. Pattern second part

1. A guard condition is added to check the value of integer variable.
2. For whichever outgoing transition, the integer variable value is true, the value is reset to "0" and transition proceeds to the urgent location.
3. After the urgent location, the original transition with the input/output channel takes place.
4. The original transition is then followed by a committed location.

5. The next transition after splitting the original transition with committed location, will have the internal synchronization channel to receive the signal.
6. Flag variable value is reset to false to end the pattern and allow another pattern application to proceed. Since we are using flag variable in environment templates only, this step is not required in controller synchronization.

3.2.2.1 Algorithm[29]

An algorithm has been developed in the form of tool to automate the synchronization of two potentially bisimilar UPPAAL models. It is developed in .NET Framework 4.7.2 using C# programming language.

The working of the tool is based on the above pattern and assumptions mentioned in 3.2.1. Since UPPAAL does not support opening multiple models at once, we merge the models in one system (See Figure 17 and 19) and make sure both models are initially independent and deadlock free. In order to handle this, if two models are opened in the tool, it supports to modify global and local declarations of the models and merge them in one system to make sure that both models are independently working before the pattern is applied to synchronize the models for bisimulation check. Otherwise, the assumption is that models are already merged and working independently in one system.

The input file(s) for the tool are XMLs generated by UPPAAL. The tool converts these file(s) into C# object by deserializing the XML and type casting using *ModelSchema* class defined. After that, it will proceed to identify the environment and controller templates in the system based on the IO actions sending and receiving signal directions. Once all template types are identified, templates are iterated and transitions are picked one by one which contain IO channels and first part of the pattern is applied. Then, based on the template name and transition source and target ids, same transitions are identified in the other possibly bisimilar template and the second part of the pattern is applied. Once the pattern application is completed on all IO channels, the new declarations are appended to the global declarations and then the new system object is serialized back in XML file format to be readable in UPPAAL tool and saved in *"My Documents"* folder in Windows

[29] https://github.com/naveedahmed986/bisim_algorithm

OS environment. Finally, the tool displays path of the saved file on the UI. The sequence diagram below describes the structure of the tool.

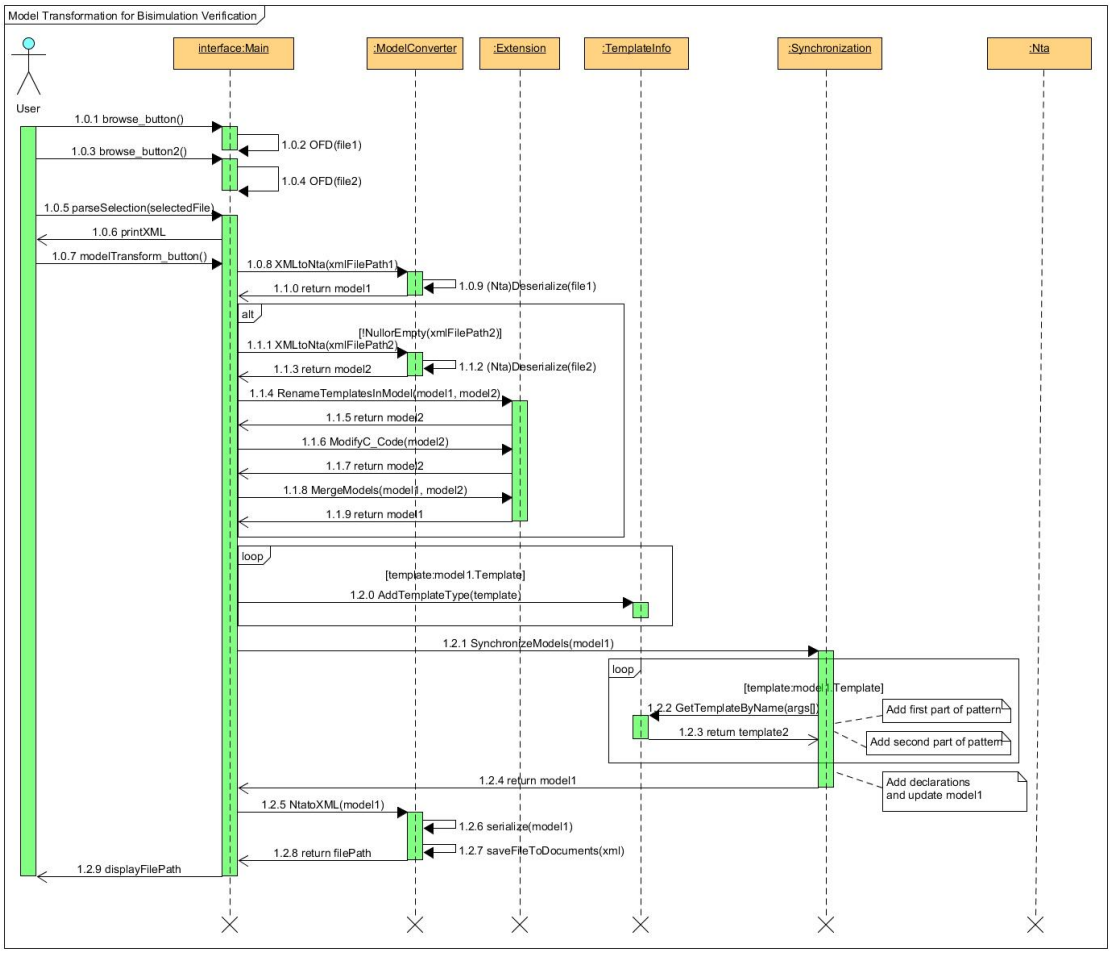


Figure 8. Bisimulation Verification Algorithm Sequence Diagram

3.2.3 Correctness of the Algorithm

Let us consider two bisimilar models Mi and Mj both containing two processes MiPi, MiPj, MjPi and MjPj. The process Pj in each model acts as a controller and Pi as its environment. A parallel synchronous composition is constructed between the models by synchronizing MiPi with MjPi and MiPj with MjPj using the pattern. Below we demonstrate the validity of the pattern with different cases in these models:

3.2.3.1 Case 1: Control flow non-determinism

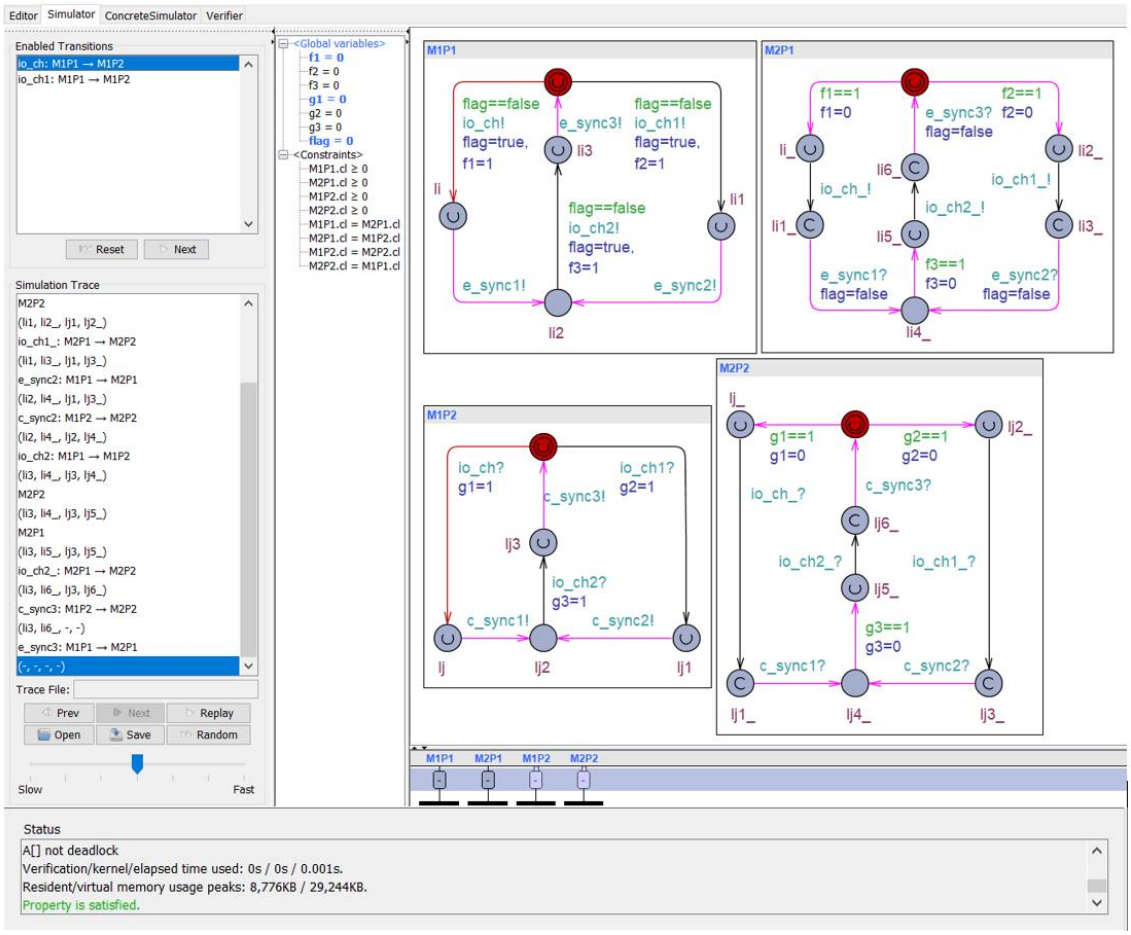


Figure 9. Control flow non-determinism correctness for algorithm

In this case, the pattern is applied on the models containing processes with non-deterministic multiple outgoing transitions from initial location with different IO channels. As shown in Figure 9 the non-deterministic selection of the transition is preserved after applying the pattern in M1P1 and synchronized with M2P1. When the process M1P1 selects an outgoing transition non-deterministically, M2P1 will proceed with the same transition. And for the transition with IO channel that receives the signal from M1P1 in M1P2, the same bisimilar transition will take place in M2P2 when it receives the signal from M2P1.

3.2.3.2 Case 2: Time non-determinism

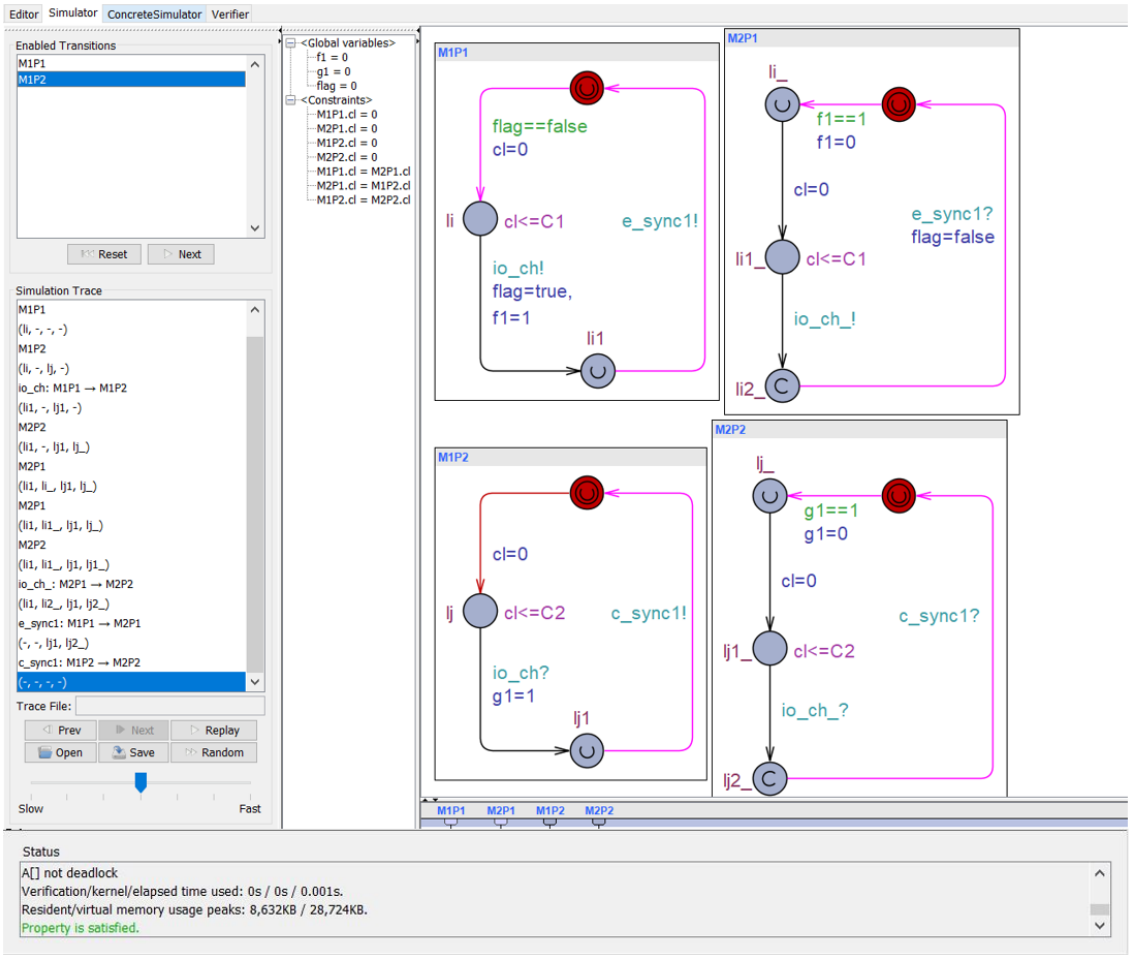


Figure 10. Time non-determinism correctness for algorithm

Figure 10 elaborates the application of the pattern in models containing time non-determinism. The transition containing IO channel after *li* location with invariant in M1P1 will send the signal to M1P2 and M1P2 also contains an invariant before receiving the signal at *li_* location. This non-deterministic timing behavior is preserved in all processes of both models M1 and M2 after synchronizing M1P1 with M2P1 and M1P2 with M2P2.

3.2.3.3 Case 3: Control flow and time non-determinism

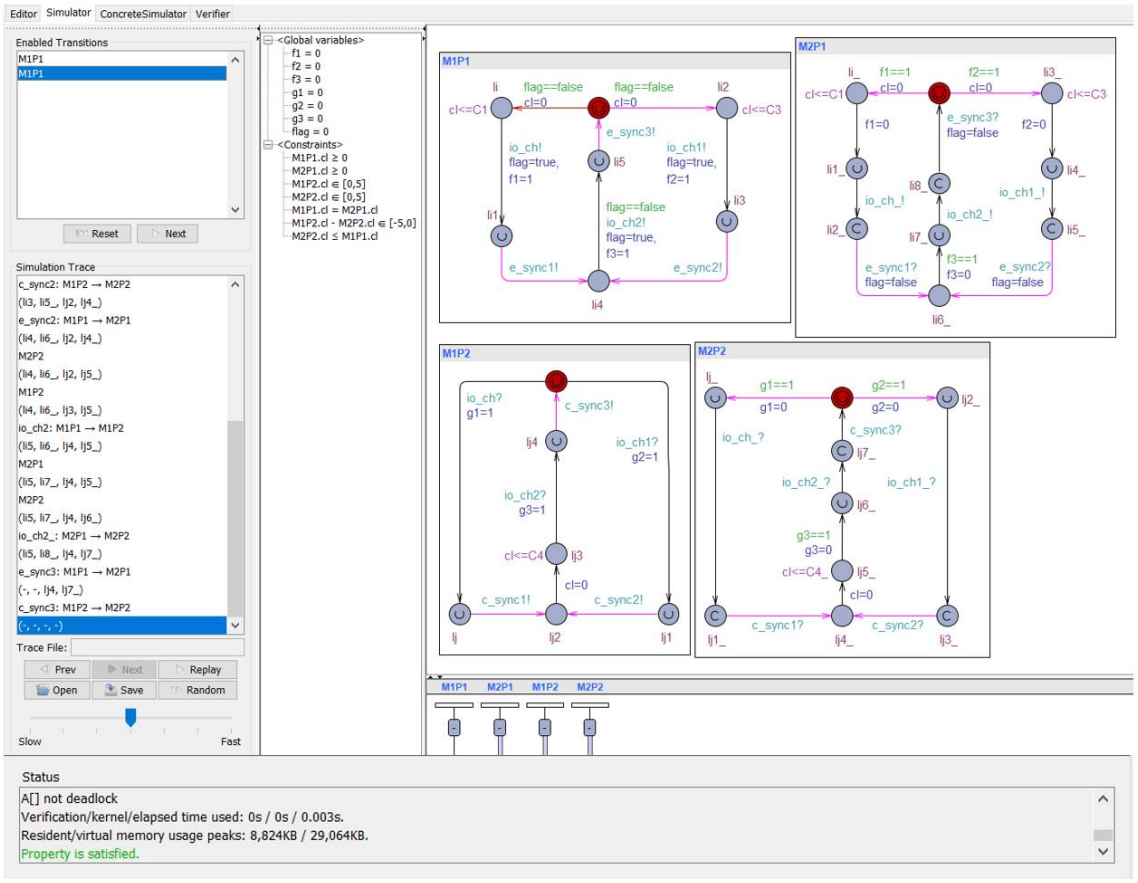


Figure 11. Control flow and time non-determinism correctness for algorithm

The models in Figure 11 contain control flow non-determinism in the form of non-deterministic outgoing transitions and non-deterministic timing using invariants on different locations. After applying the pattern, the processes M2P1 and M2P2 select the non-deterministic transitions same as the selection from M1P1 and M1P2 respectively based on the synchronization channels between the processes.

3.2.4 Correctness by structural induction

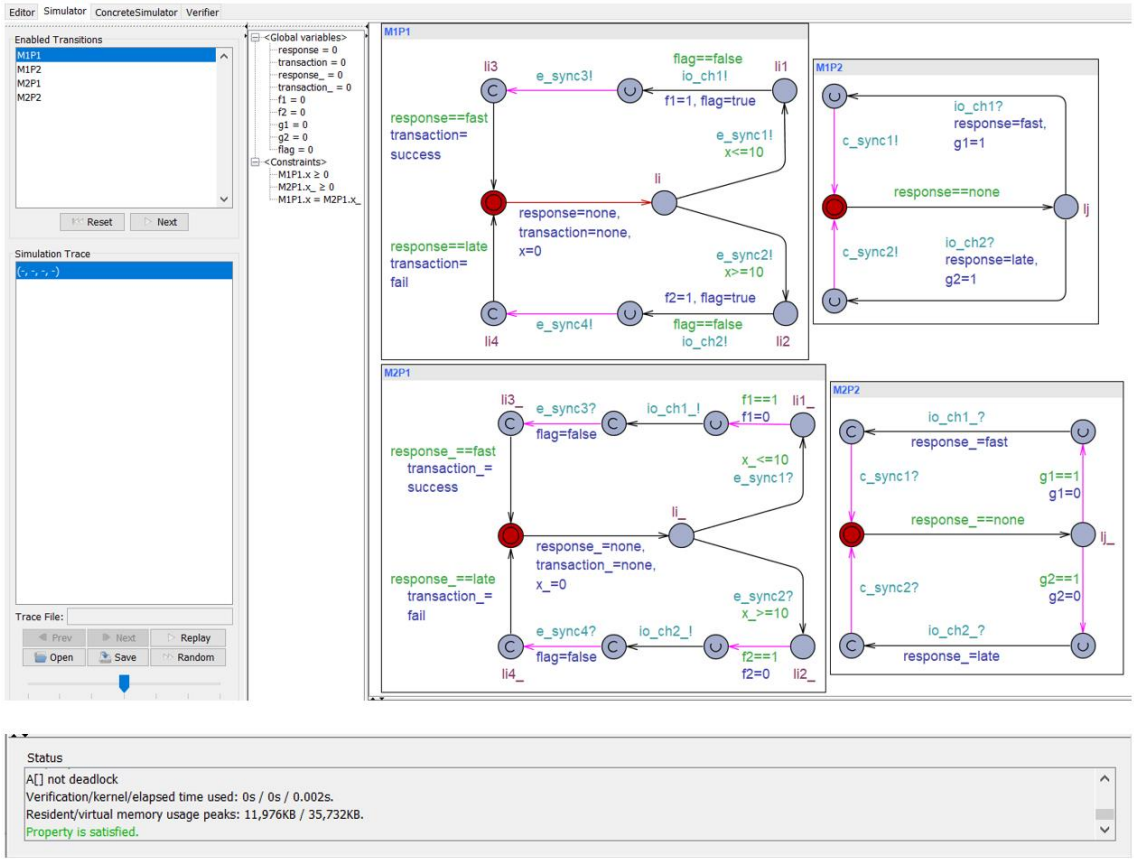


Figure 12. Correctness for algorithm by structural induction

In Figure 12 there are some existing *liX* locations in the process P1 and *lj* in P2 of model M1 and *liX_*, *lj_* in the P1 and P2 processes of M2. Other than these mentioned locations and the initial locations, rest are part of the pattern application added for synchronization on IO actions in the models. The important aspect to observe here is that the pre and post locations and edges of the pattern application do not affect the bisimilarity of the models.

3.2.5 Time and Space complexity of the pattern

The structural complexity of the models enhanced with the pattern described in section 3.2.2 is linear in the number of pattern application points, each application providing 3 extra locations and 3 extra edges in the model. This is because two bisimilar models M1 and M2 are synchronized using the pattern on the non-deterministic outgoing transitions containing IO channels.

3.2.5.1 Control flow non-determinism time and space complexity

The experimental evaluation of bisimulation time and space complexity is performed by measuring the performance metrics of UPPAAL verifier with global deadlock property check, i.e., A[ ] *not deadlock*. The configurations of UPPAAL tool are kept as default and the system used for the test contained Intel Core i5-8250U CPU @ 1.6GHz (8 CPUs), ~1.8GHz and 16GB RAM.

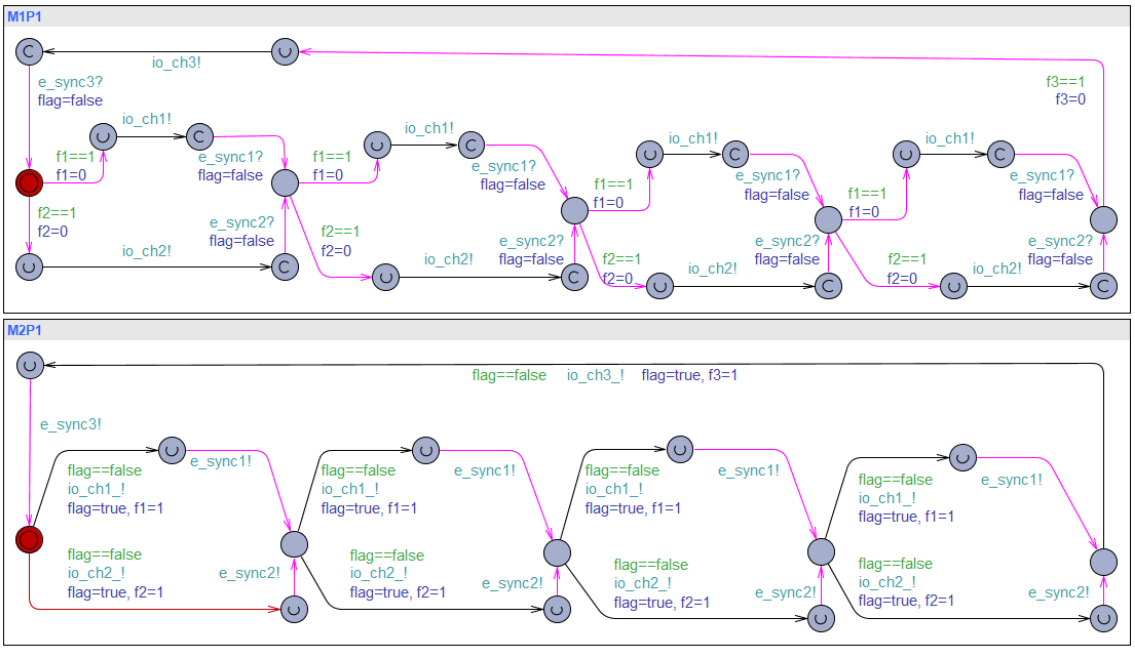


Figure 13. Algorithm repetition in control flow non-determinism

In the above figure, after every repetition of the non-deterministic outgoing transition and application of the pattern, time and memory usage have been recorded with global deadlock check in UPPAAL verifier. The results show that the increase in time and memory usage is linear with the number of pattern applications in the system containing non-determinism in control flow.

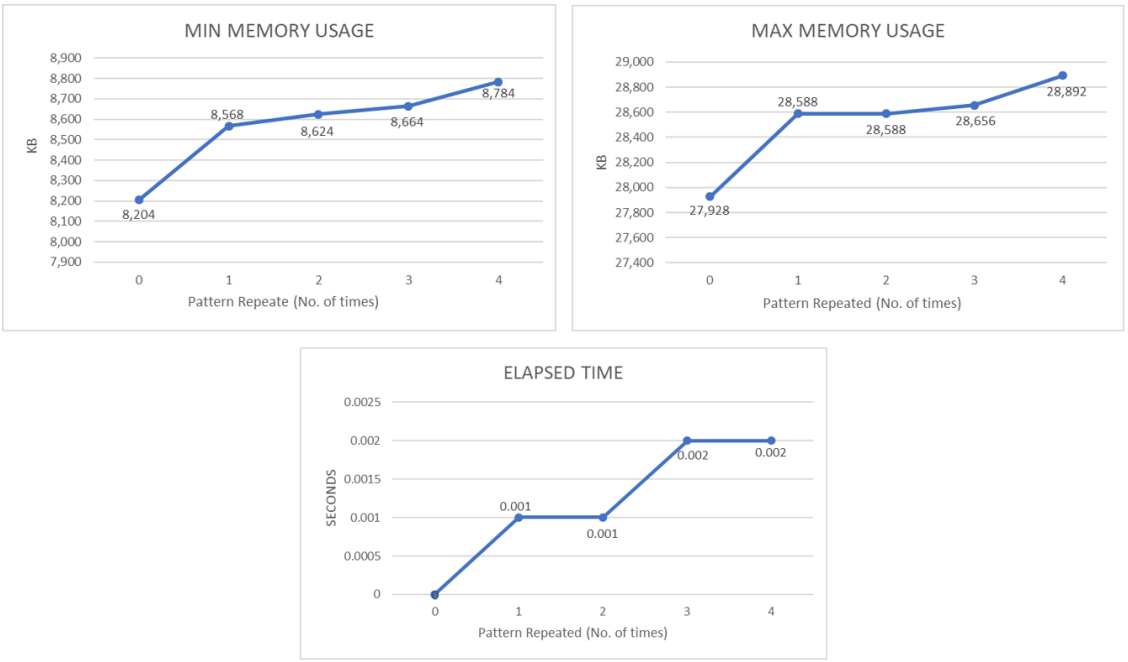


Figure 14. Algorithm repetition control flow non-determinism results

3.2.5.2 Time non-determinism time and space complexity

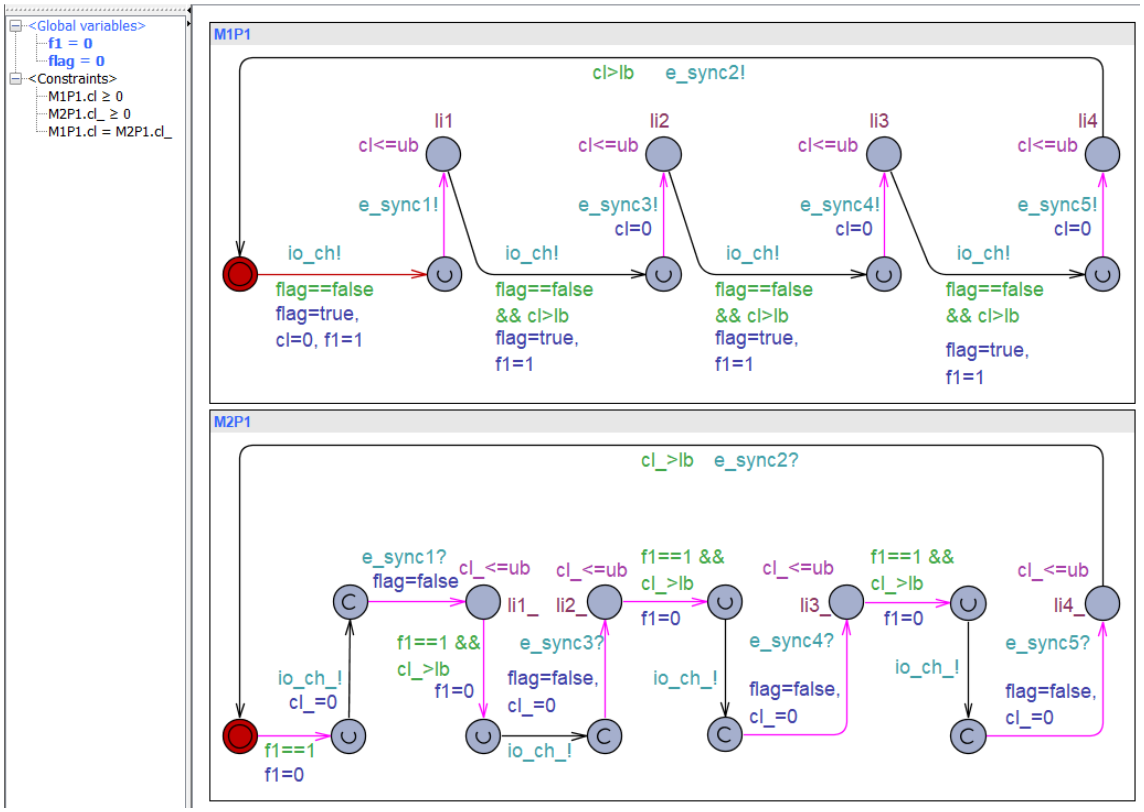


Figure 15. Algorithm repetition in time non-determinism

Following the same approach and configuration mentioned in section 3.2.5.1, the number of *li* and *li_* locations are increased in the models with the pattern application. The time non-determinism is defined by bounds *ub* and *lb* of the invariants on these locations and the clock guards respectively (See Figure 15). On each step of adding *li* and *li_* locations, time and space complexity is measured with the global deadlock check property. The measurement results indicate that memory usage is linear in the number of pattern applications for time non-determinism but the time complexity is constant for application numbers 1 to 4.

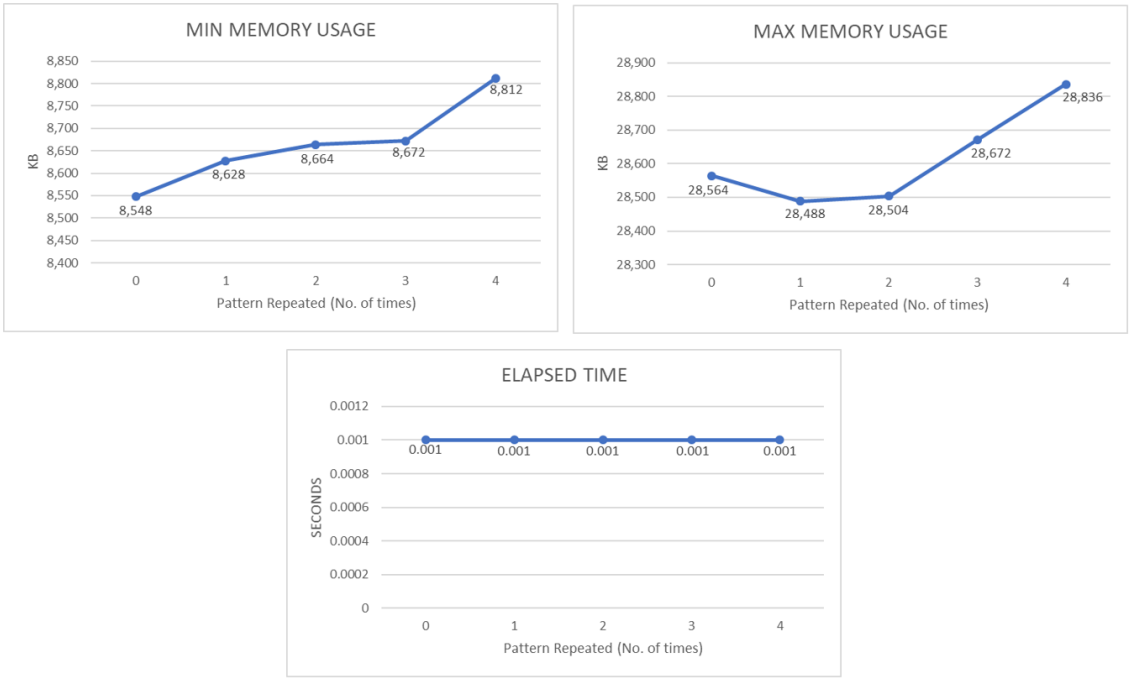


Figure 16. Algorithm repetition time non-determinism results

4 Case Studies

The proposed pattern for bisimulation verification of UPPAAL models is applied on many[28] case studies to show its generic approach and applicability on realistic models. Case studies are from different sources having different structures. The details of two case studies are mentioned below.

4.1 Train-gate [5] [27]

SUT (Controller): Gate

Environment: Train

Following is the storyline of the system:

- Multiple trains can approach the gate.
- Whenever multiple trains are approaching the gate, they are added to the queue and whenever train leaves, it is removed from the queue.
- If the gate is at initial state and the queue is not empty, it means that at least one train is at stop state. The gate then allows the train to cross or select another approaching train non-deterministically.
- The selection of the train instances in both gates to either approach and stop or leave is non-deterministic.

In this case study, there are two bisimilar gate controllers named *Gate and Gate_*. Model-wise there is a single template of train for each controller but these templates are parameterized and each gate will be interacting with multiple instances of the trains. i.e. *Train[N]* →*Gate* and *Train_[N]* →*Gate_*. In Figure 17, it is shown that both models are merged in one system and both are operating independently without any deadlocks.

[28] https://github.com/naveedahmed986/bisim-uppaal-models

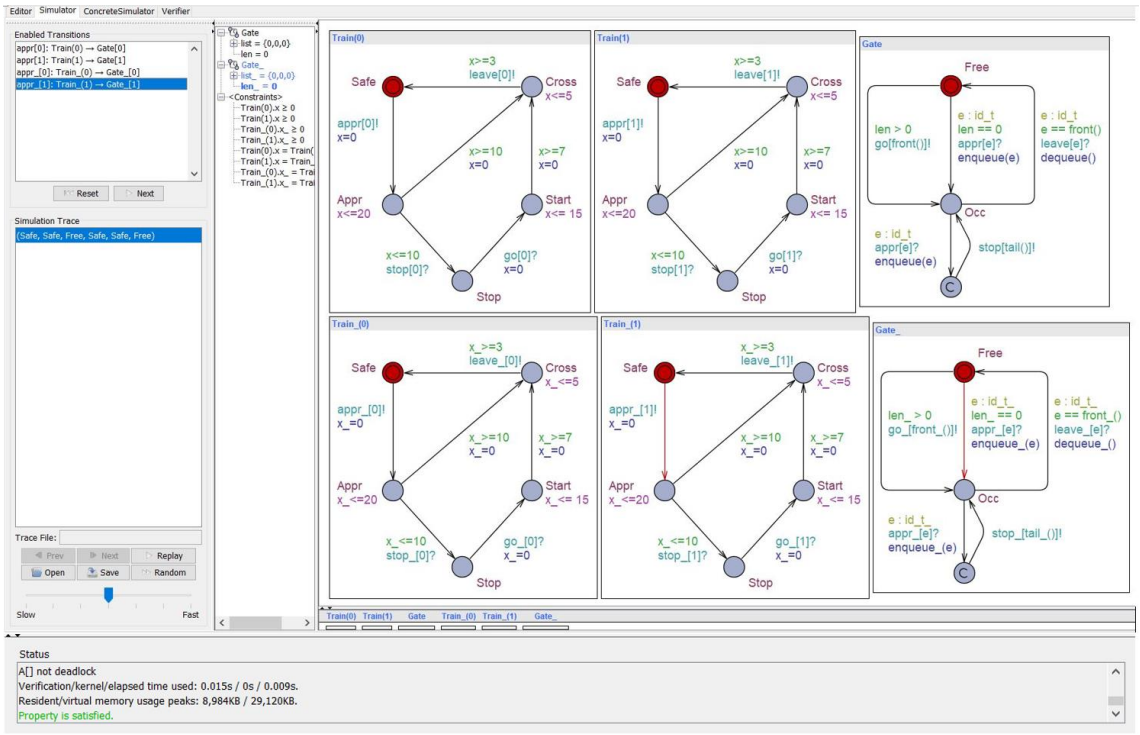


Figure 17. Train-gate independent models merged

Next, the models are synchronized for bisimulation check using the pattern defined in 3.2. Since the models contain multiple instances of the environment component then the synchronization channels are also parameterized in the *Train* and *Train_* templates as mentioned in 3.2.2. The parameterization helped to synchronize the correct pair of the instances from two environment components so that when one instance makes a move, the same instance from the other environment component must also make the exact same move. E.g. T*rain[0] ~Train_[0]*.

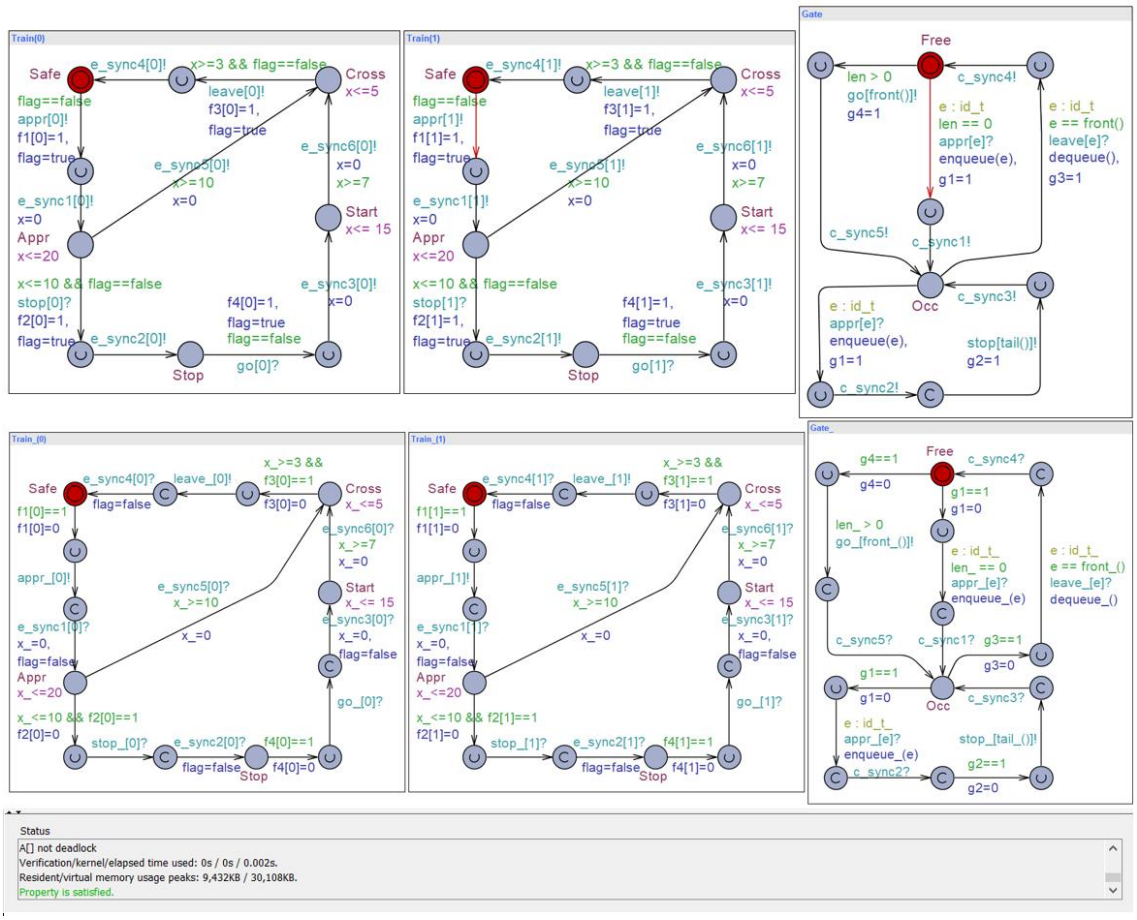


Figure 18. Synchronized Train-gate models using proposed pattern

On the controller side, when one controller selects an environment instance then the other controller must also select the same instance from its own environment instances. As shown in Figure 18, after applying the pattern fully on all components of the models where there are IO actions and synchronizing invariants with guards, the system is still deadlock free and the behavior of the system is as expected.

4.2 Coffee-machine[26]

SUT (Controller): Machine

Environment: Person

Other: Observer

The storyline of two bisimilar models in this case study is as follows:

- Person inserts a coin in the machine and then wait and get ready to receive the coffee.
- Machine receives the coin and immediately or after some delay release the coffee non-deterministically.
- Person receives the coffee and after some delay, go and publish
- Observer receives the publish and if certain time had passed then Observer will complain otherwise stay at publish.

Since in this case study, the environment instances are single for each controller then application of the pattern is rather simpler than having multiple instances of each environment. First, two models are merged into one system and the global deadlock query has been verified (See Figure 19) that the models are operating independently without any deadlocks.

[26] Model Checking and time CTL (2020), Received from: https://www.comp.nus.edu.sg/~cs5270/2006-semesterII/chapt6.pdf

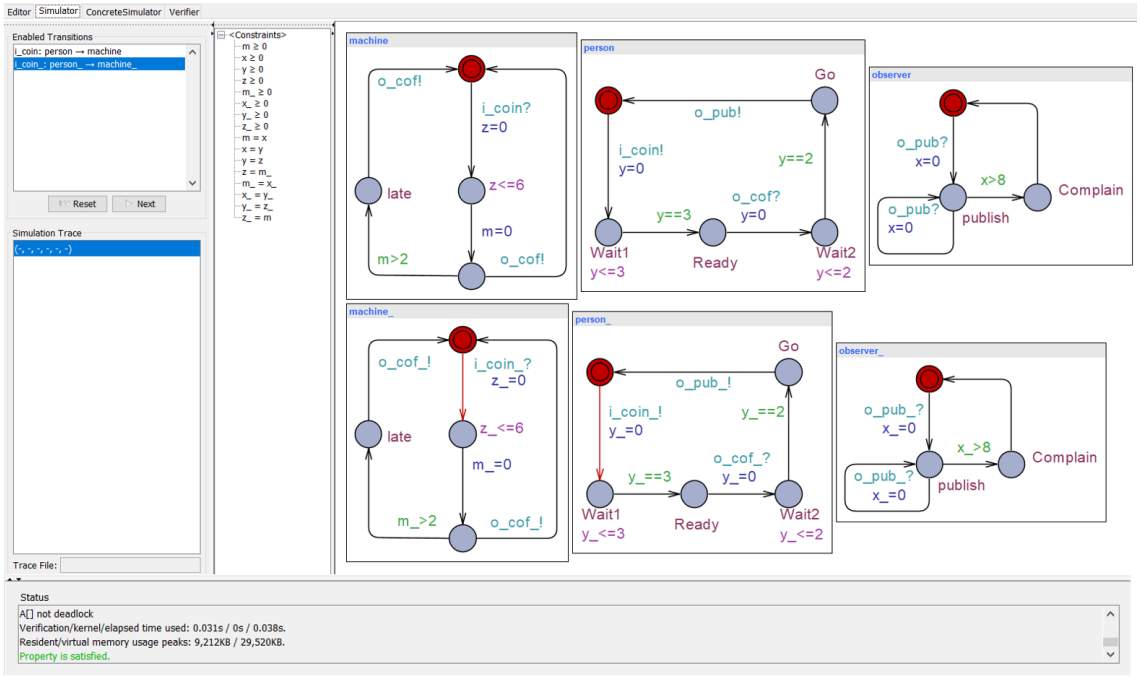


Figure 19. Coffee-machine independent models merged

After it has been verified that the models behavior is not affected by merging them into one system, the pattern is applied to synchronize *person~person_* , *machine~machine_* and *observer~observer_* as showin in Figure 20. The models are synchronized and bisimulation check is verified. Non-determinstic control flow and time non-determinism has been preserved in the models and system runs deadlock free.

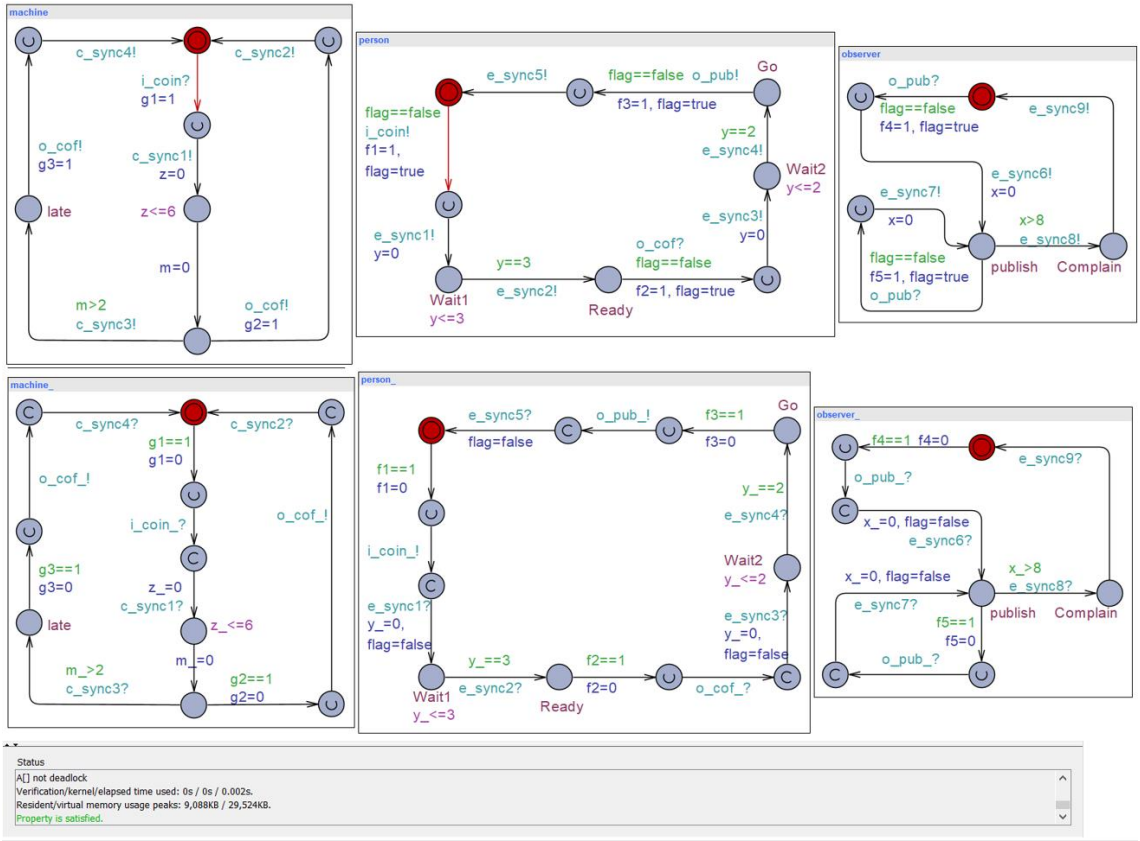


Figure 20. Synchronized Coffee-machine models using proposed pattern

5 Analysis

Initially we started with validating the theory mentioned in [1] and [24] regarding bisimulation verification. Soon we realized that we need to modify the approach and construct a pattern based on it because in our case we were considering separate environments and controllers in the potentially bisimilar models. For that, the pattern was extended by adding synchronization channel before an IO action whenever an IO action is sending a signal and after an IO action is receiving a signal. And also a synchronization channel was added on the transitions with clock guards, as shown in Figure 21.

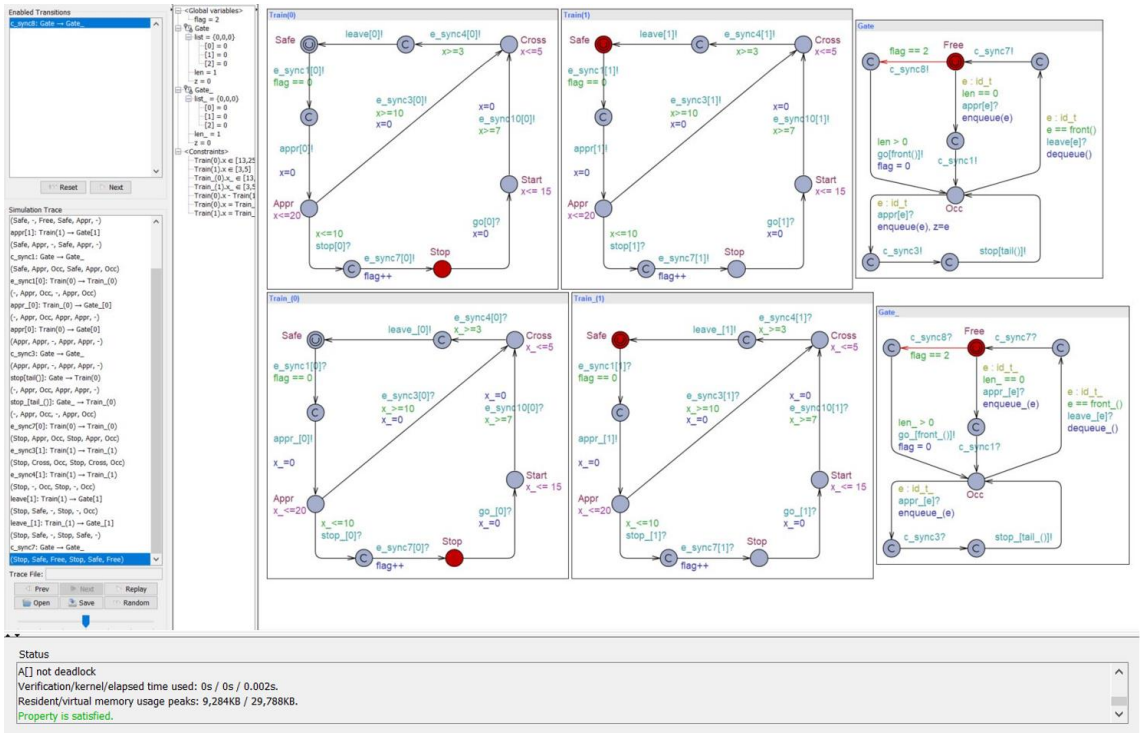


Figure 21. Synchronized Train-gate models with split edge and committed location addition pattern

Though the pattern works on some of the models and original behavior of the system is preserved, the pattern is not generic enough to be applicable on many different models. The reasons are i. The use of *flag* variable is customized to avoid a deadlock specific to the implementation of this model, ii. It is not necessary to apply the pattern on all IO actions and models will still be synchronized. Observe the *stop* channel in the controllers *Gate* and *Gate_*, the synchronization is not added after sending the signal because the choice is deterministic and the location before this transition is committed which will enforce both controllers to take the same decision, iii. Another reason that this pattern

works but not generically is that in some cases after applying this pattern on all IO actions will lead to a deadlock when one pair of the synchronized templates are forced to send the IO signal due to committed locations and the receiving end is not yet ready as shown in Figure 22.

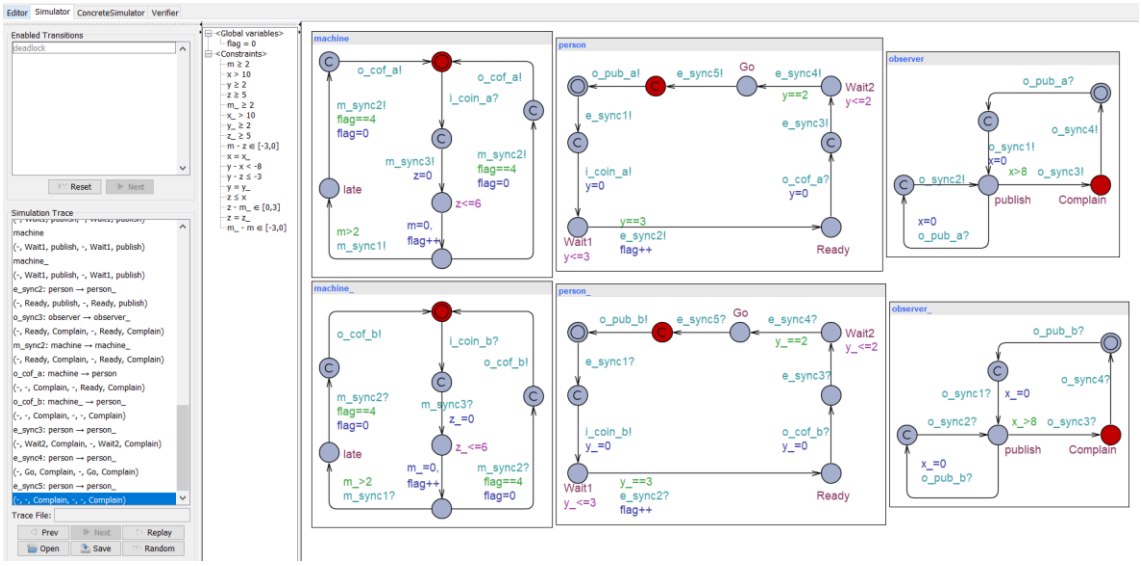


Figure 22. Synchronized coffee-machine models split edge and committed location addition pattern deadlock

The next pattern we constructed was more generic and applicable on many different complexity level models. The pattern preserved time non-determinism in models but the main challenges were i. Handling control-flow non-determinism with multiple instances of environment templates e.g. multiple trains in train-gate example, ii. Applying the pattern when two bisimilar environments are sending the same signal to their respective controllers. And at the same time, when both sending and receiving ends are synchronizing then the pattern would reset the flags before the guards hold in one side of the synchronization which was causing a deadlock, iii. The pattern allowed inter-leavings during the execution of the pattern application at urgent locations. Figure 23 shows the structure of the pattern and Figure 24 shows the application of this pattern on train-gate example.

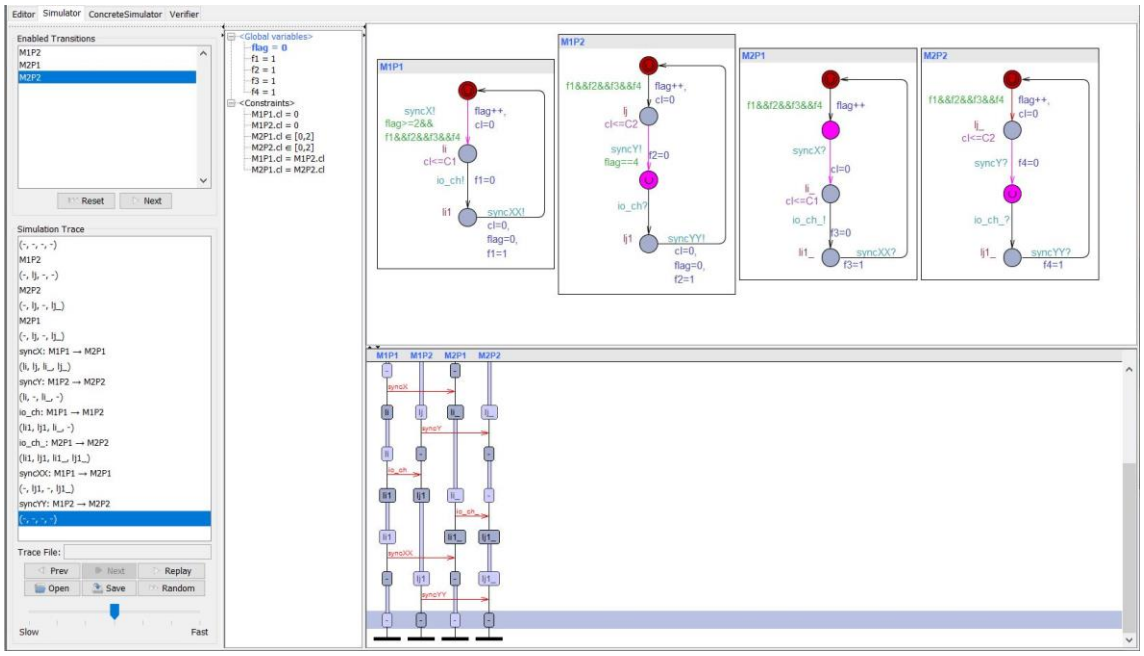


Figure 23. Bisimulation second pattern structure

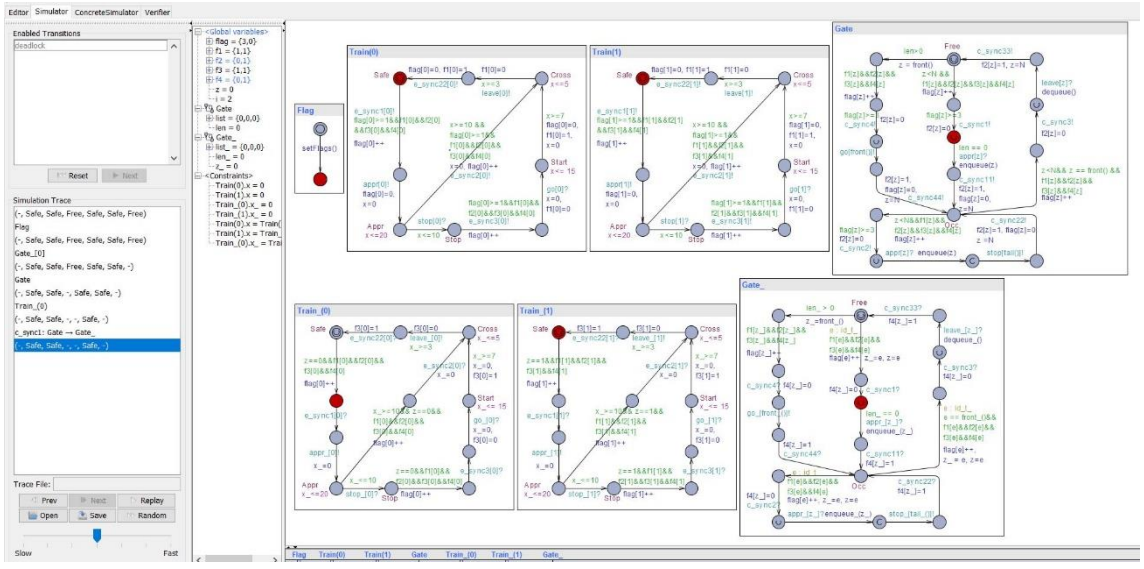


Figure 24. Non-working bisimulation second pattern application on Train-gate example

The above research led us to the pattern proposed in section 3.2 which preserves the time and control-flow non-determinism in the system and also synchronizes the moves of two bisimilar models based on their IO action and clock guards.

6 Conclusion

6.1 Summary

Bisimulation verification plays a vital role in the qualitative and quantitative comparison of models semantics of which is based on labelled state transition systems. It helps to compare the performance, trace and behavioral equivalence of the models and to reduce state-space based on the symmetries in the models.

UPPAAL is meant to develop timed models based on formal specifications and check properties on the models using UPPAAL verifier engine. Currently, any claims made on the bisimilarity of two UPPAAL models cannot be verified directly using the UPPAAL tool. Models can be opened one by one in the UPPAAL tool to verify the properties on them separately. But that does not guarantee the bisimilarity because even if the generated traces are similar there may be infinite sets of traces on which conclusive comparison is not possible.

The main goal of this thesis was, firstly, to clarify the meaning of bisimulation and bisimulation verification in formal methods, secondly, to evaluate the existing approach [1] and its applicability on more realistic models, thirdly, to implement a generic algorithm which will synchronize two comparable UPPAAL models for bisimulation checking using UPPAAL model checker and lastly, to evaluate the extended approach by applying it on UPPAAL standard case studies.

We started with the existing approach [1] and gradually moved to constructing, improving and evaluating this extension which explicitly takes into account the need to synchronize both control flow non-determinism and timing non-determinism of comparable UPPAAL models. Based on the proposed approach, an algorithm has been developed to automate the synchronization of two UPPAAL models. The correctness analysis with performance evaluation of the approach is provided and applied on few case studies of different complexity levels and two of those case studies are presented in the Chapter 4 of thesis.

6.2 Future work

During the research of this thesis, many experiments have been conducted on different UPPAAL models to validate and improve the existing approach [1] and also to come up with a better approach that is more realistic and then to improve it to be generically applicable on different models as mentioned in chapter 5. An enhanced pattern has been proposed with correctness proofs in the section 3.2 and its application on case studies in chapter 4. Though the pattern is proven to be generically applicable for bisimulation check on different models with UPPAAL semantics, due to the time constraints for this research, the application of the pattern is not verified for a very large scale and complex timed automata models.

Airplane [25] is an example of such models. In this model, to form a stronger bisimulation, not only the pattern is applied on all IO actions as mentioned in 3.2 but also other bisimilar transitions are synchronized using synchronization channels. Since there are multiple instances of environment for each controller, these instances manipulate shared variables from the original model. The synchronized transitions without IO actions do not restrict two instances from changing the value of the shared variable before the pattern execution in controllers is completed. This scenario causes a deadlock because the same shared variables are used as guards in controller and the conditions do not hold anymore.

This example can be considered as a baseline for further research on bisimulation verification of UPPAAL models using the pattern introduced in this thesis and also for improvements in the pattern to form a stronger bisimulation pattern.

References

[1] Vain, J.; Halling, E.; Kanter, G.; Anier, A.; Pal, D. (2016). Model-based testing of real-time distributed systems. *Databases and Information Systems : 12th International Baltic Conference, DB&IS 2016, Riga, Latvia, July 4-6, 2016, Proceedings*. Ed. Arnicans, G.; Arnicane, V.; Borzovs, J.; Niedrite, L. Cham: Springer, 272–286. (Communications in Computer and Information Science; 615).10.1007/978-3-319-40180-5_19.

[2] Davide Sangiorgi. 2009. On the origins of bisimulation and coinduction. ACM Trans. Program. Lang. Syst. 31, 4, Article 15 (May 2009), 41 pages. DOI:https://doi.org/10.1145/1516507.1516510

[3] Wimmer R., Herbstritt M., Hermanns H., Strampp K., Becker B. (2006) Sigref – A Symbolic Bisimulation Tool Box. In: Graf S., Zhang W. (eds) Automated Technology for Verification and Analysis. ATVA 2006. Lecture Notes in Computer Science, vol 4218. Springer, Berlin, Heidelberg

[4] Norris IP, C. & Dill, D.L. (1996). "Better verification through symmetry", Form Method Syst Des (1996) 9: 41. https://doi.org/10.1007/BF00625968

[5] Behrmann G., David A., Larsen K.G. (2004) A Tutorial on Uppaal. In: Bernardo M., Corradini F. (eds) Formal Methods for the Design of Real-Time Systems. SFM-RT 2004. Lecture Notes in Computer Science, vol 3185. Springer, Berlin, Heidelberg

[6] J. Bengtsson, W. Yi, 2004. "Timed automata: Semantics, algorithms and tools," Lecture Notes on Concurrency and Petri Nets, Lecture Notes in Computer Science vol. 3098.

[7] Norris IP, C. & Dill, D.L. (1996). "Better verification through symmetry", Form Method Syst Des (1996) 9: 41. https://doi.org/10.1007/BF00625968

[8] K. Čerāns, "Decidability of bisimulation equivalences for parallel timer processes," in Computer Aided Verification: Fourth International Workshop, CAV '92 Montreal, Canada, June 29 -- July 1, 1992 Proceedings, G. von

Bochmann and D. K. Probst, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 302-315.

[9] Fisler K., Vardi M.Y. (1999) Bisimulation and Model Checking. In: Pierre L., Kropf T. (eds) Correct Hardware Design and Verification Methods. CHARME 1999. Lecture Notes in Computer Science, vol 1703. Springer, Berlin, Heidelberg

[10] Wimmer R., Herbstritt M., Hermanns H., Strampp K., Becker B. (2006) Sigref – A Symbolic Bisimulation Tool Box. In: Graf S., Zhang W. (eds) Automated Technology for Verification and Analysis. ATVA 2006. Lecture Notes in Computer Science, vol 4218. Springer, Berlin, Heidelberg

[11] Yovine S. (1998) Model checking timed automata. In: Rozenberg G., Vaandrager F.W. (eds) Lectures on Embedded Systems. EEF School 1996. Lecture Notes in Computer Science, vol 1494. Springer, Berlin, Heidelberg

[12] K.G. Larsen, M. Mikuˇcionis, and B. Nielsen. Online Testing of Real-time Systems Using UPPAAL. In FATES'04, LNCS, pages 79–94, Linz, Austria, September 2004.

[13] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In the 5th ACM international conference on Embedded software, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.

[14] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. UPPAAL-TIGA: Time for playing games! In Proceedings of the 19th International Conference on Computer Aided Verification, number 4590 in LNCS, pages 121–125. Springer, 2007.

[15] John H˚akansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT. In ATVA, pages 252–257, 2008.

[16] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. ECDAR: An environment for compositional design and analysis of real time systems. In Proceedings of ATVA 2010, page to appear, 2010.

[17] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: a complete specification theory for real-time systems. In HSCC, pages 91–100. ACM, 2010

[18] Collins, M. 1998. Formal Methods. Carnegie Mellon University, Pittsbugh, PA. Spring, 18-849b Dependable Embedded Systems

[19] J. Lockhart, C. Purdy and P. Wilsey, "Formal methods for safety critical system specification," 2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS), College Station, TX, 2014, pp. 201-204, doi: 10.1109/MWSCAS.2014.6908387.

[20] El-Gendy, Hazem & El,Kadhi, Nabil. (2005). Formal methods: Importance, experience, and comparative analysis. Journal of Computational Methods in Sciences and Engineering. 5. 235-247. 10.3233/JCM-2005-5S118.

[21] E. -. Olderog, "Formal methods in real-time systems," Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168), Berlin, Germany, 1998, pp. 254-263, doi: 10.1109/EMWRTS.1998.685130.

[22] Voros N.S., Mueller W., Snook C. (2004) An Introduction to Formal Methods. In: Mermet J. (eds) UML-B Specification for Proven Embedded Systems Design. Springer, Boston, MA

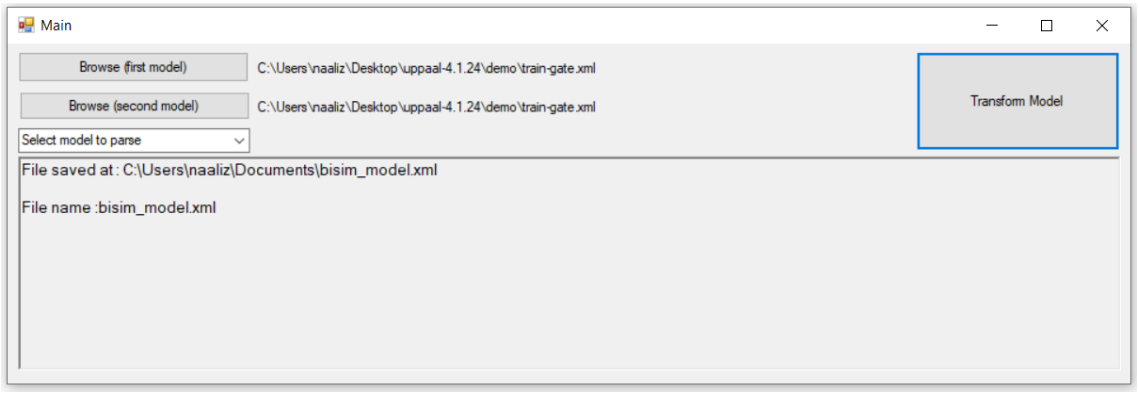
[23] Bourahla, Mustapha. (2009). Modeling and Verification of Real-Time Embedded Systems. 6-6. 10.14236/ewic/ISIICT2009.6.

[24] Sarna, K., & Vain, J. (2018). “Aspect-Oriented Model-Based Testing”, Doctoral Thesis, Tallinn University Of Technology. TTU Press ISBN 978-9949-83-344-3

[25] Shokri-Manninen, Fatima & Tsiopoulos, Leonidas & Vain, Juri & Waldén, Marina. (2020). Integration of iUML-B and UPPAAL Timed Automata for Development of Real-Time Systems with Concurrent Processes. 10.1007/978-3-030-48077-6_13.

[27] Yi W., Pettersson P., Daniels M. (1995) Automatic Verification of Real-Time Communicating Systems by Constraint-Solving. In: Hogrefe D., Leue S. (eds) Formal Description Techniques VII. IFIP Advances in Information and Communication Technology. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-34878-0_18

Appendix 1 – Model Transformation Tool for Bisimulation Checking



The tool works as described in 3.2.2.1.

1. Browse (first model) button: Mandatory to select model as xml file. Assumptions about the model are mentioned in 3.2.1.

2. (Optional) Browse (second model) button: Optional if two models are already merged and selected as first model.

3. Transform model button: Initiates the process of bisimulation synchronization between the models and saves the resulting model in “My Documents” directory.

4. (Optional) Select model to parse dropdown: shows the xml content of selected model in the textbox.

5. Textbox: Display the xml content if model is parsed or the information about the synchronized model saved location once bisimulation synchronization process is done.