TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Tanel Prikk 134526IAPB

IMPLEMENTATION OF AN INTERPRETER FOR THE TEST PURPOSE SPECIFICATION LANGUAGE TDL^{TP}

Bachelor's thesis

Supervisors: Jüri Vain
PhD
Evelin Halling
PhD

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Tanel Prikk 134526

TESTIMISEESMÄRKIDE SPETSIFITSEERIMISKEELE TDL^{TP} INTERPRETAATORI IMPLEMENTEERIMINE

bakalaureusetöö

Juhendajad:	Jüri Vain
	PhD
	Evelin Halling
	PhD

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: Tanel Prikk

21.05.2019

Abstract

In model-based testing (MBT), explicit behavioral models of a system under test (SUT) are used to generate abstract test cases (ATCs) – sequences of inputs and expected outputs which can be concretized into executable test scripts. The phase of MBT in which ATCs are produced is known as test generation. Given an appropriate modeling formalism and a language for test purpose specification, this phase can be automated.

The chosen modeling formalism depends on the nature of the SUT. For systems which exhibit time-sensitive behavior, i.e. real-time systems, a suitably flexible formalism supported by a mature set of tools is UPPAAL Timed Automata (UTA), wherein behavioral models are specified as state-transition graphs annotated with timing constraints. However, while the UPPAAL toolkit provides means for test purpose specification via its subset of Timed Computation Tree Logic (TCTL), certain syntactic limitations prevent the complete automation of ATC generation for UPPAAL models – manual modification of the SUT model is required for the implementation of certain complex test purposes.

A solution to this issue was provided by the authors of [1] in the form of a supplementary language layer known as the Test Purpose Specification Language (TDL^{TP}). Using TDL^{TP}, a test purpose can be declared as a logical expression which encodes an augmentation procedure for a UPPAAL SUT model. Once the encoded modifications are carried out, the resultant test model can be used in conjunction with UPPAAL TCTL.

With its expressive, flexible syntax, TDL^{TP} overcomes the limitations of the UPPAAL property specification language and can be treated as a major stepping-stone towards extending the practical usability of UPPAAL in MBT processes.

The objective of this thesis was to implement an interpreter which accepts a UPPAAL SUT model and a TDL^{TP} expression as input and produces an ATC in the form of a test model as output.

The author produced the result by gradually implementing and interfacing a collection of reusable components. The structure and logic of these components is discussed and explained in this thesis. It is expected that the modular structure of the interpreter will facilitate future extensions and improvements.

This thesis is written in English and is 78 pages long, including 6 chapters, 41 figures and 5 tables.

Annotatsioon

Testimiseesmärkide spetsifitseerimiskeele TDL^{TP} interpretaatori implementeerimine

Mudelipõhises testimises (*model-based testing* – MBT) kasutatakse testitava süsteemi (*system under test* – SUT) käitumismudeleid, et genereerida testsisendite ja vastavate eeldatavate väljundite jadasid ehk abstraktseid testimisjuhte (*abstract test case* – ATC). Viimaseid on võimalik konkretiseerida reaalselt täidetavateks teststsenaariumiteks. Eeldades sobivat modelleerimisformalismi ja keelt, millise abil saab spetsifitseerida testimiseesmärke, on ATC'de genereerimisfaasi võimalik automatiseerida.

Formalismi valik sõltub süsteemi üldistest omadustest. Reaalaja süsteemide jaoks, milliste käitumine on ajatundlik, on sobivalt paindlik ning laia tööriistavalikuga variant *UPPAAL Timed Automata* (UTA). UTA's spetsifitseeritakse käitumismudeleid ajapiirangutega annoteeritud seisundigraafidena. Kuigi UPPAAL'i tööriistakomplektis on olemas vahendid testimiseesmärkide spetsifitseerimiseks *Timed Computation Tree Logic* (TCTL) konkreetse alamhulga näol, seavad teatud süntaktilised piirangud ATC'de generatsiooni automatiseerimisele UPPAAL'i raames kitsendusi – mudeleid tuleb käsitsi kohendada, et teatud keerulisi testimiseesmärke realiseerida.

Ülaltoodud probleemile on artikli [1] autorid pakkunud välja lahenduse täiendava keelekihi, testimiseesmärkide spetsifitseerimiskeele TDL^{TP} näol. Kasutades mainitud keelt, saab eesmärgi deklareerida loogilise avaldisena, mis on sisuliselt SUT mudeli modifitseerimisprotseduuri kompaktne esitus. Kui vastavad modifikatsioonid teostada, saab resultaadina SUT mudelist tuletatud testmudelit edaspidi TCTL'iga kombinatsioonis kasutada.

TDL^{TP} võimaldab oma paindliku ja väljendusrikka süntaksiga ületada UPPAAL'i mudeliomaduste spetsifitseerimiskeele piiranguid. Seda võib pidada suureks sammuks UPPAAL'i kasutatavuse laiendamise poole MBT kontekstis.

Käesoleva töö eesmärgiks oli implementeerida interpretaator, mille sisendiks on UPPAAL'i formalismis esitatud SUT mudel ja TDL^{TP} avaldis. Väljundina loob interpretaator testmudeli, mis implementeerib avaldises sisalduvat testimiseesmärki.

Töö tulemuseni jõudis autor komponendipõhise lähenemisega. Alustati sisendeid teisendavatest moodulitest ning vajalikest objektmudelitest ning liidestamise abil liiguti järk-järgult kõrgemal abstraktsusastmel asuvate komponentide poole. Selles töös tutvustatakse implementeeritud artefakte ja selgitatakse nende interaktsioone testmudeli loomisprotsessi jooksul. Autor eeldab, et interpretaatori modulaarne ülesehitus soodustab tulevaste rakenduslaienduste lisamist.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 78 leheküljel, 6 peatükki, 41 joonist, 5 tabelit.

List of abbreviations and terms

ANTLR	Another Tool for Language Recognition. A framework for generating parsers.
AST	Abstract syntax tree. A tree-based form of intermediate representation which emphasizes the conceptual rather than the syntactical structure encoded in an input string.
ATC	Abstract test case. Generalization of a collection of concrete input-output sequences which can be used to test a system.
base trapset	A trapset where the mapping between the set and the transitions in the SUT model has been made explicit – possibly through the use of an auxiliary data structure.
BBT	Black-box testing. A verification method where only the inputs and outputs of the software under test are under consideration – internals are not inspected or accessed.
BNF	Backus-Naur form. A notation for describing computer languages.
CBD	Component-Based Design. A development approach which emphasizes loose coupling and the use of reusable components.
CLI	Command-line interface. A type of interface where users are expected to provide input via text commands.
DSL	Domain-specific language. A language which has a limited domain of applicability but allows for more succinct or expressive representation within the context of said domain.
GUI	Graphical user interface. A type of interface where the user interacts with visual elements in order to provide input and receive output from a software system.
JAXB	Java Architecture for XML Binding. Java framework for processing XML.
MBD	Model-based design. A design methodology where requirements for systems are represented as models.
MBT	Model-based testing. The application of MBD for software verification.
SUT	System under test.

TA	Timed Automata. A modeling formalism where system behavior is abstractly represented as a network of state transition graphs.
TCTL	Timed Computation Tree Logic. A type of logic where it is possible to reason with time-related propositions.
TDL^{TP}	Test Purpose Specification Language. A novel method for specifying test purposes designed for inclusion in a UPPAAL MBT workflow.
UI	User interface.
UPPAAL	A tool suite based on the TA formalism and intended for system verification. Developed by Uppsala University and Aalborg University.
UPPAAL system definition language	The C-like declaration language used in UPPAAL to declare variables, define transition and location labels, and specify model processes.
UQL	UPPAAL query language. TCTL-based property specification language available in the UPPAAL toolkit.
UTA	UPPAAL Timed Automata. The extension of TA made available as a modeling formalism in UPPAAL.
automaton template	In UPPAAL, an automaton prototype which defines the structure and behavior of model processes which can be instantiated from it.
code generation	Conversion from an object structure to a syntactical structure in a given language.
conditional trap	A trap which labels a transition whose inclusion in a trapset depends on a logical condition.
directed multigraph	A graph where multiple directed edges are permitted between a pair of vertices.
elementary trap	A trap which maps to a transition whose inclusion in a trapset in unconditional.
façade	A component whose purpose is to simplify another software component’s interface in order to facilitate loose coupling.
guard	An annotation that can be added to a model transition in UPPAAL for the purpose of specifying the conditions which need to be met for the transition to be enabled.
location	A vertex in a UPPAAL automaton. The active location is part of the state of an individual automaton.
model-checking	A verification method for models which may involve the generation of behavioral traces.

normalization	The process of replacing negations and recognizer-less nodes in a TDL^{TP} abstract syntax tree.
operator arity	The number of operands an operator takes.
parse tree	A tree-based form of intermediate representation which contains the complete syntactical structure of an input sentence.
recognizer	A property recognizing automaton which either collects information from other recognizers in order to notify its parent entity or collects information from the SUT model by inspecting trap variables.
recognizer model	The composition of a collection of recognizers and the test stopwatch into a tree structure based on the AST of a TDL^{TP} expression.
recognizer tree	Used interchangeably with the term *recognizer model*.
reduction	The process of substituting substructures in a TDL^{TP} expression's AST with the aim of reducing its size.
s-expression	A simplified notation for representing tree structures.
scenario composition	The process of combining a SUT model and a TDL^{TP} expression in order to construct a test model.
synchronization	In UPPAAL, a means to specify actions and corresponding co-actions.
template instantiation	Either refers to the process of providing parameters for a UPPAAL automaton template or to the modified template produced as the result of this process.
test generation	The process of generating ATCs using a test purpose and a SUT model.
test model	The model produced as the result of combining a TDL^{TP} expression with a SUT model. Produces an ATC when combined with a suitable UPPAAL TCTL formula.
test purpose	Represents a specific behavioral property that a software tester wishes to verify.
test selection criteria	The collection of coverage criterions used in the formulation of a test purpose.
tester	In the context of this work, an individual or group of people who wish to verify the behavior of a software system or artifact.
transition	A connection between two locations in a UPPAAL automaton.
trap	A member of a trapset which maps to a single transition in a UPPAAL SUT model.
trapset	A collection of traps which connects a TDL^{TP} expression to the corresponding UPPAAL SUT model.

trapset evaluation	An umbrella term for trapset extraction, trapset expression evaluation, and trapset quantifier evaluation.
trapset expression	A TDL^{TP} subexpression whose root is a trapset operator.
trapset expression evaluation	The process of deriving a base trapset from a trapset expression, thereby determining which transitions the expression maps to in the model.
trapset extraction	The process of retrieving the mapping between transitions and trapsets from a user-provided SUT model and a TDL^{TP} expression.
trapset quantifier	A logical operator in TDL^{TP} whose operand domain is the set of possible trapset expressions. Universal quantification over a trapset is true if and only if all the traps in the set have been visited. Existential quantification over a trapset is true if and only if at least one of the traps in the set has been visited.
trapset quantifier evaluation	The process of determining whether a trapset quantifier can be replaced with a Boolean literal and the execution of such a replacement when applicable.

Table of contents

List of figures

List of tables

1 Introduction

In systems design, models find use as representations of structure, data, and behavior. A collection of models based on different views of a system can serve as a reusable, shared repository of knowledge on the system. This is the essence of model-based design (MBD), wherein requirements for systems are represented as models in order to facilitate unambiguous collective understanding throughout the development process.

The abstraction capabilities and reusability offered by models naturally led to their utilization in software verification. One approach in this area where model-usage has shown success is model-based testing (MBT). MBT is a black-box testing method based on MBD which employs models for the behavioral verification of a system under test (SUT)[1]. Figure 1 outlines a possible flow of activities involved in MBT.

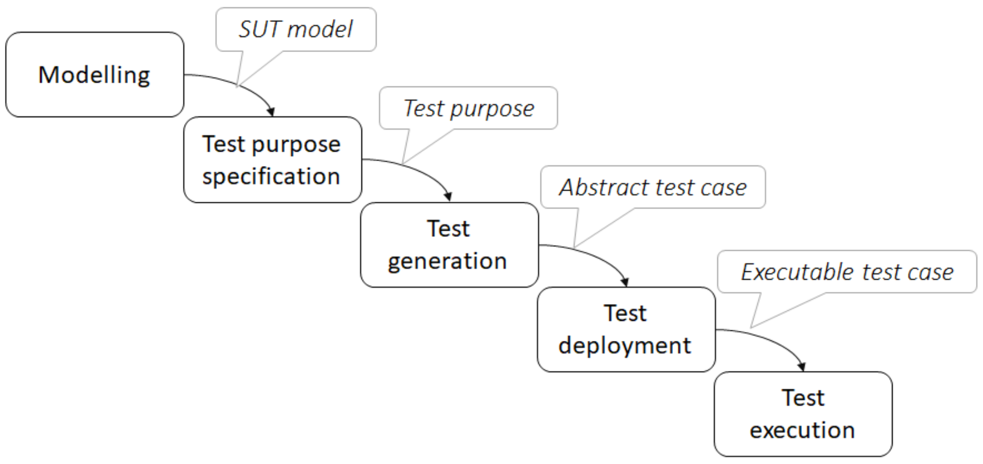


Figure 1. Example MBT Workflow.

First, in the *modelling phase*, a SUT model is constructed based on SUT specifications and a test plan. This model is a representation of SUT behavior whose level of abstraction depends on the scope of the test plan.

[1] Also referred to as the implementation under test (IUT) in the literature.

Test purposes defined during the *test purpose specification phase* represent specific behavioral properties that a software tester wishes to verify. An elementary example of a test purpose is "*Test whether state s_a always leads to state s_b*". In the *test generation* phase, the SUT model and the test purposes serve as inputs for the construction of abstract test cases (ATCs – collectively referred to as the *abstract test suite*).

ATCs produced in the test generation phase represent paths in the state-space of the test model [2, p. 10]. The construction of these paths is required to fulfil the specified test purposes.

In the *deployment phase*, ATCs are transformed into executable test scripts[1] – sequences of input-output pairs that define a concrete testing procedure with respect to the SUT. This can be done *offline*, in which case test scripts are produced before test execution, or *online*, where test stimuli are computed on-the-fly depending on the SUT state and the test goal.

Test scripts serve as the driver for the *test execution* phase, which is defined as the task of feeding the sequence of stimuli specified by a script to the SUT and verifying whether the latter responds as expected [3, p. 3]. Verification of SUT behavior according to a test script will yield a test verdict (*pass*, *fail*, *inconclusive*) based on whether the witnessed SUT behavior conforms to behavior encoded in the model.

Because the number of steps in a test script derived from a SUT model is generally too large for manual runs, test execution requires automation [2, p. 10]. The implication here is that the selected modeling formalism needs to be machine-interpretable. In addition to enabling automated execution, machine-interpretable modeling formalisms also facilitate the automation of ATC generation.

Manual construction of ATCs is a time-consuming, cognitively tasking endeavor for any non-trivial system. This is both due to the potential complexity that SUT models can exhibit and the likely redundancy involved in generating several ATCs from the same SUT model. Consequently, the benefits of automated test case generation are obvious:

[1] Often referred to as *concrete test cases*.

elimination of human error and considerable reductions in the cost of testing. For these reasons, automated test generation was chosen as the broadly stated focus of this thesis.

In order to automate test generation, the test purposes which drive the process should be presented in a formal manner. The general approach in MBT is the adoption of domain-specific languages (DSLs) for test purpose specification. As explained in [1, p. 1 – 2], this becomes an obstacle when attempting to apply these languages to different domains. As a specific example, most existing test purpose specification languages do not fully support the expression of timed behavior, which is essential for testing real-time systems.

UPPAAL Timed Automata (UTA), first presented in [4], is a widely known modeling formalism used for the specification and debugging of dynamic real-time systems. It is based on the theory of Timed Automata (TA), wherein a model is defined as a state transition system (automaton) annotated with timing constraints [5, p. 183]. The primary benefits of UTA are its support for representing timed system behavior and the existence of a mature ecosystem of UTA MBT tools [6].

The UPPAAL toolkit[1] includes a graphical environment – the UPPAAL graphical interface (GUI) – designed for the construction and testing of UTA models. The property specification language available in UPPAAL – a variant of Timed Computation Tree Logic (TCTL) – can be used for test purpose specification. However, while the language has high expressive power, it is syntactically restricted. Namely, UPPAAL only allows for the use of un-nested temporal operators, which "makes the TCTL expressions 'flat' with respect to temporal operators" [1, p. 2]. This restriction prohibits the direct specification of certain complicated test purposes in UPPAAL TCTL. Therefore, the full exploitation of UPPAAL for MBT is only feasible for expert users who know how to augment models with auxiliary property recognizing automata intended for use in combination with TCTL expressions.

A new solution to the limitations of UPPAAL MBT, presented in [1], is an additional language layer called the Test Purpose Specification Language (TDL^{TP}). TDL^{TP} is "free

[1] http://www.uppaal.org

from the limitations of flat TCTL" and provides facilities for expressive test purpose specification [1, p. 2] when integrated into a UTA MBT workflow.

The primary goal of this thesis is to implement an interpreter for TDL^{TP} based on the theory set forth in [1]**.** This interpreter will accept as input:

1. a test purpose specification in TDL^{TP}, and
2. a SUT model specified in the UTA formalism annotated with information relating transitions in the model to ground-level elements of the TDL^{TP} expression[1].

As output, the interpreter will produce a *test model* which implements the test purpose specification implicit in the combination of the expression and the SUT model.

It is expected that the interpreter will find use in a prototype environment for model-based testing of cyber-physical systems, which has been under development at Tallinn University of Technology for several years.

Our approach to implementing the interpreter is a variant of Component-Based Development. More specifically, based on an initial analysis of the sub-functionalities needed by the interpreter, we gradually build the artifact by introducing new components which depend on previously implemented ones. The expectation here is that this development method will enforce extensibility in the interpreter's internal design, and therefore facilitate possible future improvements.

In order to ensure that the interpreter correctly handles user input, a subset of the components involved will be validated by automated tests. On the other hand, tests which ascertain that the interpreter functions according to the theory it was based on will be performed manually. This will allow the project to be completed in a reasonable amount of time.

1.1 Outline

This thesis is organized as follows. Chapter 2 introduces the theoretical concepts needed to understand TDL^{TP}. Chapter 3 presents general requirements for the TDL^{TP} interpreter

[1] The technical term for these annotations is *traps* – 2.3.3 provides a definition.

produced as the result of this work. Chapter 4 provides an overview of the interpreter's architecture. Chapter 5 focuses on the algorithms embedded in the component at the center of the interpreter – the scenario composer. Finally, Chapter 6 summarizes our approach to validation.

2 Preliminaries

This chapter presents the requisite background knowledge for understanding the deployment context of a TDL^{TP} interpreter. The discussion is opened with the most general concept: model-based testing, a black-box testing strategy used for the behavioral verification of systems. Thereafter, we describe a concrete formalism for model-based testing of real-time systems: UPPAAL Timed Automata. Following this, a brief overview of the limitations of UPPAAL TCTL in MBT will lead into an introduction to the focus of this thesis – the Test Purpose Specification Language TDL^{TP}. The chapter is concluded with a brief discussion on abstract syntax trees, which sets the stage for explaining the implementation of the interpreter in Chapter 4.

2.1 Model-Based Testing

Software testing is an investigative activity whose aim is to determine the extent to which a software entity or system meets predefined expectations. An enumerative definition of the term 'expectation' here would cover a wide spectrum of properties from usability to reliability. In this thesis we use the umbrella concept of *correctness* as the evaluative criterion employed in testing. Additionally, we consider the testing task only within the context of systems development.

A system is considered correct if its response to any feasible input stimulus is admissible according to its specification. In *black-box testing* (BBT), system responses to input stimuli are exclusively available in the form of observable outputs – internal implementation details are irrelevant and only the externally perceived behavior of the system matters. The main benefit of BBT is that it forces the tester to approach the system from the perspective of the user as opposed to that of the designer, thus embedding a degree of objectivity in the testing process.

Model-based testing (MBT) is a variant of BBT that relies on models which encode the intended behavior of a system under test [7, p. 1]. Here the term 'model' denotes a simplified, possibly formal representation of an observable phenomenon (in MBT, the behavior of a system). The use of models in MBT is motivated both by their potential for facilitating automation and the support they offer for unequivocal, easily communicable understanding of system behavior.

Since testers inevitably build mental models of a SUT in order to test its intended behavior, it can be argued that *all* testing is essentially based on models. MBT is exceptional under this treatment in the sense that it requires the use of *explicit* models. Roughly speaking, MBT processes exploit reusable models to generate traces of inputs and expected outputs [8, p. 281]. Trace inputs are concretized and passed to a real system, the abstracted outputs of which can then be compared to those specified in the trace, thereby yielding a test verdict – *fail*, *pass*, or *inconclusive*.

An MBT workflow involves the following primary activities: SUT modelling, test purpose specification, test generation, test deployment, and test execution. An example configuration of these activities is depicted in Figure 1.

During the *modelling phase*, requirements for the SUT are used to formulate abstract behavioral models. The level of abstraction depends on the features of the modeling language and the scope of the testing task, generally specified in a test plan.

The assumption in MBT is that the SUT model is *valid*, i.e. that it correctly reflects the behavioral attributes of the system that are subject to testing. Model validity is a wider topic outside the scope of this thesis. Henceforth we assume the validity of SUT models in the context of MBT.

It is important to note here that in practice the SUT will operate in an *environment* which provides the stimuli (inputs) that drive its behavior. Thus, it is often the case that an environment model will be produced in addition to the SUT model – possibly in composition with the latter. As the test model is "exploited for the generation of test cases" [7, p. 4], the corresponding environment model places limits on the set of behavioral paths explored in the test proper. By including an environmental context in this manner, the tester can select specific aspects of the system's behavior for testing.

In the *test generation* phase, *test selection criteria* are used to further select a subset of SUT behavior paths. Test selection criteria (as captured in a *test purpose*), represent attributes of SUT behavior that the tester aims to verify. A relatively simple example of a test purpose is "state s_a is inevitably visited". The range of possible test purposes facilitated by a modeling formalism can be used as a measure of the its applicability for a given MBT workflow.

The entities produced as the output of test generation are referred to as *abstract test cases* (ATCs). [7] presents a succinct definition of an ATC as a "finite structure of input and expected output" [7, p. 2]. This definition elucidates the role of ATCs in MBT. In simple terms, an ATC generalizes a collection of concrete input-output sequences that the SUT's behavior should conform to. The 'abstract' qualifier here implies that ATCs reside on the abstraction level of the SUT model, so some transformative mechanism is required to utilize ATCs in testing. This mechanism is applied in the test deployment phase.

During *test deployment*, ATCs are converted into *executable test scripts* which serve as input for *test executions* against the SUT. The manifest separation between ATCs and executable test scripts supports the notion of platform independence in MBT workflows: test cases can be specified as ATCs in one language, then converted to test scripts in another.

The conversion process involved in transforming ATCs to executable test scripts defines the *mode* of test generation. Based on whether test generation is performed strictly before test execution or interleaved with the latter, MBT is divided into two modes: offline and online, respectively. It should be noted that the these modes merely represent the extremes on a spectrum of possible approaches [6, p. 78 – 79].

In *offline* MBT, test generation occurs prior to test execution. The relative isolation of the two phases in this approach makes it possible to generate tests once and subsequently execute them any number of times. Additionally, the time cost related to generating tests is minimized – an obvious benefit when the cost is not negligible.

In *online* MBT, test generation and test execution are dovetailed. An adapter is provided in the deployment phase as the mediator between the test generator and the SUT. Here the test generator, which encapsulates a model of the SUT and its environment, receives output from the system during the test execution and determines which path the SUT has taken in the state-space of the model. This allows the generator to compute the next test input for the SUT on-the-fly. The reactive nature of online MBT makes it the preferred method for testing systems that exhibit nondeterministic behavior [7, p. 9].

Regardless of the mode of MBT used, the end goal is to produce a *test verdict* as the result of a test execution. An execution *passes* if expected and actual SUT outputs conform, it *fails* if they do not, and it is *inconclusive* when the decision cannot be made [7, p. 3 – 4]. Assuming model validity with respect to the SUT, a *failure* verdict is enough to suggest the existence of a software bug. Its root cause can subsequently be backtracked and resolved.

In summary, MBT is a testing method where explicit models derived from system requirements are used in the process of verifying the behavior of a SUT. The SUT behaves correctly if, given a test input, the outputs produced by the SUT and its model conform to one another. MBT's benefits echo those of model usage in general: clarity, reusability and potential for automation.

2.2 UPPAAL Timed Automata

"UPPAAL is a tool suite for verification of real-time systems, jointly developed by Uppsala University and Aalborg University." [9, p. 1] The modeling formalism made available in the UPPAL toolkit (known as UPPAAL Timed Automata – UTA) is based on the theory of *timed automata* (TA) set forth by Alur and Dill in [5]. In a nutshell, TA is a formal notation for annotating state-transition graphs with timing constraints, thus supporting the modeling and analysis of systems whose behavior is time-sensitive.

UTA extends TA by facilitating the definition of networks of timed automata via its graphical user interface. An example of a small UTA network is presented in Figure 2.

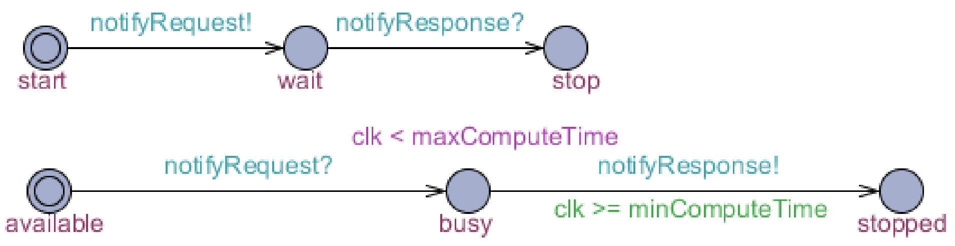


Figure 2. Example UPPAAL automata network.

In the following subsections we provide the minimum requisite syntax and semantics for understanding how UTA could be utilized in an MBT context. For a tutorial introduction to UPPAAL, [9] is suggested.

2.2.1 **Formal Definition of Timed Automata**

In this section we briefly reproduce the basic formal definition of timed automata as provided in [9, p. 2 – 3]. It should be noted that the modeling features suggested by this definition do not represent the entire set of features available in the UPPAAL tool – the interested reader is referred to UPPAAL's online user manual[1] for more details.

In the definitions below, assume C is a set of clocks and $B(C)$ is a set of conjunctions of conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$, and $\bowtie \ \in \{<, \leq, =, \geq, >\}$.

"**Definition 1 (Timed Automaton (TA)).** *A timed automaton is a tuple* (L, l_0, C, A, E, I) *where* L *is a set of locations,* $l_0 \in L$ *is the initial location,* C *is the set of clocks,* A *is a set of actions, co-actions and the internal* τ*-action,* $E \subseteq L \times A \times B(C) \times 2^C \times L$ *is a set of edges between locations, each with an action, a guard and a set of clocks to be reset, and* $I : L \rightarrow B(C)$ *assigns invariants to locations.*" [9, p. 2]

To contextualize Definition 1, two examples of UPPAAL automata are presented in Figure 3.

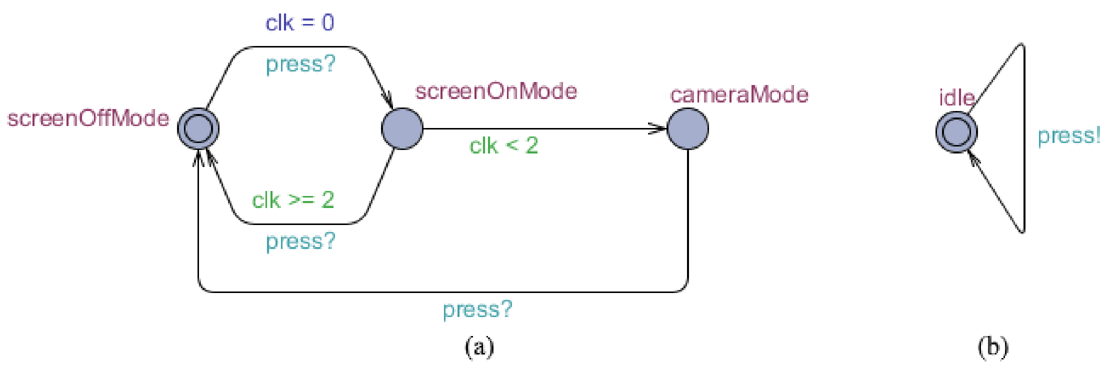


Figure 3. Example UPPAAL automata: (a) smartphone, (b) user.

The figure above depicts a UTA network consisting of models for a smartphone (a) and a human user (b). The human user automaton has a single location (*idle*) and a looping transition attached to said location. The smartphone automaton has three locations: *screenOffMode* (the initial location), *screenOnMode*, and *cameraMode*.

[1] http://www.it.uu.se/research/group/darts/uppaal/help.php

When the power button for the smartphone is pressed by the user once, the screen turns on (*screenOnMode*). Henceforth, when the power button is pressed again, the screen will turn off (*screenOffMode*). However, when the button is pressed twice in rapid succession, the smartphone switches to camera mode instead (*cameraMode*).

The assignment *clk = 0* attached to the transition *screenOffMode → screenOnMode* is a reset of the clock variable *clk*. This variable is used to determine whether the screen should turn off as the result a subsequent press (guard *clk ≥ 2* on transition *screenOnMode → screenOffMode*), or switch to camera mode (guard *clk < 2* on transition *screenOnMode → cameraMode*). The labels *press!* and *press?* – which are synchronizations over the channel *press* – denote a specific action and its co-action, respectively.

A location invariant (not depicted in Figure 3) can be described as a conditional expression that must be true at any time the automaton is in the corresponding location. Put simply, the automaton must exit the location prior to the moment the invariant no longer holds. A common use case for invariants in UPPAAL is the modeling of progress conditions.

To define the semantics of TA, we use the following assistive notions: let $u : C \to \mathbb{R}_{\geq 0}$ denote a *clock valuation* and $\mathbb{R}^C$ denote the set of all clock valuations. For all $x \in C$, let $u_0(x) = 0$ (all clocks start at 0). Guards and invariants are considered sets of clock valuations for the purposes of Definition 2 below, e.g. $u \in I(l)$ means u satisfies the invariant of location l.

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

- $(l, u) \overset{d}{\to} (l, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \implies u + d' \in I(l)$, *and*

- $(l, u) \overset{a}{\to} (l', u')$ *if there exists* $e = (l, a, g, r, l') \in E$ *[such that]* $u \in g$, $u' = [r \mapsto 0]u$, *and* $u' \in I\ (l)$, *where for* $d \in \mathbb{R}_{\geq 0}$, $u + d$ *maps each clock* x *in* C *to the value* $u(x) + d$, *and* $[r \mapsto 0]u$ *denotes the clock valuation which maps each clock in* r *to* 0 *and agrees with* u *over* $C \setminus r$" [9, p. 3].

[9, p. 4] elucidates Definition 2 as follows: "From a given initial state, we can choose to take an *action* or a *delay* transition […]. Depending [on] the chosen delay, further actions may be forbidden." If more than one action is enabled, the choice between them is made non-deterministically.

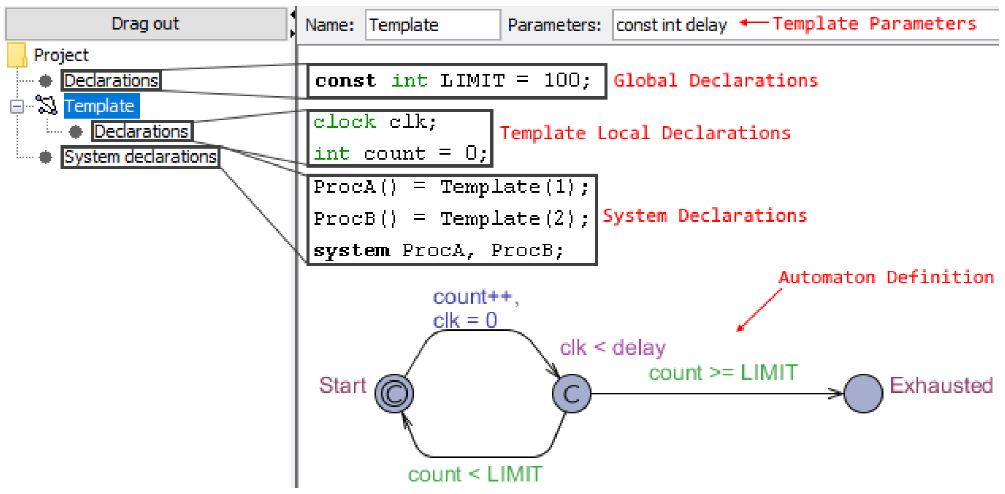
TA can be composed into *networks of timed automata*. Such networks consist of $n > 1$ timed automata $T_1, \ldots, T_n$ which share a common set of clocks and actions [9, p. 4]. The state of a TA network at time t is given by the tuple $\langle \bar{l}, u \rangle$. Here, $\bar{l} = \langle l_1, \ldots, l_n \rangle$ is a location vector where l_i is the location of automaton T_i at time t, and u is a valuation for all clocks in the network. In UPPAAL, TA networks share a set of global variables.

For a full definition of the operational semantics of UTA, the reader is advised to review [10].

2.2.2 **UPPAAL Modelling Language**

In this section we briefly present some core UPPAAL language features in the interest of self-containment. An exhaustive listing is available in the UPPAAL manual[1].

A UPPAAL model consists of the following basic elements: global declarations, automata templates, and system declarations. These elements are illustrated in Figure 4.



Name: Template
Parameters: const int delay

```
const int LIMIT = 100;
```

```
clock clk;
int count = 0;
```

```
ProcA() = Template(1);
ProcB() = Template(2);
system ProcA, ProcB;
```

Figure 4. Core UPPAAL language features.

The user can define variables whose scope covers the entire model in the *global declarations* section. Variable types include (but are not limited to): clocks, synchronization channels, integers, Booleans, records, and multidimensional arrays of the previously listed types. We will refer to the C-like sublanguage UPPAAL uses for these declarations as the *UPPAAL system definition language*.

UPPAAL automata are defined as named parameterized *templates* – directed multigraphs whose nodes are locations and whose edges are transitions.

Automata transitions can be supplied with guard, assignment, synchronization, and selection labels, while locations can be decorated with an invariant label. These labels can access variables from the global scope as well as the local scope of the template.

[1] http://www.it.uu.se/research/group/darts/uppaal/help.php

Variables local to a template are defined in the corresponding *template declarations* section. The local scope of a template also includes its parameters.

Locations can be marked as *initial*, *urgent*, or *committed*. A template must have exactly one initial location (displayed as a double circle) but may optionally have any number of either *urgent* or *committed* locations (the latter two exclude one another and are marked with an uppercase 'U' or 'C', respectively).

Per [9, p. 6], "time is not allowed to pass when the system is in an urgent location". Committed locations further restrict the behavior of the automaton: "A state is committed if any of the locations in the state is committed. A committed state cannot [cause a] delay and the next transition must involve an outgoing edge of at least one of the committed locations" [9, p. 9]. The difference between the two is that urgent locations allow interleaving with other automata [6, p. 84].

In order to compose a network of automata in UPPAAL, the requisite templates should be instantiated inside the *system declarations* section and listed as part of its *system line*. It should be noted here that a template can be instantiated partially (leaving some parameters unbound). This facilitates the compact declaration of automata subtypes.

Template instantiations enumerated in the system line represent *processes* that are part of the system model. Local template variables for processes instantiated from the same template are independent from one another. In other words, a template's local declarations are prototypes for variables belonging to the local scopes of processes spawned from said template.

In summary, UPPAAL timed automata are declared as named parameterized templates. The latter are directed multigraphs with transitions as edges and locations as vertices. Transitions and locations may be annotated with one or more labels. Labels act as behavior specifiers and, as such, can reference state variables from both the model's global scope and the template's local scope (including the parameters of the template). A UPPAAL system model is constructed from UTA templates via composition. The model processes that form the system are enumerated as part of the system line in the model's system declarations section.

2.2.3 UPPAAL in Model-Based Testing

In this section we briefly introduce UPPAAL's requirement specification language and outline an MBT test generation method built on it. For a baseline treatment of UPPAAL in MBT, [6] is recommended. Further, [11] describes a provably correct test development method for timed systems based on the UTA formalism.

The UPPAAL toolkit includes a model-checking utility called verifyta. Per [9, p. 7], "the main purpose of a model-checker is [to] verify the model [with respect to] a requirement specification." In order to facilitate the formalization of such requirements, UPPAAL offers a limited version of Timed Computation Tree Logic (TCTL).

UPPAAL's flavor of TCTL is referred to as the UPPAAL query language (UQL) in this section. Expressions in UQL can be applied to UPPAAL models using the *verifyta* tool. In addition to merely outputting whether the expression is satisfied by the input model, verifyta can generate symbolic witness traces demonstrating how the verdict was reached. These traces can be exploited subsequently for offline MBT.

UQL can be used to specify *state* and *path* formulae. The former describe individual model states and the latter quantify path formulae over model traces. A state formula can be as simple as `stopwatch.done` which asserts that the process `stopwatch` is in the `done` location. Path formulae, however, are classified into *reachability, safety,* and *liveness* properties. In what follows, only reachability properties will be considered because of their significance for UPPAAL MBT. An introduction to safety and liveness properties is provided in [9, p. 8 – 9].

Reachability properties ask whether a given state formula, `p`, can possibly be satisfied by the model. The UQL syntax for expressing the proposition that some state satisfying `p` can be reached by the model is `E<>p`. Figure 5 depicts an example state-space search where the marked node represents the state where reachability property `p` has been satisfied, and the bold arrows specify a path towards this state.

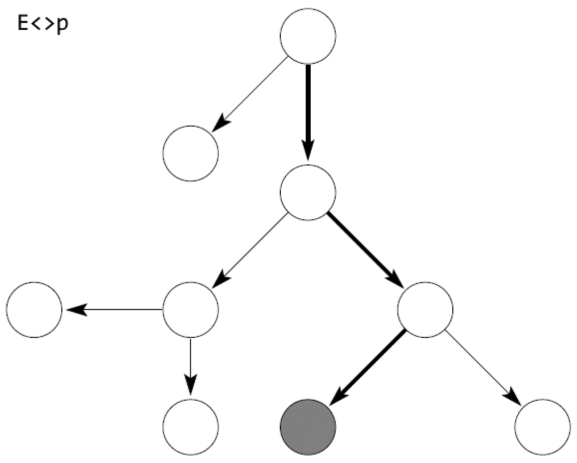


Figure 5. Example state-space search for reachability property `E<>p` [9, p. 8].

[6, p. 91 – 94] describes a procedure for generating MBT test cases using reachability properties derived from test purposes[1]. In broad terms, either the test purpose is directly convertible to a simple reachability property, or the model needs to be decorated with utility constructs (e.g. globally scoped Boolean flag variables) in order to facilitate the expression of the test purpose. By generating a symbolic trace affirming that the property holds for the model (using verifyta), an abstract test case has essentially been constructed. The latter will be subject to adjustment before it can be used to test the modelled system.

While UQL is a powerful tool, it is not entirely without limitations in terms of ease-of-use. The language is limited in that path formulae do not allow nesting (for example, `E<>(x imply E<>y)` is incorrect UQL syntax). This makes it difficult to express certain complex test purposes, requiring testers to substantially modify their models in order to fully exploit UTA for MBT. A more powerful alternative for test purpose specification suited for both offline and online UTA MBT is introduced in 2.3.

In summary, test generation can be formulated as a model-checking task in UPPAAL. Namely, test purposes are formalizable as reachability properties in UPPAAL's property specification language. A reachability property asks whether a given state configuration can possibly be reached by the model. When provided with a SUT model and a reachability property, UPPAAL's model-checking tool verifyta can generate a witness trace that represents an ATC for the SUT. However, due to the manifest limitations of the

[1] Test purposes are introduced in Section 2.1.

abovementioned query language (lack of support for nested properties), an alternative approach is needed.

2.3 Test Purpose Specification Language (TDL^{TP})

As explained in 2.2.3, UPPAAL has usability limitations when applied in an MBT context. UPPAAL's property specification language, while relatively expressive, does not support nested reachability properties and is thus unsuited for the direct specification of certain complex test purposes. In this section we introduce a solution to the problem – the Test Purpose Specification Language (TDL^{TP}), an abstract notation language relevant for constructing UPPAAL test models that encode advanced test purposes. The following subsections are based on the theory presented in [1].

2.3.1 **Test Purpose Specification via TDL^{TP} Expressions**

Using TDL^{TP}, a test purpose is specified as an expression. When applied to a SUT model annotated with trap variables[1] according to the logic laid out in [1, pp. 4 - 11], an interpretation of the test purpose expression can be used to transform the SUT model into a *test model*. The latter is simply the composition of the original SUT model, a *test stopwatch*, and a collection of *property recognizing automata* (henceforth *recognizers*). The behavior of the test model is identical to that of the original SUT model. However, the supplementary recognizers can be exploited in order to generate symbolic traces representing test purpose conformant ATCs.

The recognizers in a test model are implicitly organized into a tree whose structure mirrors that of the corresponding TDL^{TP} expression's abstract syntax tree (AST)[2]. More precisely, each recognizer uniquely maps to a subexpression of the TDL^{TP} expression. An individual recognizer thus has a single parent recognizer[3] and zero or more child recognizers, analogously to the AST node it represents. This structuring is achieved via globally scoped synchronization channels that function as dedicated means of

[1] Explained in Section 2.3.3.

[2] The notion of an AST is explained in Section 2.4.

[3] Except the root recognizer, which has the test stopwatch as its parent.

communication between recognizers. An example mapping between a test purpose expression and its recognizer tree is depicted in Figure 6.

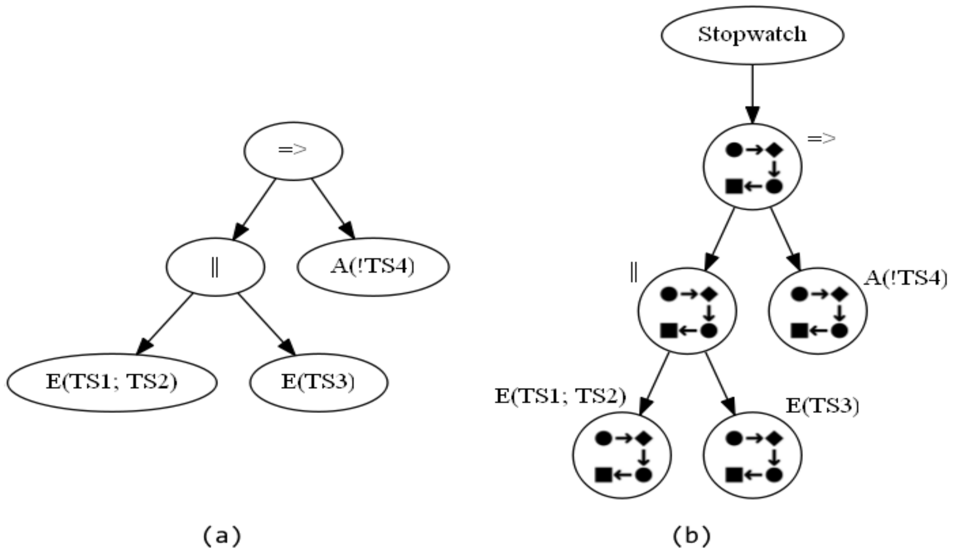


Figure 6. Comparison of TDL^{TP} AST (a) and corresponding recognizer tree (b).

The test stopwatch (described in Section 8 of [1]) has two terminal locations – one representing test success and the other failure. It is attached to the root node of the recognizer tree. When UPPAAL's model-checker is invoked with the goal of affirming that the success location of the stopwatch can be reached in some model state, the stopwatch sends an initializing signal down to its child (the root recognizer). This signal is subsequently propagated to the leaves of the recognizer tree in a depth-first manner. As the result of this process, all the recognizers in the test model enter their recognizing states. Essentially this means that they begin to wait for their corresponding TDL^{TP} subexpressions to be satisfied.

When a subexpression has been satisfied at some step in the model-checker's state-space search, the corresponding recognizer resets itself and notifies its parent with a success signal. This signal is subsequently propagated up the recognizer tree (according to the logic encoded in each recognizer along the path), until it can reach the test stopwatch. When the stopwatch receives a success signal from its child, the state-space search can be terminated. The resultant symbolic trace generated by UPPAAL's model-checker represents an abstract test case which, when concretized for the SUT, can be used for testing[1].

In what follows we will not consider the details of integrating TDL^{TP} into a UPPAAL MBT workflow. Instead we treat the theory of the language as the specification of a tool which accepts as input a trapset annotated SUT model and a TDL^{TP} test purpose expression. As output, the tool produces a test model per the logic described in [1]. Subsections 2.3.2 – 2.3.5, give a concise description of the syntax and semantics of TDL^{TP}.

2.3.2 **Expression Syntax**

The syntax for TDL^{TP} expressions is presented in Backus-Naur form (BNF) in Figure 7.

```
<Expression> ::= '(' <Expression> ')'
               | 'A' '('<TrapsetExpression>')'
               | 'E' '('<TrapsetExpression>')'
               | <UnaryOp> <Expression>
               | <Expression> <BinaryOp> <Expression>
               | <Expression> ~> <Expression>
               | <Expression> ~> '['<RelOp><NUM>']' <Expression>
               | '#'<Expression><RelOp><NUM>

<TrapsetExpression> ::=!'<ID>
                      | <ID> '\' <ID>
                      | <ID> ';' <ID>
                      | <ID>
s
<UnaryOp>      ::= 'not'
<BinaryOp>     ::= '&' | 'or' | '=>' | '<=>'
<RelOp>        ::= '<' | '=' | '>' | '<=' | '>='
<ID>           ::= ('TS')<NUM>
<NUM>          ::= ('0'...'9')+
```

Figure 7. TDL^{TP} expression grammar in BNF form [1, p. 4].

The leaf nodes of a TDL^{TP} expression are references to *trapsets* (denoted `TS1`, `TS2`, …). Trapset references can appear as leaves on their own or as the operands of a trapset expression – the former is just an edge case of the latter.

A trapset expression always appears as the immediate child of a *trapset quantifier*, of which there are two types: universal (`A`) and existential (`E`). These, in turn, are the operands

[1] In this thesis the *test model* is referred to as an abstract test case instead as it contains the information needed to generate symbolic traces.

for TDL^{TP} logical operators, which are classified as either connectives or temporal operators. In the following subsections we introduce the basic semantics for each type of TDL^{TP} expression node.

2.3.3 **Semantics of Trapsets**

A trapset in TDL^{TP} is defined as a set of assignments to Boolean *trap variables* (*traps*) [1, p. 4]. Each trap is mapped to a single transition in the SUT model via an assignment label[1]. If the right-hand side of a trap assignment is syntactically reducible to a Boolean literal, the corresponding trap is classified as *elementary*, otherwise the trap is classified as *conditional*. An example of two trapsets in a UPPAAL model fragment is depicted in Figure 8, Section 2.3.4.

During a simulation run of the model, the value of a specific trap variable at any given instant denotes whether the corresponding transition has been taken[2]. According to this interpretation, trapsets are an extension of the model transition coverage criterion described in [6, p. 93]. Roughly put, trapsets allow higher-level TDL^{TP} expression operators to reason with groupings of transitions instead of individual transitions.

2.3.4 **Semantics of Trapset Expressions and Trapset Quantifiers**

Per the BNF grammar in Figure 7, trapsets can appear as simple references or as the operands of *trapset expressions* in a test purpose expression. Trapset expressions represent trapsets that can be derived from their operands based on Mappings M1 – M3 provided in Section 5.1 of [1]. We use the term *base trapset* to distinguish explicitly mapped trapsets from trapset expressions.

Trapset expressions are divided into three types based on their root operators: *absolute complement*, *relative complement*, and *linked pair*. In what follows, each type of trapset expression is briefly introduced. For contextualization, an example model fragment containing two trapsets is provided in Figure 8.

[1] UPPAAL assignment labels are introduced in 2.2.2.

[2] Since the last time the trap variable was reset. For conditional traps, the condition simply restricts the notion of *taken* for the transition in question.

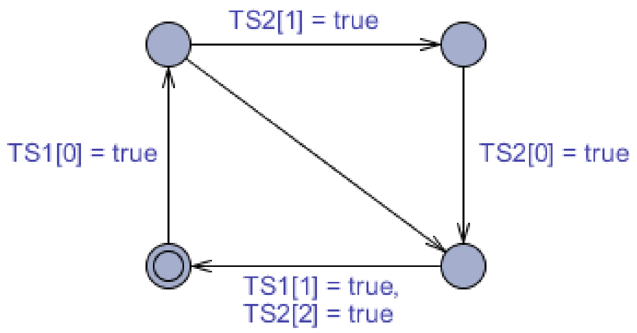


Figure 8. Example model fragment with two trapsets: `TS1` & `TS2`.

The *relative complement* of trapsets `TS1` and `TS2` (denoted `TS1\TS2`) maps to all transitions that are included in `TS1` and excluded from `TS2`. A formal definition is provided in Section 4.2 (2) of [1]. Mapping M1 in Section 5.1 of [1] describes how the relative complement of two trapsets can be used to derive an equivalent base trapset. The figure below depicts the result of applying a relative complement operator on the two trapsets from Figure 8.

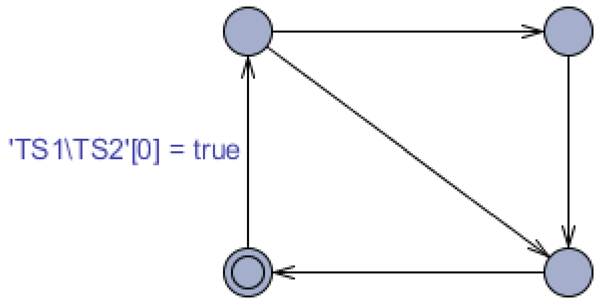


Figure 9. Relative complement of trapsets `TS1` & `TS2` from Figure 8.

The *absolute complement* of a trapset `TS` (denoted `!TS`), maps to all transitions that are excluded from `TS`. A formal definition is provided in Section 4.2 (3) of [1]. Mapping M2 in Section 5.1 of [1] describes how the absolute complement of a given trapset can be used to derive an equivalent base trapset. The figure below depicts the result of applying the absolute complement operator on trapset `TS1` from Figure 8.

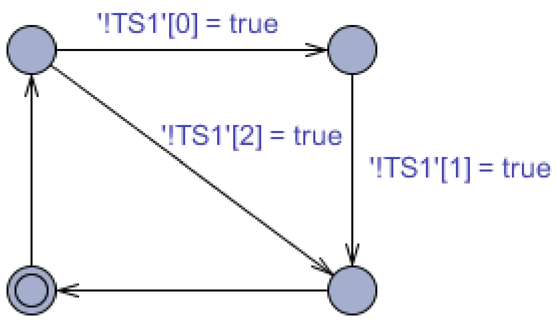


Figure 10. Absolute complement of trapset `TS1` from Figure 8.

The *linked pairs trapset* of trapsets `TS1` and `TS2` (denoted `TS1; TS2`) maps to all transitions of `TS2` whose starting location is a terminal location for some transition in `TS1`. Such transition pairs are known as *linked pairs*. A formal definition is provided in Section 4.2 (4) of [1]. Mapping M3 in Section 5.1 of [1] describes how an equivalent base trapset can be produced for this trapset expression type. Figure 11 depicts a linked pairs trapset generated from the trapsets shown in Figure 8.

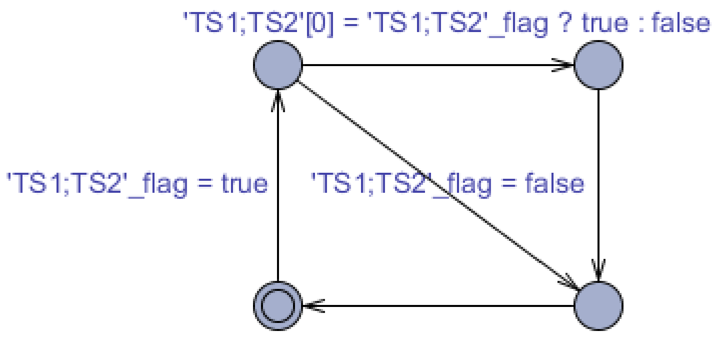


Figure 11. Linked pairs trapset of trapsets `TS1` & `TS2` from Figure 8.

In order to succinctly state whether all or some traps must be true or false at once, TDL^{TP} introduces *trapset quantifiers* of two types: existential (`E`) and universal (`A`). These operators only admit trapset expressions as operands.

Given a trapset expression `TS`, `E(TS)` states that *some* trap of the set `TS` must be true and `A(TS)` states that *all* traps of the set `TS` must be true. In terms of the recognizer tree concept introduced in 2.3.1, trapset quantifiers correspond to leaf recognizers. Mappings M4 – M5 in Section 5.2 of [1] describe how trapset quantifiers map to recognizing automata. It should be noted that trapset quantifiers can appear with a negation modifier in a test purpose expression. In this case the semantics from equivalences (14) – (15) in Section 4.3 of [1] apply.

2.3.5 **Semantics of Logical Connectives and Temporal Operators**

TDL^{TP} supports the following logical operators: *disjunction*, *conjunction*, *implication*, *equivalence*, *leads to*, *timed leads to*, and *bounded repetition*. A TDL^{TP} logical operator can have either a trapset quantifier or another logical operator as an operand. In broad terms, these operators represent propositions about the SUT model's current trace during a simulation or state-space search. Logical operators are considered separate from trapset quantifiers here because their operand domains do not overlap, even though they are similar in terms of recognizer output.

Logical operators are further classified into connectives and temporal operators. This distinction is immaterial in the context of this thesis. Put simply, the recognizer architecture for logical connectives is based on their counterparts in classical logic, while recognizers for temporal operators facilitate the expression of conditions applying to entire traces or subsequences of traces.

The negation modifier known from standard logic is made available for all logical operators. Per [1, p. 7], negation will not map to a separate recognizer and should instead be eliminated from top level expression nodes by the negation rules defined by equivalences (14) – (22) in Section 6 of [1]. Essentially, when a negation applies to a logical operator in a TDL^{TP} expression, it should be removed by the application of reduction rules so that the resultant expression only has negations at the level of trapset quantifiers.

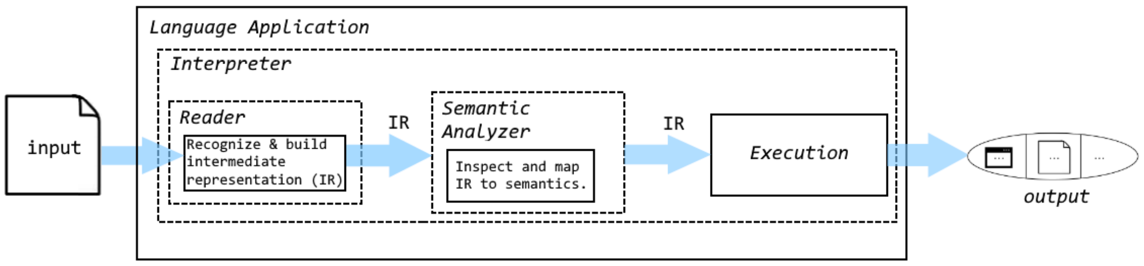
Items (7) – (13) in Section 4.3 of [1] supply formal definitions for each logical operator type. Recognizer generation for these operators is described in Mappings M6 – M10 in Section 5.2 of [1]. It should be noted that such mappings do not currently exist for *implication* and *equivalence* since they can be replaced by lower level operators [1, p. 7].

Section 6 of [1] provides some reduction rules whose application to a TDL^{TP} expression prior to recognizer generation could result in a smaller recognizer tree, and, consequently, a smaller test model. A reduction in the size of a test model which preserves the model's input-output behavior, will lead to a reduction in its state-space. Model-checking algorithms (as a category of state-space search algorithms) are faster for smaller state-spaces, as evidenced by the fact that UPPAAL's model-checker computes results faster for reduced models.

2.4 Abstract Syntax Trees

To provide some context for the sections of this thesis which discuss the implementation of the TDL^{TP} interpreter, we will briefly introduce the terms *parse tree* and *abstract syntax tree* in this section.

[12, p. 20 – 21] describes a typical language application as a pipeline of interrelated components. A simplified version of this pipeline is provided in Figure 12.



Language Application
Interpreter
input
Reader
Recognize & build intermediate representation (IR)
IR
Semantic Analyzer
Inspect and map IR to semantics.
IR
Execution
output

Figure 12. Language application pipeline (adapted from Figure 1.1 in [12, p. 21]).

The interpreter subcomponent in Figure 12 contains a *reader* and a *semantic analyzer*. The reader builds a data structure called an *intermediate representation* (IR) according the input it receives from the user. The input for a language application could be anything from a text file to binary data.

After the reader parses the input, the resultant IR is forwarded to the semantic analyzer. This component examines and collects information from the IR according to the semantics of the language. Put simply, the semantic analyzer will assign a meaning to the input [12, p. 21].

After a meaning has been assigned (possibly through the use of an auxiliary data structure), the IR is used to drive the *execution* performed by the interpreter. As the result of execution, information will be produced in some output format. The format depends on the application – it can range from simple terminal output to program code.

In the process described above – after the input is parsed by the reader into an intermediate form – components in the pipeline will examine the IR instead of reparsing the input. To facilitate this, the IR should [12, p. 92]:

1. Contain no unnecessary information;
2. Be easily traversable;
3. Emphasize the relationship between the abstract constructs of the language rather than the tokens of its grammar.

Abstract syntax trees (ASTs) are a commonly used form of IR which satisfies the three requirements presented above. For a given input, the corresponding AST is a tree structure which reduces the input to its essential elements. A possible AST for the UPPAAL assignment statement '`flag = true;`' is depicted in Figure 13.

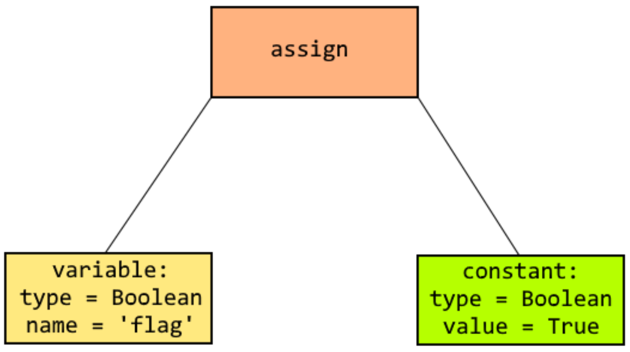


Figure 13. Example AST for the assignment statement ‘`flag = true;`’.

A *parse tree* (also known as a syntax tree) is another form of tree-based IR. Parse trees represent the complete syntactical structure of an input sentence. An example parse tree for ‘`flag = true;`’ is presented Figure 14.

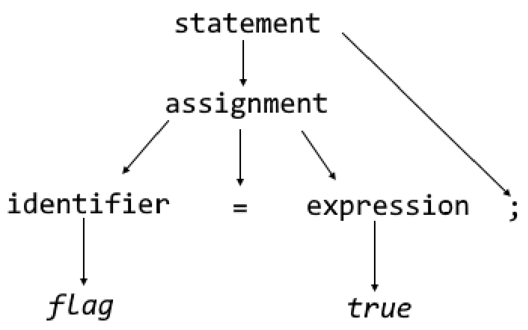


Figure 14. Example parse tree for the assignment statement ‘`flag = true;`’.

As can be seen in the figure above, a parse tree is far less compact and contains unnecessary information compared to the corresponding AST. Abstract syntax trees generalize the notion of parse trees and represent the input conceptually rather than structurally.

As will be shown in Chapters 4 and 5, ASTs are involved in many phases of the test model construction algorithm implemented by the TDL^{TP} interpreter.

3 Requirements

This chapter provides informal requirements for the interpreter implementation produced as the result of this thesis.

Figure 15 illustrates how a human user (a software tester) interacts with the interpreter, and how the interpreter responds.

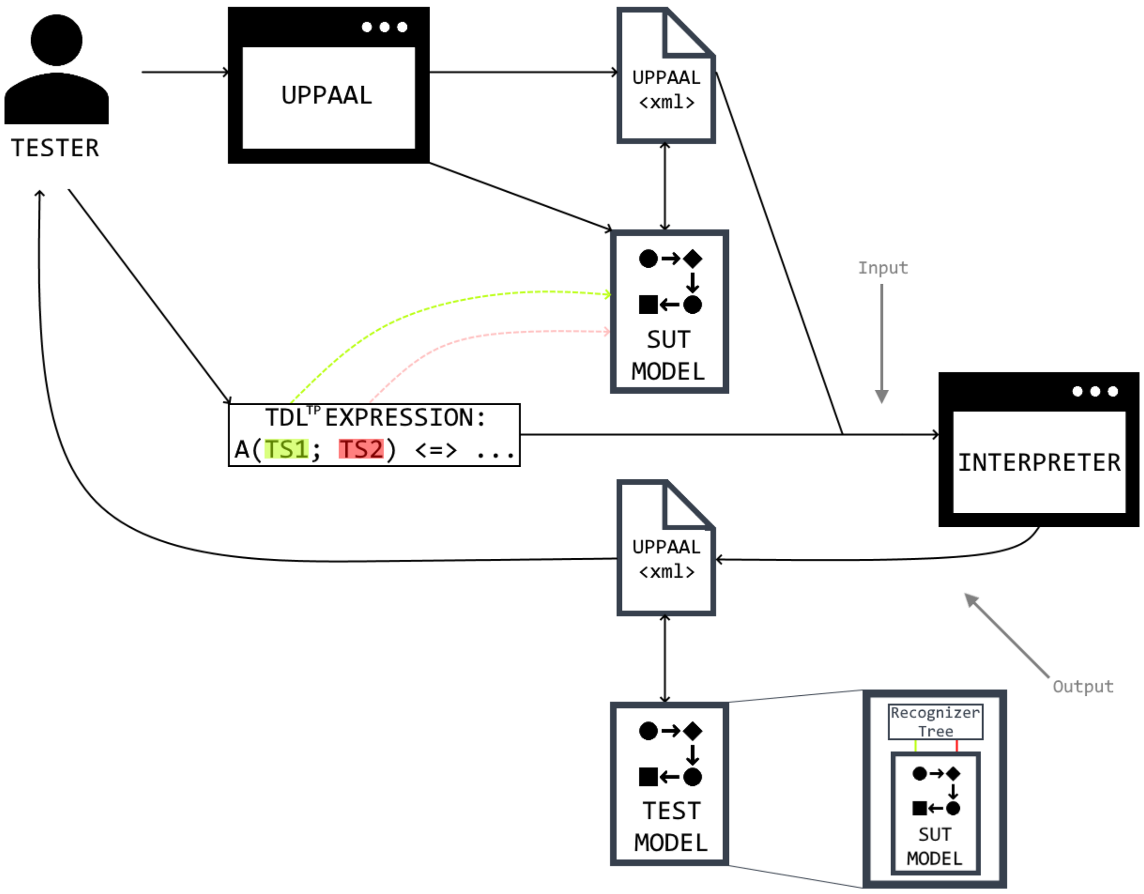


Figure 15. Interaction with the TDL^{TP} interpreter.

In the interaction, the software tester provides the interpreter with a UPPAAL SUT model and a TDL^{TP} expression. At the time of writing, the latest version of UPPAAL provides facilities for storing models as XML files – this is the format in which the interpreter shall accept models as input. As for the TDL^{TP} expression, the interpreter should be able to receive it as a simple string or plain text file.

As described in [1, p. 4], the SUT model should be linked to the TDL^{TP} expression via trapset labels[1]. It is the responsibility of the tester to ensure that this labeling is available in the SUT model provided for the interpreter.

Having received the requisite inputs, the interpreter shall use them to generate an XML file that represents a test model. The logic used for the generation of this test model should be equivalent to the theoretical derivation procedure described in [1]. The user should be able to open the output XML file which encodes the test model in UPPAAL, i.e. the format of the XML file should match the format used by UPPAAL for model storage.

We note here that no explicit restriction shall be placed on the lower bound of the performance of the interpreter. Obviously, it should produce the output XML in a sensible amount of time, but the term 'sensible' will not be attached to a concrete metric here. This concession will allow for more time to be spent on ensuring the correctness of the implementation.

To conclude this chapter, we present some capabilities that the interpreter should be instrumented with so that it could fulfil the requirements presented above. Since the generation of a test model involves processing UPPAAL models and TDL^{TP} expressions, an implementation of the interpreter needs to be able to (at minimum):

1. parse TDL^{TP} expressions;
2. parse and generate UPPAAL XML files;
3. parse and generate code in UPPAAL's system definition language[2].

All of the capabilities listed above were integrated into the interpreter produced as the result of this thesis. Our approach to implementing them is described in Chapter 4 below. Chapter 5, on the other hand, focuses specifically on the algorithm we developed for composing a test model based on a SUT model and a TDL^{TP} expression.

[1] Trapsets are explained in Section 2.3.3.

[2] The UPPAAL system definition language is described in Section 2.2.2.

4 Implementation

In this chapter we provide a high-level description of the implementation of the TDL^{TP} interpreter.

We begin in Section 4.1 by introducing the development approach used to produce the artifact – Component-Based Development. Following this, the artifact's component structure is outlined in Section 4.2. Then, after listing the most relevant technological decisions in Section 4.3, Section 4.4 will present overviews for a selection of crucial components. A user guide for the interpreter is available in Appendix 9.

4.1 Approach

The development approach used for implementing the interpreter is a customized variant of Component-Based Development (CBD). In this thesis we define a *component* as a software entity (module) that either:

a. aggregates and provides access to a collection of domain-specific or general-purpose object templates (classes), or
b. encapsulates complex logic that operates on facilities provided by lower-level components and provides a simplified reusable interface to top-level dependent components or users.

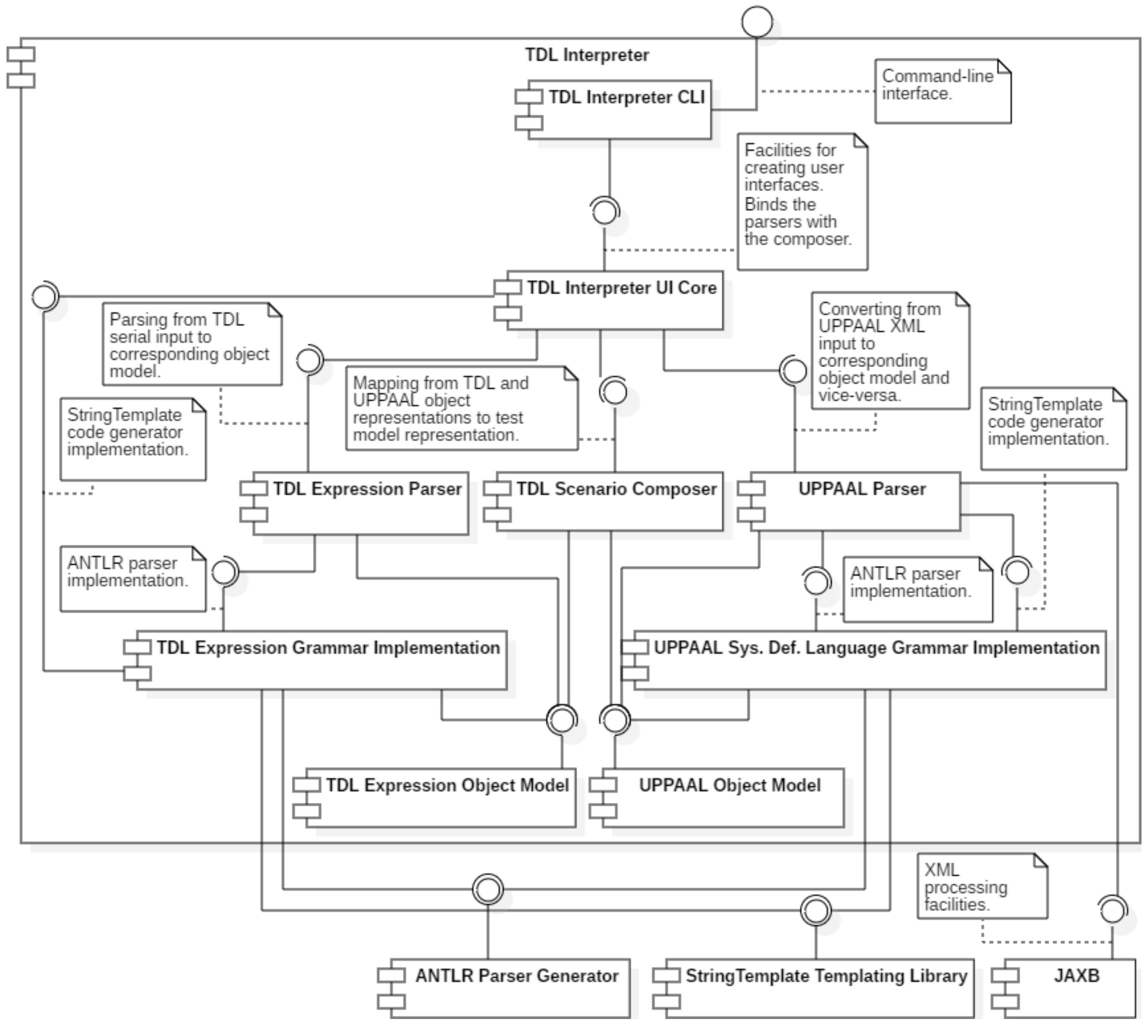
Based on this definition, we define CBD as an approach to software development where the resultant artifact is a composition of relatively isolated components which communicate via software interfaces.

We include a managerial dimension to this definition – in our version of CBD, development is organized per component in a bottom-up manner starting with the components whose dependencies are only external. This allows for revisions of the original component structure during later stages of development. Emergent common logic can be extracted to shared components, thus reducing duplication and simplifying future refactoring efforts.

The basis of our approach was an initial component structure subject to repeated modification during the development process. In Section 4.2 we describe the finalized component structure of the TDL^{TP} interpreter.

4.2 Component Structure

A component diagram for the TDL^{TP} interpreter is provided in Figure 16.



TDL Interpreter
TDL Interpreter CLI
Command-line interface.
Facilities for creating user interfaces. Binds the parsers with the composer.
TDL Interpreter UI Core
Parsing from TDL serial input to corresponding object model.
Converting from UPPAAL XML input to corresponding object model and vice-versa.
StringTemplate code generator implementation.
Mapping from TDL and UPPAAL object representations to test model representation.
StringTemplate code generator implementation.
TDL Expression Parser
TDL Scenario Composer
UPPAAL Parser
ANTLR parser implementation.
ANTLR parser implementation.
TDL Expression Grammar Implementation
UPPAAL Sys. Def. Language Grammar Implementation
TDL Expression Object Model
UPPAAL Object Model
XML processing facilities.
ANTLR Parser Generator
StringTemplate Templating Library
JAXB

Figure 16. TDL^{TP} interpreter component diagram.

The components depicted above do not encompass the entire set of artifacts produced as the result of this thesis. Omissions include shared utility components, test-specific supplementary components, and interface simplifying façade modules[1]. Where applicable, the omitted components will be mentioned in Section 4.4.

In Table 1, we describe the internal[2] components displayed in Figure 16.

Table 1. Core TDL^{TP} interpreter components and their purposes.

Component name	**Purpose**
TDL^{TP} expression object model	Define classes for representing TDL^{TP} expressions as object structures.
UPPAAL object model	Define classes for representing UPPAAL models as object structures.
TDL^{TP} expression grammar implementation	Provide a parser and a code generator[3] for TDL^{TP} based on its grammar.
UPPAAL system definition language grammar implementation	Provide a parser and a code generator for the UPPAAL system definition language.
TDL^{TP} parser	Simplify the interface provided by the TDL^{TP} grammar implementation and transform the latter's parse results into the internal TDL^{TP} expression object model.
UPPAAL parser	Provide a simplified interface for the UPPAAL grammar implementation and transform the latter's parse results into the internally used object model. Provide logic for parsing UPPAAL XML.
Scenario composer	Encapsulate logic for applying a TDL^{TP} expression to a UPPAAL model according to the rules described in [1].
TDL^{TP} interpreter UI (user interface) core	Provide a basis for implementing user interfaces for the interpreter.
TDL^{TP} interpreter CLI (command-line interface)	Provide a concrete user interface implementation based on the UI core.

[1] In this thesis we define a façade as an abstraction component which simplifies the interface of some lower-level component and hides the latter's implementation details from higher-level dependent components.

[2] Put simply, *internal components* are those presented inside the larger *TDL interpreter* component shown in Figure 16. Components displayed outside of the boundaries of the interpreter component are considered *external*.

[3] We define a code generator (or, more exactly, a *source code generator*) as a software entity which accepts object structures as input and produces code in some predetermined language as output.

In accordance with the development approach outlined in Section 4.1, the interpreter was implemented in a bottom-up manner starting with the components that only have external dependencies (or no dependencies at all). In this respect, Figure 16 serves as a historic roadmap for the development effort when visually traversed beginning from the bottom-most independent components. Section 4.4 provides information on the implementations of the components discussed above.

4.3 Technological Choices

This section presents major *technological choices* made during the development of the components introduced in the previous section. We define a technological choice as the *decision to use a specific programming language, library or software framework to facilitate some required functionality*. Here, the term *required functionality* refers to the facilities an individual component needs in order to fulfil its purpose, rather than something a user could expect from the interpreter. Subsections 4.3.1 – 4.3.6 present technological choices in order of scope, with more impactful items occurring first.

4.3.1 **Programming Language: Java**

Because of the diversity of concepts involved[1], an object-oriented general-purpose language with a wide collection of supporting libraries is best-suited to implement the interpreter. In order to save time, the main contenders were restricted to three languages whose popularity has at least stayed consistent since their inception – Python[2], Java[3] and C++[4]. For any given issue, assistive libraries are simply more likely to exist for mainstream languages such as these.

The primary considerations used for evaluating each language were ease-of-use and platform-independence. As stated in Section 3, performance is not a significant concern for the interpreter.

[1] XML, domain-specific languages, and object manipulation to name a few.

[2] https://www.python.org.

[3] https://www.java.com.

[4] https://isocpp.org.

Of the languages in question, **Python** is arguably the easiest to learn and use. As it is an interpreted language, platform-independence is also not an issue. However, since a component-oriented approach has been chosen (per Section 4.1), the language is not a definite front-runner. More specifically, while Python's high-level syntax and dynamic typing allow for faster development, these features do not facilitate rigorous structuring and clear component interfaces without manual boiler-plate constructs such as type hints[1].

With performance not under consideration, **C++** offers little to tip the scales in its favor. The low-level constructs and operations made available in the language are redundant for the task at hand and serve more as cognitive overhead. Another detriment is the fact that C++ is not platform-independent – an artifact written in the language will be compiled for a specific target environment.

Java is an object-oriented general-purpose programming language. According to StackOverflow's 2018 developer survey[2], Java was ranked as the fifth most popular programming language in the developer community. Considering the language's relatively robust syntax, its built-in organizational facilities (packages, modules) and its platform-independence, Java was chosen as the programming language in which to write the interpreter.

4.3.2 **Dependency Management: Maven**

When a software artifact is expected to rely on several externally provisioned components (dependencies) and especially when the artifact itself is a composition of reusable modules, *dependency management* becomes a necessity. Dependency management is defined as an automated means of declaring and retrieving dependencies for use in a component.

For the interpreter project, Apache Maven[3] was chosen as the dependency management tool. This decision was informed by the relative ease with which Maven allows the user to manage, aggregate and build components. Compared to its newer alternative, Gradle[4],

[1] https://docs.python.org/3/library/typing.html.

[2] https://insights.stackoverflow.com/survey/2018, accessed May 2019.

[3] https://maven.apache.org.

[4] https://gradle.org.

Maven also provides helpful structuring conventions. The latter could be considered restrictive when flexibility is a necessity but in the context of this thesis, they serve more as an assistive measure which speeds up development. An added benefit is the wide array of build plugins available for Maven.

4.3.3 **Parser Generator: ANTLR**

The TDL^{TP} interpreter needs to be able to parse both TDL^{TP} expressions and UPPAAL project XML files. The latter contain embedded code snippets in UPPAAL's system definition language. Both TDL^{TP} and the UPPAAL system definition language are examples of domain-specific languages (DSLs) – languages that only have a limited domain of applicability but allow for more succinct or expressive representation within the context of said domain.

The decision here is whether to implement parsers for these DSLs from the ground up, or to use some existing tool capable of generating the required parsers – a *parser generator*. Since efficiency is not a concern in the context of this thesis, there is little reason to undertake the effort of constructing bespoke DSL parsers. Time saved on account of this can be spent on refinement of higher-level components which depend on the parsers.

ANTLR (ANother Tool for Language Recognition)[1] is the generator chosen for implementing the parser components described in Sections 4.4.3 and 4.4.4. ANTLR is a Java library which can easily be incorporated into a Maven build process. Additionally, the parsers it generates represent parse results in the form of parse trees[2] composed of Java objects. This means that once a parser has been generated, dependent components can ignore the parsing logic and remain in the object-oriented domain of Java.

4.3.4 **Code Generation: StringTemplate**

The output of the TDL^{TP} interpreter is a UPPAAL project file. As mentioned in the section above, this file may contain embedded code written in UPPAAL's system definition language. Since the UPPAAL parser component, described in Section 4.4.6, needs to support both parsing and serialization of such embedded code, some technological

[1] https://www.antlr.org.

[2] Parse trees are introduced in section 2.4.

solution is needed to support *code generation* – conversion from an object structure to a code string according to some predetermined syntax.

In the realm of web development, a common approach to transforming data models into HTML code is the employment of a *templating engine*. A templating engine accepts as input an object structure and a fixed template. Then, according to the rules encoded in the template, the engine transforms the object structure into an output string which can be processed further. It is self-evident how a templating engine could be used to generate code in UPPAAL's system definition language.

Of the various templating engines available on the market, StringTemplate[1] proved to be an expedient choice. Due to the strict model-view separation enforced by this templating engine, the generator templates written as part of this thesis are devoid of complex logic and are thus relatively easy to both understand, modify, and reuse.

4.3.5 **XML Processing: JAXB**

As both the output and part of the input for the TDL^{TP} interpreter is encoded in XML, a means for parsing (unmarshalling) and serializing (marshalling) XML files is needed. JAXB (Java Architecture for XML Binding)[2] is a Java framework specialized to this end. Based purely on the programming language of choice (as explained in Section 4.3.1), JAXB was chosen as the XML marshalling tool for the UPPAAL XML parser component outlined in Section 4.4.5.

It should be noted that as of Java 11, JAXB is no longer part of the Java standard edition platform (Java SE)[3]. However, at the time of writing, earlier versions of Java with JAXB included are still available. Additionally, the JAXB project is currently continued under the Jakarta EE[4] effort.

[1] https://www.stringtemplate.org.

[2] https://github.com/javaee/jaxb-v2.

[3] http://openjdk.java.net/jeps/320.

[4] https://jakarta.ee.

4.3.6 **Command-Line Option Parser: args4j**

The top-level component of the TDL^{TP} interpreter is a command-line interface (CLI). The implementation of this component is outlined in Section 4.4.9 and a user guide is provided in Appendix 9.

The TDL^{TP} CLI is a Java application. The entry point for a Java application is its `main` method. If the application is executed from the command-line, any options appended to the initializing call are forwarded to the `main` method in the from of an array of strings. This is the way the CLI component shall receive input, i.e. options, from users.

Processing string arrays is not a particularly difficult task, but since specialized libraries exist for transforming Java `main` option arrays into more useful object representations, there is no reason to undertake the additional burden of writing a bespoke option parser. To reduce time spent on implementation, it was decided that a supplementary Java option parser was needed.

The parser of choice is args4j[1]. Using args4j, the set of possible options made available for the user can be defined as a simple Java class whose fields are decorated with annotations. Each field maps to an option and a wide variety of field types is supported. Using a single utility class provided by the library, the array of strings provided to the `main` method can be transformed into an instance of this option-encapsulating class. The simplicity and customizability inherent in this approach are the reasons args4j was chosen.

4.4 Component Overviews

This section provides implementation overviews for the components listed in Table 1 (Section 4.2). Given that this thesis is not intended to be a highly detailed technical document, the presentations in Sections 4.4.1 – 4.4.9 are kept relatively succinct.

4.4.1 **TDL^{TP} Expression Object Model**

The ANTLR-generated parser described in Section 4.4.3 contains classes for representing TDL^{TP} expressions as parse trees. While it would be possible to exploit these classes

[1] https://github.com/kohsuke/args4j.

directly in higher-level components, there are some limitations. The instantiation logic for ANTLR-generated parse tree classes is tightly coupled to ANTLR's logic. Put simply, it is cumbersome to create new object instances using classes generated by ANTLR – they are meant to be internally instantiated during the parsing process.

To overcome the issues described above, it was decided to apply a layer of abstraction over the ANTLR parser in the form of the TDL^{TP} parser façade (described in Section 4.4.5). The latter encapsulates and provides a simplified API for the ANTLR-based TDL^{TP} parser, converting the latter's parse results into ASTs which do not depend on ANTLR. The component discussed in this section – the TDL^{TP} expression object model – contains the classes used in these independent ASTs.

Due to the nature of ASTs, the object model discussed here naturally manifests tree properties such as parent-child relationships. Such relationships can be modeled by a recursively defined class with reference variables pointing to a single parent instance and zero or more child instances. Figure 17 presents an informal class diagram for the abstract tree-based class structure at the center of the object model for TDL^{TP} expressions.

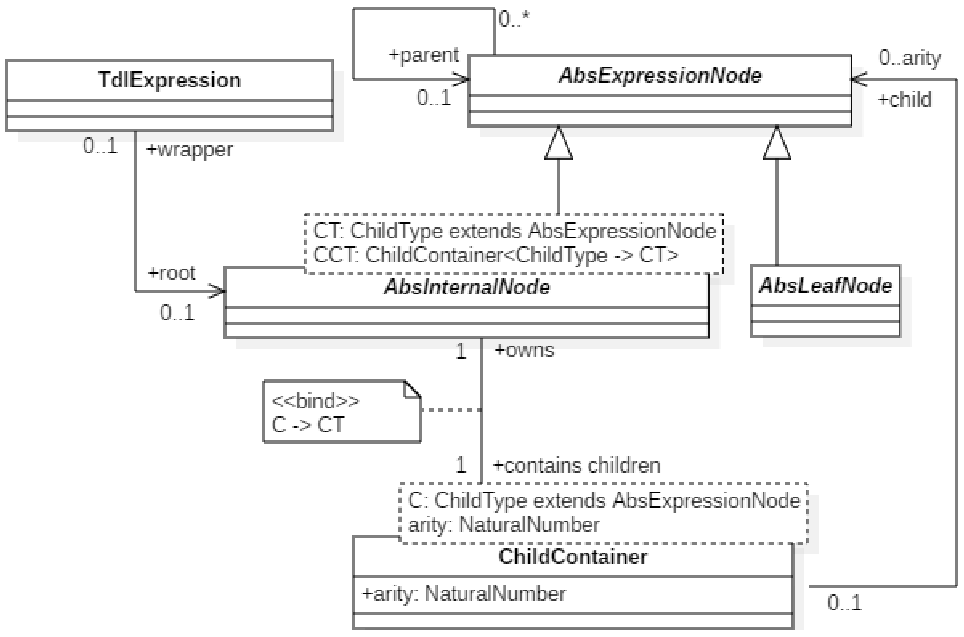


Figure 17. Class diagram for the base classes in the TDL^{TP} expression model.

As illustrated in Figure 17, the AST representation of a TDL^{TP} expression instance is wrapped in an object (of type `TdlExpression`) that points to a root `AbsInternalNode` instance (unless the AST is empty). `AbsInternalNode` is one of two top level concretizations[1] of the base AbsExpressionNode type with the other being AbsLeafNode. AbsInternalNode and AbsLeafNode respectively model internal expression tree nodes (AST nodes that can only occur with child nodes) and leaf nodes.

The abstract node classes mentioned above are concretized further into classes that represent distinct operator and leaf nodes present in the BNF grammar for the TDL^{TP} language. For example, logical connectives and trapset expressions are modeled as descendant classes of AbsInternalNode. To reduce the scope of this discussion, the aforementioned concretizations are not individually introduced here. Instead, we will conclude this section by listing some more salient implementation details for the object model.

Child containers. When inspecting the class diagram presented in Figure 17, the obvious question to ask is: "Why are child nodes of an AbsInternalNode instance encapsulated in a ChildContainer instance instead of being included as aggregated members of the AbsInternalNode class?"

Delegation of child node containment to ChildContainer was required to enforce interface consistency with respect to operator arity. The *arity* of an AbsInternalNode inheritor is defined as the number of child nodes the represented operator node is expected to have. To provide a simple interface for dependent components, one would naturally encode the arity of a concrete AbsInternalNode inheritor in its class contract. For example, a class representing the binary disjunction operator would have methods setLeftChild and setRightChild instead of an abstruse setChild method that accepts an ordinal argument. This reduces cognitive overhead when developing dependent components.

Since the AbsInternalNode class is extended further with abstract classes that generalize logical operators, trapset operators, and trapset quantifiers, encoding the arity contract

[1] These two classes are still abstract, so the term 'concretization' here refers to the fact that they are extensions of the abstract AbsExpressionNode class.

into the AbsInternalNode class causes the class structure to become too large[1]. The solution is to encapsulate the arity contract into a generic container class (ChildContainer) whose concrete implementations (UnaryChildContainer, BinaryChildContainer) can subsequently be used throughout the class layout of the object model via Java's parameterized typing feature.

Subtree hashing. As evident from Sections 5 and 6 of [1], TDL^{TP} expression nodes are subject to normalization and reduction prior to the construction of the UPPAAL test model. This is explained further in Chapter 5, where an algorithm is outlined for these two operations. Here we simply note that normalization requires the ability to replace the child subtrees of an AbsInternalNode instance.

The mechanism which facilitates child subtree replacement is made available for each AbsInternalNode concrete inheritor class via the replaceChildNode method provided by the encapsulated ChildContainer object. This method accepts two AbsExpressionNode instances as arguments and replaces the child subtree that matches the first argument with the subtree represented by the second argument. The term *matching* here can refer to both referential identity[2] and object equivalence[3] between two AbsExpressionNode instances.

The path of least resistance would lead one to simply rely on referential identity in the replaceChildNode method. However, considering possible future applications, it is better to include object equivalence, which is an extension of referential identity. For this reason, AbsExpressionNode classes in the object model were supplied with facilities for determining equivalence. This is exploited in the replaceChildNode method of

[1] Since we concretize AbsInternalNode into a generalizing class for logical operator nodes (AbsLogicalOperatorNode), if we, for example, define arity extensions AbsBinaryInternalNode and AbsUnaryInternalNode from AbsInternalNode, then we would have to define two abstract AbsLogicalOperatorNode arity classes which inherit from them: AbsBinaryLogicalOperatorNode and AbsUnaryOperatorNode. This also applies for other general operator types such as trapset expression operators. Since Java does not support multiple inheritance, an alternative would be to use interfaces (for example, IBinaryNode, IUnaryNode), but this would result in code duplication.

[2] Referential identity is defined here in terms of object variables. Two object variables are referentially identical if they point to the exact same object in memory. An example of Java code utilizing referential identity is `objectA == objectB`.

[3] Roughly put, object equivalence allows two objects to be considered identical for some purpose if they contain the same data – even if they occupy non-overlapping sections of memory. An example of Java code utilizing object equivalence is `objectA.equals(objectB)`.

ChildContainer. In a nutshell, replaceChildNode iterates the child nodes of the container and compares each child to the node provided as its first argument (using the Java equals method). When a match is found, the replacement is performed. This is depicted in Figure 18.

```
public void replaceChildNode(
  ChildType prevChild, ChildType newChild
) {
  for (int i = 0; i < arity; i++) {
    ChildType childCandidate = getChildNode(i);
    if (prevChild.equals(childCandidate)) {
      setChildNode(i, newChild);
      break;
    }
  }
}
```

Figure 18. Basic child replacement procedure for the TDL^{TP} child container class.

Assertion of subtree equivalence (represented by prevChild.equals(childCandidate) in the figure above) is computationally expensive (all the nodes of the subtrees need to be traversed to ensure equivalence), so hashing is required to speed up the process.

Providing a simple hashing mechanism that itself relies on subtree traversal is not much of an improvement. To save time on traversals, ChildContainer has been equipped with encapsulated hash caching logic. Whenever a ChildContainer instance is modified (a child node instance is attached or detached), the container flips on an internal modification semaphore. The next time a hash is requested from the ChildContainer, if the modification semaphore is set, it will recompute the hash of the subtree it represents and cache the result internally; otherwise the container will simply return the previously cached hash. The benefit here is that repeated hash calculations for an expression subtree that has not mutated are avoided. This hashing feature is subsequently exploited in the improved replaceChildNode method depicted in Figure 19.

```
public void replaceChildNode(
 ChildType prevChild, ChildType newChild
) {
  for (int i = 0; i < arity; i++) {
    ChildType childCandidate = getChildNode(i);
    if (childCandidate == null
        || (prevChild.hashCode() != childCandidate.hashCode()))
      continue;

    if (prevChild.equals(childCandidate)) {
      setChildNode(i, newChild);
      break;
    }
  }
}
```

Figure 19. Better child replacement procedure for the TDL^{TP} child container class.

Facilities for subtree inspection. Since TDL^{TP} expression tree objects are expected to be inspected via traversal for a multitude of reasons, and the number of classes present in the corresponding object model is relatively large, some means for generalizing traversal for these objects was needed. To achieve this, the Visitor design pattern introduced in [13, p. 331 – 344] was used throughout the class structure in question. This design pattern supports the implementation of new operations on TDL^{TP} expression trees without changing the classes involved.

4.4.2 **UPPAAL Object Model**

The TDL^{TP} interpreter needs to be able to parse and generate UPPAAL XML files. An individual file defines the structure of a UPPAAL model. The structural elements (e.g. template declarations, transition labels) of such a model may contain embedded code in UPPAAL's system definition language. The component discussed in this section defines classes for representing both the structural and language features of a UPPAAL model.

The justification for providing an object model for UPPAAL's system definition language (separate from the one generated by ANTLR as described in Section 4.4.4), is analogous to the reasoning presented in Section 4.4.1 for the TDL^{TP} expression model. Because of the similarities between the implementations of the two models and owing to the sheer size of the model implemented for the system definition language[1], a description of the

[1] More than 100 classes at the time of writing.

language part of the UPPAAL object model is omitted. The rest of this section is devoted to the classes used for representing the structural aspects of a UPPAAL model.

Figure 20 informally depicts the core classes for the UPPAAL structural model implemented in the component.

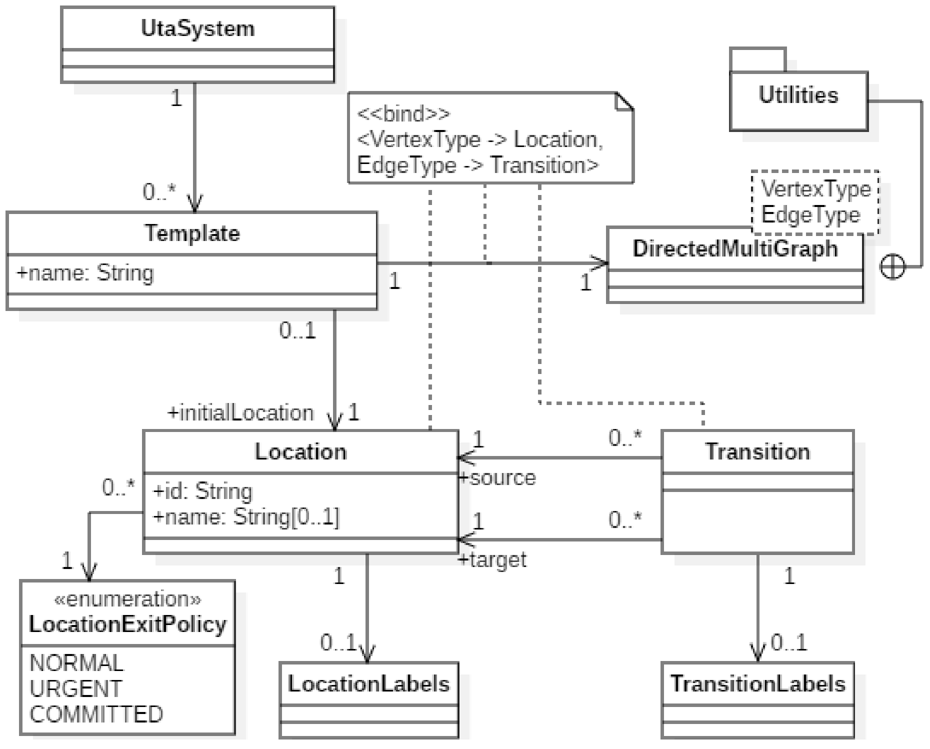


Figure 20. Class diagram for structural classes in the UPPAAL object model.

Notable omissions in Figure 20 include linguistic and purely graphical elements such as nails, colors, and coordinates – these UPPAAL features are of course facilitated by classes in the model, but there is little reason to dissect all of them in this section. Additionally, the two label classes depicted in the figure (`LocationLabels` and `TransitionLabels`) are essentially containers for UPPAAL system definition language ASTs. Due to the simplicity of these container classes, they are also excluded from the discussion that follows.

The object representation of the structure of a UPPAAL model is rooted in the `UtaSystem` class. Objects instantiated from this class can reference zero or more instances of the `Template` class. A template represents the prototype for a UPPAAL timed automaton. As such, it must contain some substructure for encoding `Locations` and the `Transitions` between them.

The abovementioned substructure was implemented via the generic `DirectedMultiGraph` class. As evident from the name of the employed class, the relationship between transitions and locations in a timed automaton can be modeled as a directed multigraph. A *directed multigraph* is a graph in which multiple directed edges are permitted between a pair of vertices. In this case the vertices are locations and the edges are transitions. The `DirectedMultiGraph` class facilitates the definition of directed multigraphs for arbitrary classes based on object equivalence. It is defined in a shared utility module that will not be detailed in this thesis in the interest of brevity.

4.4.3 **TDL^{TP} Grammar Implementation**

As mentioned in Section 4.3.3, the approach taken for implementing language parsers in this thesis was to use the ANTLR parser generator. The component this section is devoted to – the TDL^{TP} grammar implementation – houses an ANTLR-generated parser for TDL^{TP} together with the static specification file used as input for the generator.

Parsers generated by ANTLR include object models for representing language inputs as parse trees. Such object models are tightly coupled to ANTLR's general parsing logic. As this is also the case for the ANTLR TDL^{TP} parser, a façade component was written on top of the base TDL^{TP} grammar implementation described here (Section 4.4.5). This façade component utilizes the AST object model for TDL^{TP} expressions introduced in Section 4.4.1.

In addition to the base TDL^{TP} parser, the component at hand also contains the implementation of a code generator for the language. This generator relies on the StringTemplate engine introduced in Section 4.3.4. The engine, a small collection of transformer classes, and a single template file was needed to compose the generator. The reason it was included as part of the TDL^{TP} grammar component rather than the TDL^{TP} parser façade which higher-level dependent modules are expected to use, is that this avoids having information on the grammar of TDL^{TP} duplicated across multiple components (given that the generator template and the ANTLR parser specification file essentially present the same information).

Regarding TDL^{TP}'s grammar, there are a few key differences between the grammar supported by the generated parser and the grammar presented in Figure 7. These are listed and justified in Table 2.

Table 2. Differences between TDL^{TP} BNF and ANTLR grammars.

Difference	**Justification**
Negation operation is recognized as '~' instead of 'not'.	This adjustment allows for more compact presentation of expressions. Additionally, most of the other operators in the grammar are represented using symbols rather than words.
Disjunction operation is recognized as '|' instead of 'or'.	Similar to the justification for using a different symbol for negation.
The condition part of a conditional repetition subexpression is expected to be wrapped in square brackets. For example: '`#[>10]A(TS1)`' is valid instead of '`#>10A(TS1)`'.	This is the way conditions are represented for the time-bounded leads to operator. The adjustment makes the resultant grammar more consistent.
Trapset quantifiers 'A' and 'E' can be presented in lower-case.	User convenience.
Trapset identifiers are still expected to have the 'TS'-prefix, but letter case is immaterial, so that 'ts1' is also recognized as a trapset identifier.	User convenience.

The adjustments enumerated in Table 2 were applied primarily in order to enforce consistency in the grammar supported by the interpreter. When the grammar for a language is consistent, users will have an easier time memorizing its syntax.

4.4.4 **UPPAAL System Language Grammar Implementation**

As mentioned in Section 4.4.2, UPPAAL XML contains structural and linguistic elements. The linguistic elements are snippets of code written in UPPAAL's system definition language. The TDL^{TP} interpreter needs to be able to parse and generate code in this language. The component under consideration in this section provides the basic facilities that support these requirements.

As is the case for the TDL^{TP} grammar implementation described in Section 4.4.3, the UPPAAL grammar implementation houses an ANTLR-generated parser and a StringTemplate-reliant code generator. There are no significant differences between the two components in terms of general implementation approach.

Of note is the fact that the UPPAAL ANTLR parser discards source code comments[1]. As these comments are not removed from the input SUT model for the TDL^{TP} interpreter, there was no reason to exert additional effort in order to retain them in the corresponding test model produced by the interpreter.

The question could be raised as to why the official UPPAAL language parser – libutap[2] – wasn't used in the interpreter. The answer is that libutap is written in C++, and while there are ways to use C++ libraries in Java, it was simply more expedient to implement a custom parser with a more limited domain of application.

Additionally, the official site for the libutap parser lists an outdated version[3] of the artifact at the time of writing. A newer version was uploaded to a public repository[4] after the implementation effort had begun. The timing of the upload was unfortunate – the custom UPPAAL parser discussed here had already been implemented at the time of upload. Because of the modular structure of the TDL^{TP} interpreter, it is not outside the realm of possibility for future iterations of the artifact to rely on libutap.

4.4.5 **TDL^{TP} Parser**

ANTLR-generated parsers contain a class structure for representing parse results as trees. These structures are tightly coupled with ANTLR runtime libraries and parsing logic. This is at odds with the component-oriented development approach chosen for this thesis (Section 4.1). If the chosen parser solution is to be replaced at some point in time, and the ANTLR-generated classes are in use throughout the component structure that comprises the TDL^{TP} interpreter, a lot more work needs to be done in order to facilitate such a replacement.

For the reasons outlined above, a façade was implemented on top of the parser component described in Section 4.4.3 and the object model described in Section 4.4.1. This façade is

[1] Of the form '`//single line comment`' or '`/* multi-line comment */`'.

[2] http://people.cs.aau.dk/~adavid/utap.

[3] Released 2007 while the newest version of UPPAAL was released in 2014.

[4] https://github.com/mikucionisaau/utap.

called the TDL^{TP} parser because higher-level components will use it for parsing TDL^{TP} expressions instead employing the ANTLR parser implementation directly.

The main objective of the TDL^{TP} parser is to accept a TDL^{TP} expression as a string or stream, pass this input to the TDL^{TP} grammar implementation, and convert the resultant ANTLR parse tree into an AST built from instances of the classes introduced in Section 4.4.1. Because analogous logic was needed for parsing UPPAAL's system definition language, a generic ANTLR façade component was extracted and utilized for both the TDL^{TP} parser under consideration here and the UPPAAL parser described in the following section.

4.4.6 **UPPAAL Parser**

The UPPAAL parser component contains a façade for the UPPAAL grammar implementation detailed in Section 4.4.4. In this respect the UPPAAL parser's goal is similar to that of the TDL^{TP} parser – abstract away ANTLR-related implementation details. The main difference between the two components is that the UPPAAL parser façade needs to support deserialization and serialization between UPPAAL XML and the UPPAAL structural model outlined in Section 4.4.2. As the ANTLR-encapsulating logic in the UPPAAL parser does not differ in any significant manner from the logic implemented for the TDL^{TP} expression language, we focus the discussion in this section on processing UPPAAL XML[1].

A high-level specification of the structure of a UPPAAL XML file was available at the time of writing in a publicly accessible definition file[2]. As the XML processor of choice in this thesis is JAXB (Section 4.3.5), which is capable of generating Java classes from XML schema definition[3] (XSD) files, the previously mentioned definition file had to be manually converted into an XSD file. Then, using JAXB's xjc[4] tool via Maven, this XSD

[1] As the UPPAAL parser component both deserializes XML into object structures and serializes object structures into XML, it could be argued that the name of the component ('parser') is inaccurate. However, it was deemed simpler to follow the same naming conventions between the TDL^{TP} parser and the UPPAAL serializer/deserializer.

[2] http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd.

[3] https://www.w3.org/TR/xmlschema11-1.

[4] https://docs.oracle.com/javase/tutorial/jaxb/intro/custom.html.

file was used produce a set of classes whose instances the JAXB framework can both serialize into XML and deserialize from XML.

To conceal irrelevant JAXB-related details from higher-level components, the UPPAAL parser converts object structures instantiated from xjc-generated classes via the JAXB parser into the UPPAAL object model described in Section 4.4.2. When a UPPAAL object model is to be serialized into XML, the converse of the previously described operation is executed – general-purpose UPPAAL objects are converted into JAXB-specific objects and subsequently serialized into XML.

Special steps were taken to keep code parsing/generation relatively isolated from structural conversion in the component. When a UPPAAL XML file is parsed, JAXB is used to generate the corresponding intermediate object structure where UPPAAL system definition language snippets are nested in simple string fields. During the conversion process from the JAXB-specific UPPAAL object structure to the general-purpose UPPAAL object structure, when a code snippet is encountered, a parse operation is added to an operation queue. An example is depicted in Figure 21.

```
TransitionLabels labels = new TransitionLabels();
...
AssignmentsLabel assignmentsLabel;
label = (assignmentsLabel = new AssignmentsLabel());

getParseQueue().enqueue(
  transitionLabelXml.getValue(),
  getParserFactory().assignmentsParser(),
  assignmentsLabel::setContent
);

labels.setAssignmentsLabel(assignmentsLabel);
...
transition.setLabels(labels);
```

Figure 21. Example of parse deferral during UPPAAL structural conversion.

As parse operations are deferred when converting from the JAXB UPPAAL model to the object model intended for use in higher-level components, the conversion logic is kept relatively clean – the details of language parsing can be ignored to an extent when converting between object models. Conversely, parsing logic can remain isolated from conversion logic.

A high-level view of UPPAAL parsing logic is depicted in Figure 22.

```
public UtaSystem parse(InputStream in)
    throws MarshallingException,
       InvalidSystemStructureException,
       EmbeddedCodeSyntaxException {
  UtaNode utaNode = unmarshal(in);
  validateStructure(utaNode);

  ParseQueue parseQueue = new ParseQueue();
  UtaSystem utaSystem = convert(utaNode, parseQueue);
  parseQueue.executeRemaining();

  return utaSystem;
}
```

Figure 22. High-level view of parsing logic used in the UPPAAL parser.

The approach to deserializing UPPAAL XML outlined above is also applicable for the serialization process. The operations involved – conversion between object structures, execution of deferred code processing operations – are simply reordered.

4.4.7 **Scenario Composer**

At the heart of the TDL^{TP} interpreter is the scenario composer. This component accepts as input:

a. a TDL^{TP} expression represented by an AST consisting of instances of the classes defined in the TDL^{TP} object model (Section 4.4.1), and
b. a UPPAAL model represented by an object structure consisting of instances of the classes defined in the UPPAAL object model (Section 4.4.2).

Having received its inputs, the component then applies the mapping rules described in [1, p. 6 – 11] to produce a resultant test model in the form of an object structure. This structure can then be serialized into UPPAAL XML. It should be noted that the component has been kept isolated from parsing and generation logic. This enforces simplicity and reuse.

Due to the significance of the scenario composer, Chapter 5 is devoted entirely to its inner workings.

4.4.8 **User Interface Core**

The TDL^{TP} component structure is relatively difficult to use unless supplemented with a user interface (UI). To support the production of such interfaces, the UI core component was implemented. This component binds the functionalities provided by the UPPAAL parser (Section 4.4.6), the TDL^{TP} parser (Section 4.4.5), and the scenario composer (Section 4.4.7) into a single class called the `TdlInterpreterUI`.

`TdlInterpreterUI` is meant to be used as the scaffolding for graphical and command-line UIs for the interpreter. It provides means for reporting errors and hooking into the state of the interpretation via basic listener interfaces. Because of the component's relatively simple structure, we exclude the presentation of its internals here.

4.4.9 **Command-Line Interface**

The only user interface implemented for the TDL^{TP} interpreter is a simple Java-based command-line interface (CLI). This component extends the UI core introduced in Section 4.4.8. Its implementation was simplified by the exploitation of the args4j library for Java (discussed in Section 4.3.6).

A basic user guide for the TDL^{TP} CLI, including a description of the inputs it can accept, is provided in Appendix 9. Because of the relative triviality of the CLI component with respect to the complexity of the other modules introduced in this section, discussion of its implementation is omitted.

5 Scenario Composition

The scenario composer module introduced in Section 4.4.7 contains the core logic for applying a TDL^{TP} expression to a UPPAAL model. The composer accepts these two inputs in the form of object structures instantiated from classes defined in the TDL^{TP} expression object model (Section 4.4.1) and the UPPAAL object model (Section 4.4.2). The output of the composer is an object structure which represents a test model. This chapter describes the logic embedded in the implementation of the composer. The theory behind the implementation is introduced in Section 2.3 and is based on the information provided in [1].

Section 5.1 presents an overview of the composition procedure developed for the scenario composer. Characteristics of the most significant steps in the procedure – trapset evaluation, normalization, reduction, and construction of a recognizer model – are discussed in Sections 5.2 – 5.5.

5.1 Overview

Section 8 in [1, p. 10 – 11] lists the steps needed to construct a test model based on a TDL^{TP} expression and a UPPAAL SUT model as follows:

1. The test purpose is specified as a TDL^{TP} expression.
2. Trapsets[1] TS_1 – TS_n which occur in the expression are defined in the UPPAAL SUT model via assignment labels (traps) attached to automata transitions.
3. The AST of the TDL^{TP} expression is traversed. Each of its operator sub-nodes is mapped to an independent recognizer template using Mappings M4 – M10 from Section 5.2 of [1].

[1] Trapsets are introduced in Section 2.3.3.

4. The labeling of the SUT model is simplified by applying the rules for trapset expressions described in Mappings M1 – M3 from Section 5.1 of [1]. Recognizer templates are linked together via broadcast signaling channels.
5. The root recognizer is linked to the stopwatch automaton.

The summary presented above omits normalization and reduction. These steps are described in Sections 5.3 and 5.4, respectively. Additionally, steps 1 and 2 are outside the scope of this thesis in that they are performed by the user.

In order to simplify the discussion in this chapter, we concretize the general logic encoded in steps 3 – 5 into a sequence of operations which includes normalization and reduction. These operations are implemented in the interpreter produced as the result of this thesis.

Given an input TDL^{TP} expression ε and a UPPAAL SUT model M, a resultant test model M^{TEST} is derived as follows:

(1) **Base trapset extraction.** Leaf trapset nodes present in ε's AST are collected into the set S^{TS} and mapped to transitions in M.
(2) **Trapset expression evaluation.** Trapset expression nodes that occur as parents of the nodes in S^{TS} are collected into the dictionary structure m^{EVAL} which maps trapset expressions to object representations of the corresponding base trapsets[1]. These representations are constructed according to the information in S^{TS}, the entire collection of transitions in M, and the rules for trapset operators defined in Mappings M1 – M3 of [1].
(3) **Trapset quantifier evaluation.** Trapset quantifiers in ε are iterated. Each trapset quantifier maps to a trapset expression contained in the keys of m^{EVAL}. If the trapset expression child of a quantifier node maps to an empty base trapset, the quantifier can be replaced with a Boolean literal in ε's AST[2].

[1] The notion of a base trapset is defined in Section 2.3.4.

[2] This is not fully explained in [1]. More information is provided in Section 5.2.

(4) **Normalization.** ε's AST is traversed top-down starting from its root in order to eliminate top-level negations[1] and operators that do not have a recognizer mapping[2].

(5) **Reduction.** After normalization, ε is traversed bottom-up starting from the injected Boolean literal leaves, if they exist. An individual literal is pulled up the tree by application of the reduction rules defined in equivalence listing (14) from [1]. This is done to reduce the size of ε's AST. Steps are taken to avoid traversing subtrees that become detached during the reduction process.

(6) **Construction of a recognizer model and injection of trapsets into the SUT model.** After reduction, the possibly modified version of ε's AST is traversed top-down yet again. This time each remaining logical operator is mapped to a UPPAAL recognizer template and a corresponding template instantiation[3]. These artifacts are inserted into an initially empty[4] UPPAAL model M^{WRAP}, i.e. the *recognizer model*. Additionally, each trapset expression that is still referred to in ε's AST is injected into M as a base trapset according to previously stored information in m^{EVAL}. This results in a modified model, M'. The test stopwatch, trapset label collection, and recognizers involved here are implicitly connected via channel references by the end of the step.

(7) **Test model composition.** M' and the recognizer model M^{WRAP} are merged into the test model M^{TEST}. The latter is therefore an extension of M with a modified configuration of trapset variables and an injected recognizer structure.

In steps (1) – (7) above, four core sub-procedures can be distinguished: **trapset evaluation**, **normalization**, **reduction**, and **construction of the recognizer model**. These are discussed in Sections 5.2 – 5.5, respectively. In the remainder of this section we provide a sequence of diagrams (Figure 23 – Figure 28) which illustrate steps (1) – (7).

[1] Negation is only supported for trapset quantifiers.

[2] Recognizers are introduced in Section 2.3.1. The relation between TDL^{TP} operators and recognizers is a partial function. Some TDL^{TP} operators do not map to a recognizer. This is permissible since the unmapped operators can be represented by equivalent combinations of other operators which do have recognizer mappings.

[3] UPPAAL templates and template instantiations are defined in Section 2.2.3.

[4] M^{WRAP} is empty only in the sense that it does not initially contain any operator recognizer templates or instantiations when constructed. It will, however, contain the stopwatch automaton template and some globally declared variables.

Figure 23 presents an example TDL^{TP} expression (`E(TS1; TS2) & A(!TS3)) | E(TS1)`) using the version of TDL^{TP} syntax accepted by the interpreter[1] and a simple UPPAAL SUT model annotated with traps linking the model to the expression.

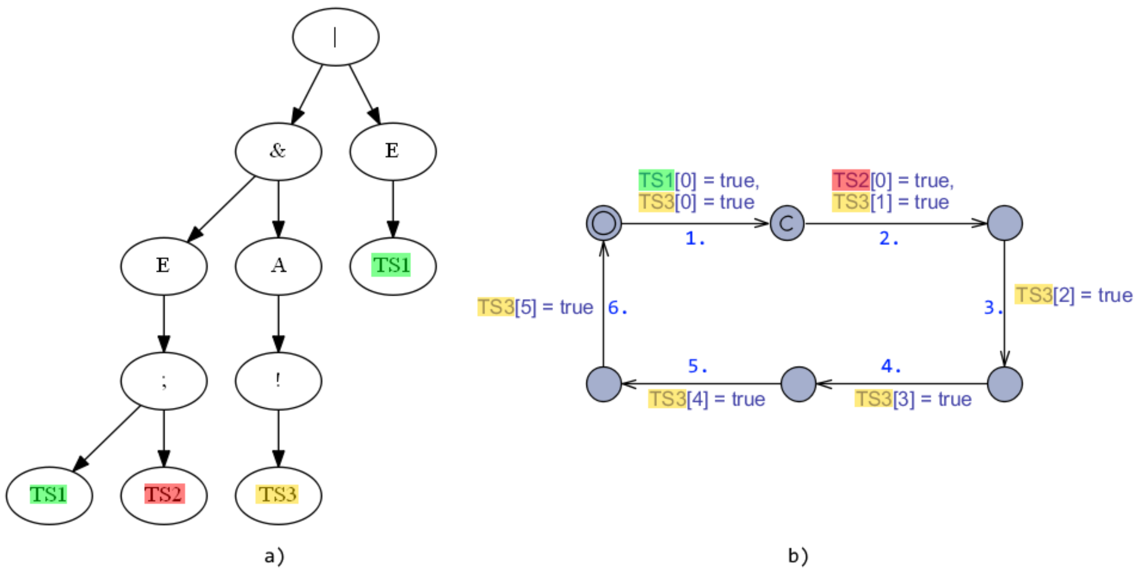
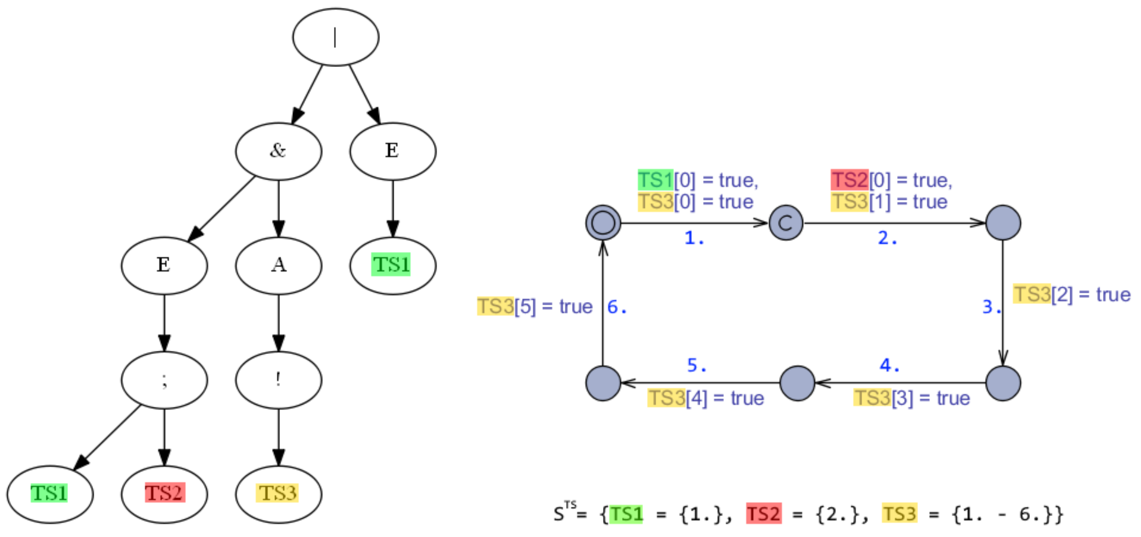


Figure 23. Example TDL^{TP} expression (a) and a model annotated with trapsets (b).

The expression refers to three trapsets: TS_1, TS_2, and TS_3. TS_1 and TS_2 are both mapped to a single transition in the model (1 and 2 respectively), while TS_3 is mapped to every transition (1 – 6).

[1] Differences between the BNF for TDL^{TP} provided in Section 3 of [1] and the grammar the TDL^{TP} interpreter accepts are listed in Table 2, Section 4.4.3.

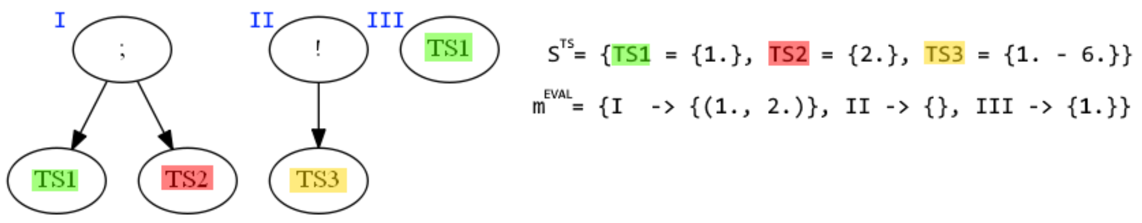
The first step of the composition procedure involves collecting trapset information from the expression and the SUT model into the set S^{TS}. This is depicted in Figure 24.



```
S^TS = {TS1 = {1.}, TS2 = {2.}, TS3 = {1. - 6.}}
```

Figure 24. First step of composition: base trapset extraction.

The second step of the composition procedure involves the evaluation and storage of trapset expressions in the m^{EVAL} map. This is illustrated in Figure 25.



```
S^TS = {TS1 = {1.}, TS2 = {2.}, TS3 = {1. - 6.}}
m^EVAL = {I  -> {(1., 2.)}, II -> {}, III -> {1.}}
```

Figure 25. Second step of composition: evaluation of trapset expressions.

The three trapset expressions present in the AST in Figure 23 map to three entries in the m^{EVAL} table as shown in the figure above. They are marked in the latter figure with Roman numerals I – III. Trapset expression II (!TS3) is mapped to the empty set because it is the absolute complement of a trapset which covers the entire model.

The third step of test model construction involves iteration of the trapset quantifier nodes in the AST. Each quantifier is mapped to a trapset expression, which m^{EVAL} correspondingly maps to an evaluation (a base trapset). If the transitively retrieved base trapset for a quantifier is empty (it contains no traps), a reduction rule is applied. This is called trapset quantifier evaluation. A depiction is provided in Figure 26.

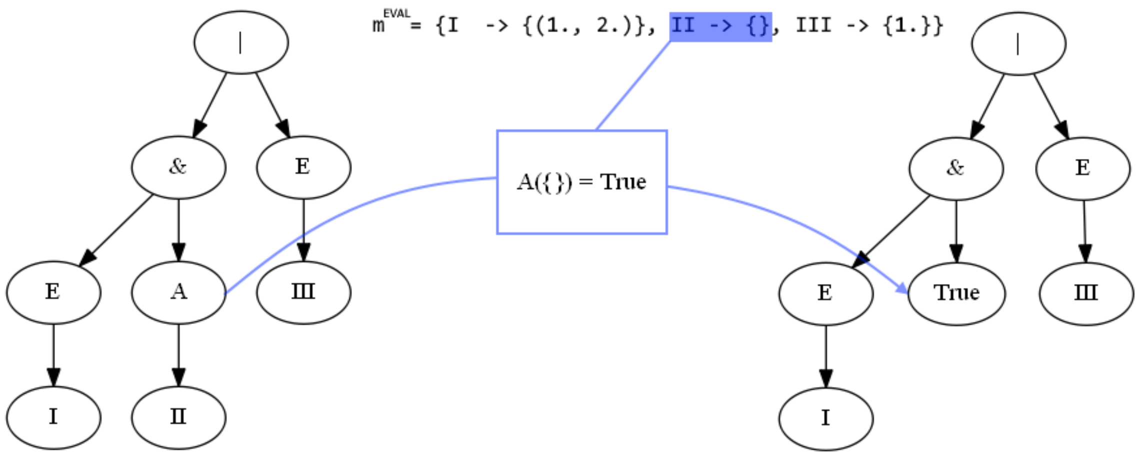


Figure 26. Third step of composition: evaluation of trapset quantifiers.

According to Figure 26, the subexpression `A(!TS3)` was replaced with a Boolean true literal. This is because the absolute complement of trapset TS_3 is empty and in our interpretation of the TDL^{TP} language, universal trapset quantification over an empty trapset maps to a true literal. An explanation is provided in Section 5.2.2.

The fourth step of composition (normalization) involves the application of logical equivalences in order to push negation to the level of trapset quantifiers and eliminate operators that do not have a recognizer mapping. As the example at hand contains neither negation nor recognizer-less operators, a depiction of this step is omitted. We continue the example with the fifth step – reduction of the AST via elimination of Boolean literal nodes. This step is illustrated in Figure 27.

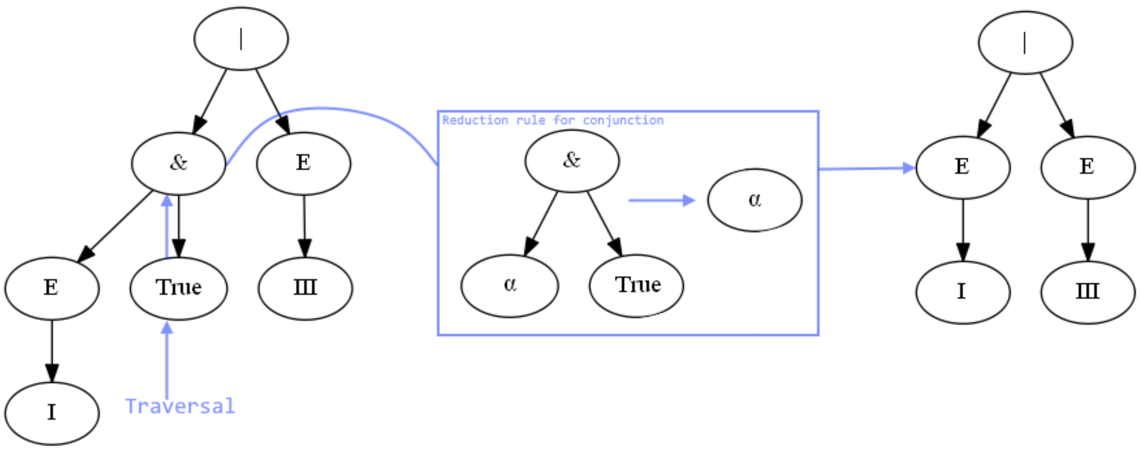


Figure 27. Fifth step of composition: elimination of Boolean literals.

The modified AST produced as the result of reduction is smaller in size and represents the TDL^{TP} expression `E(TS1; TS2) | E(TS1)`. It should be noted that [1] also defines reduction rules that do not involve Boolean literals. These rules were excluded from the implementation of the $\mathrm{TDL}^{\mathrm{TP}}$ interpreter in order to limit the scope of the project. Reduction is explained in Section 5.4.

The sixth step of test model composition involves the generation of a recognizer model $\mathrm{M}^{\mathrm{WRAP}}$, and the application of trapset expressions to M. This is illustrated in Figure 28.

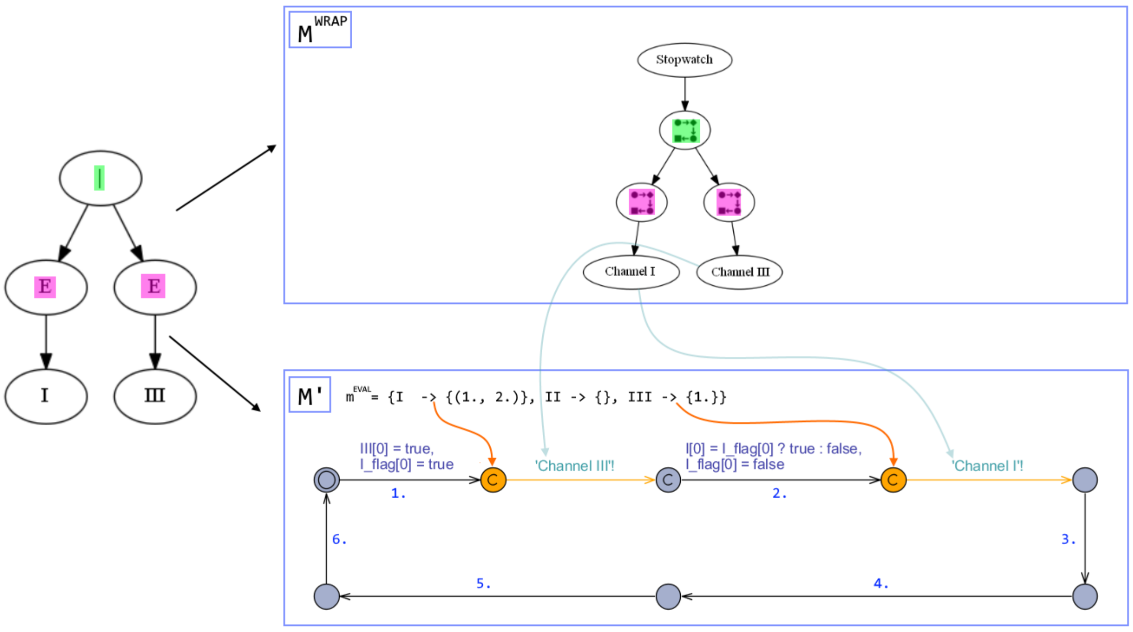


Figure 28. Sixth step of composition: constructing $\mathrm{M}^{\mathrm{WRAP}}$ and M'.

Compared to the original UPPAAL model M in Figure 23, the modified model M' in the figure above does not contain any traps for TS_3. This is because the expression subtree which contained a reference to this trapset was eliminated (Figure 24 – Figure 27). Additionally, the remaining trapset-marked transitions have been extended with auxiliary transitions labeled with channels that link model M to the leaf trapset quantifier recognizers realized in $\mathrm{M}^{\mathrm{WRAP}}$. Step six is explained in Section 5.5.

The last step required to produce a test model is the merging of $\mathrm{M}^{\mathrm{WRAP}}$ and M'. Because the merging process is relatively straightforward (essentially a sequence of setter method calls initiated by `sutModel.merge(recognizerTreeModel)`), a separate figure is not provided. The rest of this chapter is devoted to the most significant steps involved in test model composition – trapset evaluation, normalization, reduction, and the construction of a recognizer model.

5.2 Trapset Evaluation

Trapsets are the means by which the user maps the input SUT model to the input TDL^{TP} expression. The semantics of trapsets is introduced in Section 2.3.3. This section focuses on the segment of the scenario composer that implements base trapset extraction, trapset expression evaluation and trapset quantifier evaluation – collectively referred to as *trapset evaluation*.

Trapset extraction is the process of determining which transitions in the SUT model are mapped to trapsets and trapset expressions. A brief explanation is given in Section 5.2.1. The process corresponds to step (1) and step (2) described in Section 5.1.

Quantifier evaluation corresponds to step (3) in Section 5.1. It involves replacing trapset quantifier nodes in the TDL^{TP} expression's AST with Boolean literals based on a small set of rules. Quantifier evaluation is explained in Section 5.2.2.

5.2.1 **Trapset Extraction**

In this section we describe the process of extracting trapset related data from a SUT model based on a TDL^{TP} expression's AST and the set of transitions in the SUT model. The notion of trapset extraction includes the evaluation of trapset expressions.

At this point it is pertinent to point out a difference between the TDL^{TP} interpreter and the theory outlined in [1]. Namely, a decision was made to allow for the use of simple UPPAAL Boolean variables as trapset labels in the input SUT model instead of the array variables suggested by the presentation in [1]. Essentially this means that when a user wishes to include a transition in trapset TS_1, they should use the variable declaration `bool TS1` and the transition assignment `TS1 = true|<condition>` instead of the variable declaration `bool TS1[<array size>]` and the assignment `TS1[<array index>] = true|<condition>`.

This decision was made for two reasons:

1. It is more convenient for the user to annotate their input model with simple Boolean variable assignments instead of array variable assignments. If the user is required to use arrays, they also need to keep track of array indices and size restrictions. This

manual effort is unnecessary as the interpreter does not need the additional information encoded in array indices and would discard it.

2. The algorithm for finding trapset annotated transitions in the UPPAAL model becomes simpler. Only basic Boolean variable declarations[1] and grouped Boolean variable declarations[2] need to be considered.

As a trapset should only be mapped to a specific transition in the SUT model once (according to Section 4.2 in [1]), the trapset labeling formalism accepted by the TDL^{TP} interpreter is equivalent to the array-based input format described in [1].

We present simplified pseudocode for the base trapset extraction algorithm implemented in the interpreter in Figure 29.

```
trapsetNodes ← leaf trapset nodes in TDL^TP expression;
trapsetMap ← empty map;

for globalDeclaration in system:
  if globalDeclaration declares a Boolean variable:
    variableDeclaration ← globalDeclaration
    variableName ← variableDeclaration.name
    if trapsetNodes contains a trapset with name variableName:
      trapsetMap[variableName] ← {
        trapsetNode,
        variableDeclaration,
        traps: []
    }

for template in system:
  for transition in template:
    if transition has no assignment labels:
      NEXT
    for assignment label attached to transition:
      variableName ← assignment.variableName
      if variableName in baseTrapsetMap:
        add (transition, assignment) to
          trapsetMap[variableName].traps
```

Figure 29. Pseudocode for base trapset extraction.

[1] For example: ‘`bool TS1;`’, which declares the Boolean variable `TS1`.

[2] For example: ‘`bool TS1, TS2, TS3, flags[10];`’, which declares the Boolean variables `TS1`, `TS2`, `TS3`, and the Boolean array `flags`.

The most time-consuming part of the algorithm depicted in Figure 29 is the iteration of all transitions in the SUT model in the second loop. As the number of assignment labels per transition and the number of templates per model is generally smaller than the total number of transitions, the time complexity of the algorithm is roughly O(*e*) where *e* is the total number of transitions in the model. Iteration of all transitions is unavoidable – the interpreter needs to ensure that trapsets specified by the user are collected appropriately.

Having determined which transitions each trapset in the TDL^{TP} expression maps to, the composer can compute similar data for trapset expression nodes – this is what is meant by *evaluation of trapset expressions*. The evaluation process is performed according to Mappings M1 – M3 in [1, p. 6 – 7]. We present the pseudocode for evaluating an instance of the absolute complement operator in Figure 30.

```
trapset ← operand of absolute complement
absoluteComplement ← empty map

for template in system:
  for transition in template:
    if transition not in trapset:
      add (transition, NIL) to absoluteComplement
    else:
      trap ← trap data for transition in trapset
      if trap is conditional:
        condition ← negation of trap.condition
        add (transition, condition)
          to absoluteComplement
```

Figure 30. Pseudocode for evaluating absolute complement.

In the algorithm depicted above, all the transitions in the system are iterated. If a transition is not mapped to the trapset to which absolute complement was applied, this transition is added to the resultant mapping. Otherwise, if the transition *does* map to a trap in the operand trapset, then the corresponding trap label is checked for conditionality.

A conditional trap in the operand trapset is mapped to a trap in the absolute complement whose condition is negated. Trap conditionality is explained in Section 2.3.3. In the interest of conciseness, we simply mention that the interpreter was built with facilities for supporting it.

Of the three trapset operators, absolute complement has the most time-consuming trapset expression evaluation procedure. Assuming there are *n* distinct absolute complement nodes in the TDL^{TP} expression tree, the time complexity of repeatedly applying the procedure in Figure 30 is roughly O(*ne*). Though there are probably faster alternatives, performance was not a critical requirement for the interpreter.

Pseudocode for the other two trapset operators – linked pairs and relative complement – will not be provided. Logic for their evaluation algorithms is similar to the logic embedded in the absolute complement algorithm.

5.2.2 **Quantifier Evaluation**

As mentioned in the introduction to Section 5.2, quantifier evaluation is the mapping of trapset quantifiers to Boolean literals.

The reduction formulae presented in Section 6 of [1] provide rules for eliminating Boolean literals from a TDL^{TP} expression. However, the article itself does not exactly describe how these literals could come up during the evaluation of an expression[1]. Boolean literals are also not included in the grammar of TDL^{TP}.

It was decided after consultation with the authors of [1] that under certain limited conditions, trapset quantifiers can be replaced with either true or false in a TDL^{TP} AST. The semantics behind these literals is rudimentary in anticipation of future theoretical improvements. *True* maps to a recognizer that immediately returns a success signal to its parent recognizer after activation. *False* maps to a recognizer that never returns a success signal to its parent recognizer – it will immediately reset itself after receiving an activating signal.

[1] On page 8 of [1], the following equivalence is provided: trapset $TS \equiv false$ if $TS = \emptyset$. However, as trapsets are not directly mapped to Boolean values themselves (they are containers for variables which map to Boolean values), the equivalence is not implementable in its present form.

The rules for replacing a trapset quantifier with a Boolean literal are listed in Table 3.

Table 3. Trapset quantifier replacement rules.

Trapset quantifier	**Operand trapset**	**Replacement literal**	**Justification**
Universal (A)	Empty trapset (does not map to transitions in the model).	True	By analogy with $\forall t \in T(P(t))$ where $T = \emptyset$ from predicate logic.
Existential (E)	Empty trapset (does not map to transitions in the model).	False	By analogy with $\exists t \in T(P(t))$ where $T = \emptyset$ from predicate logic.

The algorithm for replacing trapset quantifiers in a TDL^{TP} AST is relatively simple: for each trapset quantifier in the tree, find the corresponding trapset expression and check whether the latter's trap count is equal to zero, then apply one of the rules listed in Table 3.

5.3 Normalization

Normalization is performed after the quantifiers in the TDL^{TP} AST have been checked for immediate evaluability, as described in the section above. Roughly put, normalization is the process of replacing substructures in the AST for which no recognizer mappings exist. The mappings are not needed since the corresponding substructures can be simplified. Normalization corresponds to step (4) from Section 5.1.

Per Section 2.3.5, some logical operators in TDL^{TP} are defined via equivalence. For example, the implication `α => β` can be replaced with the equivalent disjunction, `~α | β` (as is the case in classical logic). Since no recognizer mappings exist for the TDL^{TP} equivalence ('`<=>`') and implication ('`=>`') operators, such replacements are mandatory.

Additionally, TDL^{TP} also provides a version of the negation operator ('`~`'). A recognizer mapping exists for this operator only in the sense that recognizers for trapset quantifiers support a negated mode. When negation is applied to any *other* logical operator in a TDL^{TP} expression, a substitution needs to be performed.

Rules for the removal of negation exist for most operators in TDL^{TP}. For example, the negation of a disjunction can be replaced with a conjunction of the negations of the disjuncts: `~(α | β) ≡ ~α & ~β`. A removal rule generally moves a negation applied to an operator to its child operands. By repeated application of the negation rules provided in Section 6 of [1], top-level negations in the AST can be pushed down towards the leaves until only trapset quantifiers are negated.

Application of the substitution rules described above is what is meant by *normalization* in the context of this thesis. The corresponding algorithm is relatively simple: traverse the AST depth first, replacing the current node whenever a substitution rule is applicable.

The only problem is that there are some operators such as leads to (‘`~>`’) whose negation rule maps to a construct which is not suitable for the recognizer architecture described in TDL^{TP} theory. This was discovered after the implementation effort for the TDL^{TP} interpreter was nearing completion[1]. We list these operators in Table 4.

Table 4. Unimplementable negation rules.

Unimplementable negation rule	**Justification**
Negation of leads to: `~(A ~> B)`.	`~(A ~> B)` maps to a theoretical construct whose direct recognizer implementation needs to wait until the end of a test run and verify that `B` did *not* occur during said run. This is not supported in the recognizer architecture laid out in [1].
Negation of timed bounded leads to: `~(A ~> [* n] B)` where `*` is in `{<, <=, =, >=, >}`.	The justification is identical to the one provided for negation of the leads to operator.

The items in Table 4 are subject to exploration in future work. As they are problems which exist in the theoretical domain of TDL^{TP}, we will not attempt to provide a solution here.

[1] As per Section 4.2, the scenario composer was one of the last components implemented.

5.4 Reduction

Two factors necessitate the existence of reduction rules for TDL^{TP} expressions: the need to reduce the state space of the test model produced by the composer, and the need to eliminate the Boolean literals introduced in Section 5.2.2. The latter factor is subsumed by the former. Application of the reduction rules provided in Section 6 of [1] corresponds to composition step (5) described Section 5.1 and is the topic of this section.

In order to decrease the effort needed to implement the TDL^{TP} interpreter, several reduction rules provided in the theory were not implemented. In fact, only the rules needed to eliminate Boolean literals were encoded in the scenario composer. Future iterations of the interpreter can incorporate more reduction rules after a minimal refactoring effort.

Pseudocode for the literal reduction algorithm employed by the scenario composer is presented in Figure 31.

```
Procedure eliminateLiterals(expression):
  booleanLeaves <- extractBooleanLeaves(expression)
  literalStack <- empty stack

  initialize literalStack with booleanLeaves
  while literalStack isn't empty:
    currentLiteral <- pop first item from literalStack
    if currentLiteral is root of expression:
      EXIT

    if currentLiteral is no longer attached to expression:
      NEXT

    eliminateLiteral(expression, currentLiteral, literalStack)
```

Figure 31. Pseudocode for literal reduction.

The algorithm encodes a bottom-up traversal of the AST starting from its Boolean leaves and moving towards the root. References to leaves that need to be dealt with are kept in a stack. If this stack isn't empty, its first entry is removed for inspection. If the entry is in fact the root of the tree, then this means the entire TDL^{TP} AST has been reduced to a Boolean literal – this is an edge-case that will be reported to the user by the interpreter.

When the reduction algorithm is executed, it may occur that a literal present in `literalStack` becomes detached from the AST. This is depicted in Figure 32.

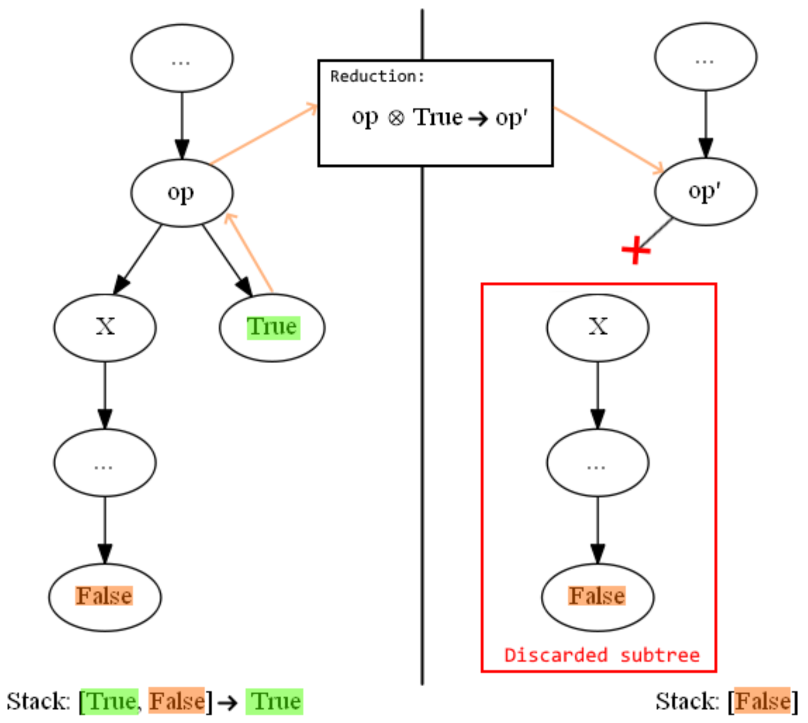


Figure 32. TDL^{TP} AST reduction edge case.

In order to solve the issue, the `eliminateLiterals` procedure in Figure 31 checks whether each literal it tries to remove is still attached. This can be achieved by walking up the parent branch for the literal until either the root is reached, or it is discovered that the literal no longer exists in the TDL^{TP} AST.

`eliminateLiterals` in Figure 31 calls the `eliminateLiteral` procedure for applicable nodes. Simplified pseudocode for the elimination rule for disjunction nested in this procedure is provided in Figure 33.

```
Procedure eliminateLiteral(expression, literal, literalStack):
  parent <- literal.parent
  // ...
  If parent is disjunction:
    If literal is TRUE:
      newLiteral <- TRUE literal
      push newLiteral to literalStack
      replace parentNode in expression with newLiteral
    Else If literal is expression.leftChild:
      replace parent in expression with expression.rightChild
    Else:
      replace parent in expression with expression.leftChild
```

Figure 33. Pseudocode for eliminating the Boolean child node of a disjunction.

Reduction algorithms for other TDL^{TP} operators are omitted here due to the space limitations of the thesis.

5.5 Construction of a Recognizer Model

In step (6) of the composition procedure presented in Section 5.1, the TDL^{TP} AST that remains after normalization and reduction is traversed depth-first and mapped to a recognizer tree[1]. Additionally, trapset expression evaluations are applied to the input SUT model, producing a modified model capable of communication with the tree.

Since the data for trapset evaluations is available from the results collected in step (1) and step (2) (discussed in Section 5.2), application of trapset expression evaluations to the SUT model is fairly straightforward. The traps in each evaluation are iterated and supplementary assignment labels are appended to applicable edges in the model. For each new trap attached to an edge, the edge is split into two with an output channel present on the additional transition (this channel provides the SUT model output which the recognizer tree will consume). The initial trapset annotations provided by the user are stripped from the SUT model because their purpose was simply to define base trapsets for the interpreter.

Due of the simplicity of injecting trapset evaluations into the SUT model, we devote the final section on scenario composition to the construction of a recognizer model. Section 5.5.1 discusses our approach to implementing the recognizer templates included in the model and Section 5.5.2 focuses on the construction algorithm that employs these templates.

5.5.1 **Approach to Implementing Recognizer Templates**

Mappings M4 – M10 in Section 5.2 of [1] describe recognizers for operators in the TDL^{TP} language. Section 7 of [1] describes how these recognizer templates are extended with wrapping constructs in order to support communication both within the recognizer tree and between the tree and the SUT model.

Though [1] presents recognizer template mappings and the communication wrapping mechanism separately, it was decided that the implementation of the scenario composer

[1] Recognizer trees are introduced in Section 2.3.1. We use the terms *recognizer tree* and *recognizer model* interchangeably in this section.

will not include this distinction. Communication facilities were integrated into recognizers.

Two alternate approaches were initially considered for implementing recognizer templates in the scenario composer. Both are listed in Table 5.

Table 5. Recognizer template implementation approaches.

Approach	Advantages	Disadvantages
Implement in Java code using the UPPAAL object model[1].	The scenario composer does not have to know anything about parsing UPPAAL XML – it can remain at its abstraction level. It is easier to implement configurability for the recognizers.	It would take too long to implement all mappings manually. Additionally, it will be difficult to make changes to the templates due to the amount of code involved.
Implement in one or more UPPAAL XML files using the UPPAAL GUI application.	Both the easiest and the most obvious way to implement recognizer templates. GUI facilitates visual 'beautification' of templates.	For the scenario composer to be able to use these XML files, it must be supplied with access to the UPPAAL parser. This is an abstraction leak. Additionally, wrapper classes in Java would still be needed in order to keep code that uses the templates relatively readable. This results in partial duplication between the XML and the Java implementation.

The chosen implementation approach is a compromise between the two alternatives presented in the table above. A supplementary Maven plugin – the UPPAAL pickler – was developed. The pickler accepts a UPPAAL model in XML format as input and

[1] Described in Section 4.4.2.

generates Java factory classes that can be used to instantiate object structures equivalent to the input model[1]. These factories employ classes from the UPPAAL object model.

This approach made it possible to design recognizer automata using UPPAAL's GUI application while remaining at the appropriate level of abstraction in the scenario composer's source code. This is illustrated in Figure 34.

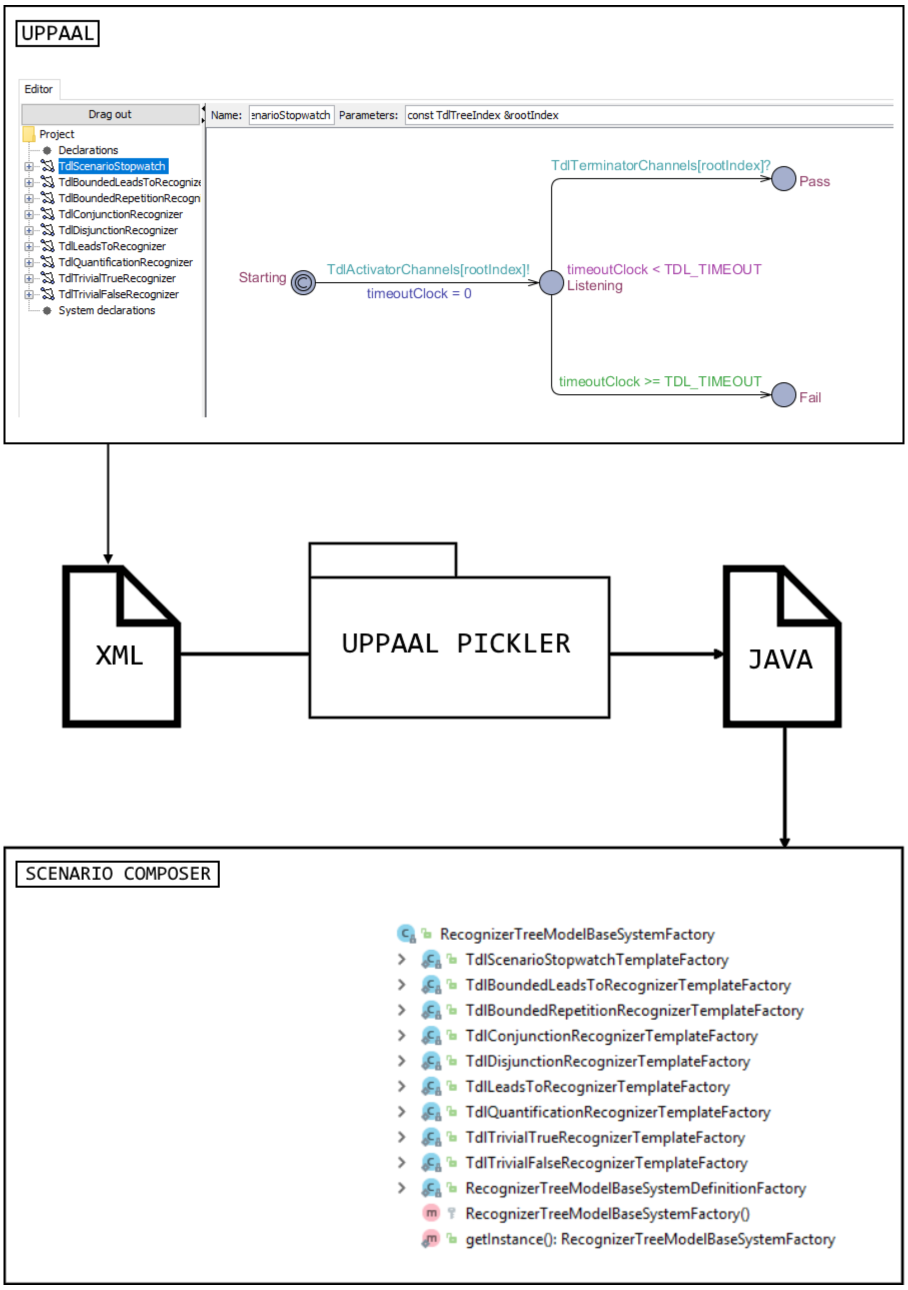

UPPAAL
Editor
Drag out
Name: enarioStopwatch
Parameters: const TdlTreeIndex &rootIndex
Project
Declarations
TdlScenarioStopwatch
TdlBoundedLeadsToRecognize
TdlBoundedRepetitionRecogn
TdlConjunctionRecognizer
TdlDisjunctionRecognizer
TdlLeadsToRecognizer
TdlQuantificationRecognizer
TdlTrivialTrueRecognizer
TdlTrivialFalseRecognizer
System declarations
Starting
TdlActivatorChannels[rootIndex]!
timeoutClock = 0
Listening
timeoutClock < TDL_TIMEOUT
TdlTerminatorChannels[rootIndex]?
Pass
timeoutClock >= TDL_TIMEOUT
Fail
XML
UPPAAL PICKLER
JAVA
SCENARIO COMPOSER
RecognizerTreeModelBaseSystemFactory
TdlScenarioStopwatchTemplateFactory
TdlBoundedLeadsToRecognizerTemplateFactory
TdlBoundedRepetitionRecognizerTemplateFactory
TdlConjunctionRecognizerTemplateFactory
TdlDisjunctionRecognizerTemplateFactory
TdlLeadsToRecognizerTemplateFactory
TdlQuantificationRecognizerTemplateFactory
TdlTrivialTrueRecognizerTemplateFactory
TdlTrivialFalseRecognizerTemplateFactory
RecognizerTreeModelBaseSystemDefinitionFactory
RecognizerTreeModelBaseSystemFactory()
getInstance(): RecognizerTreeModelBaseSystemFactory

Figure 34. Implementing recognizer templates using the UPPAAL pickler.

[1] As a side note, the term 'pickler' was used due to superficial similarity between the function of the plugin and the well-known food preservation technique.

The factory classes produced by the UPPAAL pickler are designed to be extensible. This extensibility was utilized in the scenario composer. Consequently, the implementation of the construction algorithm for recognizer models discussed in Section 5.5.2 was kept at under 400 lines of code.

The UPPAAL pickler itself required a little over 1000 lines of code (including a relatively small StringTemplate[1] file). At the time of writing the generated recognizer factory classes together consist of over 4000 lines of code. The large amount of generated code coupled with the fact that the recognizer templates were refactored several times during the development process implies that the chosen approach saved a considerable amount of time.

In summary, recognizers were implemented as automata templates in a single XML file using UPPAAL's GUI. This file was transformed into Java code via the UPPAAL pickler tool developed as part of this thesis. The resultant generated code is used by the scenario composer to construct recognizer trees as explained in the following section.

In the interest of brevity, a discussion on the recognizer template implementations embedded in the UPPAAL file mentioned above is omitted here. The interested reader can review the implementations in Appendices 1 – 8.

5.5.2 **Construction Algorithm**

In this section we present the algorithm used by the scenario composer to construct a recognizer tree (M^{WRAP}) according to a normalized and reduced TDL^{TP} AST. The algorithm maps to step (6) in Section 5.1

Two phases are needed in order to construct a recognizer model:

1. Each (non-trapset) operator node in the AST of the TDL^{TP} expression is mapped to a unique index.
2. Each (non-trapset) operator node in the AST of the TDL^{TP} expression is mapped to a recognizer template.

[1] StringTemplate is introduced in Section 4.3.4.

The indices from phase 1 are used for communication by recognizer processes instantiated from the corresponding templates[1]. Indices can be supplied during either a depth-first traversal or a breadth-first traversal of the AST. Simplified pseudocode for the indexing algorithm is depicted in Figure 35.

```
mode ← either depth-first or breadth-first
expression ← TDL^TP abstract syntax tree

currentIndex ← 0
indexMap ← empty map

deque ← empty deque
add expression.root to front

while deque isn't empty:
  node ← NIL
  if mode is depth-first:
    node ← deque.RemoveFirst()
  else:
    node ← deque.RemoveLast()

  indexMap[node] ← currentIndex
  currentIndex ← currentIndex + 1

  for child of node:
    deque.AddFirst(child)
```

Figure 35. Pseudocode for TDL^{TP} AST indexing algorithm.

The second step of the construction of M^{WRAP} requires another traversal of the AST. As was the case for the indexing algorithm, the traversal can be depth-first or breadth-first. The scenario composer implementation uses depth-first traversal.

For each node in the AST that does not represent a trapset expression, a recognizer template instantiation is added to M^{WRAP}. Additionally, if the recognizer template does not already exist in model M^{WRAP}, it will be added to it. Simplified pseudocode for this algorithm is presented in Figure 36.

[1] Each index maps to an ordinal for a globally declared broadcast channel array. Recognizer processes communicate by synchronizing over these shared channels. The UPPAAL manual provides more information on synchronizations: http://www.it.uu.se/research/group/darts/uppaal/help.php.

```
expression ← AST of TDL$^{TP}$ expression
indexMap ← map from nodes in expression
  to unique indices
templateManager ← recognizer template manager
M$^{WRAP}$ ← basic scaffolding for recognizer model

queue ← { expression.root }
while queue isn't empty:
  node ← queue.Dequeue()
  nodeIndex ← indexMap[node]
  parentIndex ← indexMap[node.parent]
  childIndices ← {}

  for child of node:
    childIndices.Append(indexMap[child])
    queue.Enqueue(child)

  type ← node.type
  recognizerTemplate ← templateManager.TemplateFor(type)
  instantiation ← templateManager
   .CreateInstantiation(
      recognizerTemplate,
      nodeIndex,
      parentIndex,
      childIndices,
      node
    )

  add instantiation to M$^{WRAP}$
  if recognizerTemplate not in M$^{WRAP}$:
    add recognizerTemplate to M$^{WRAP}$
```
Figure 36. Pseudocode for recognizer model construction algorithm.

In the algorithm above, the results of the previously described indexing phase are available in the `indexMap` variable. These indices are used to produce instantiations of recognizer templates. The `templateManager` object is simply a placeholder for logic in the scenario composer which hooks into the recognizer template factory classes described in Section 5.5.1.

There are subtle differences between the pseudocode presented in this section and the actual implementation of the scenario composer. Since these minor divergences are technical in nature, separate discussion is not needed here.

6 Validation

The following chapter presents our approach to validating the artifact produced as the result of this thesis.

As evident from Chapters 4 and 5, multiple relatively complex component interactions are involved in producing a test model based on user input. Thus, the need for automatically executed tests which verify the operation of these components is self-evident. However, due to time restrictions, only a limited set of fully automated tests was produced for the interpreter. Section 6.1 discusses this subject further.

In order to ensure the correct operation of the interpreter regardless of the lack of automated tests, manual integration testing was performed both during the development of individual components, and on the resultant artifact. Section 6.2 provides a summary of the results gathered from manual tests executed via the UI of the interpreter.

6.1 Automated Tests for Language Parsers

While there are several components in the architecture of the TDL^{TP} interpreter that can be considered candidates for automated testing, a selection had to be made. As parsers for the TDL^{TP} expression language (Section 4.4.6) and the UPPAAL system definition language (Section 4.4.5) play a major role in guaranteeing that the scenario composer receives data provided by the user accurately, they were chosen as targets for automated tests.

Verifying language parsers is not a trivial task. There are three immediate issues to resolve:

1. **Test input selection.** Since the length of an input string for the languages involved is only restricted by the memory limitations of the host system, and the set of valid character sequences alone is countably infinite, a subclass of inputs needs to be selected for testing the parser. The broad question here is which class of inputs to choose – valid, invalid, or some combination from both categories? Once this has

been decided, further specification of a finite set of concrete test inputs belonging to the class is needed.

2. **Testing strategy.** Simply selecting a set of input strings is not enough. A strategy for verifying that the parser handles them correctly is required. For example, would it be enough to check that the parser merely rejects invalid inputs?
3. **Approach for automation.** When both the inputs and the testing strategy have been selected, a method is needed for automating the execution of the corresponding test cases.

To resolve the first issue, it was decided to restrict the class of test inputs to syntactically valid strings. For the corresponding tests to be implementable in a realistic amount of time, this class was further reduced to a set of strings representing essential language constructs[1]. By verifying that the parsers can handle these basic constructs, we at least have some guarantee that the composed artifact will not produce an invalid test model due to misinterpretation of user input.

After selecting test inputs, a strategy for applying them had to be formulated. The trivial approach of checking whether the parser consumes a test input without error is not sufficiently informative, thus it was not chosen. A more suitable option is to inspect the AST generated by the parser as the result of processing an input string. The AST's structure will not vary for the same input over multiple test runs and accurately encodes the information that the parser is capable of gathering.

The term "inspection" used above implies the need for some representation of expected output. The classes discussed in Sections 4.4.1 and 4.4.2 – which the parsers in question employ to produce their output ASTs – could be used for this end. A test case would therefore be an input string combined with a procedure which constructs the corresponding expected AST. A test passes if the AST produced by the parser and the AST generated by the test procedure are equivalent. However, the overhead of having to write AST constructor code for each test input eliminates this option.

[1] An example of an essential language construct for TDL^{TP} is a subexpression containing the absolute complement operation: `!TS1`. For the UPPAAL system definition language, a basic construct could be selection from a bounded type: '`t : int[5, 10]`'. However, there are some more elaborate constructs subject to testing such single-line declarations of multiple variables: '`int var1, var2, array[10];`'.

The solution was to implement another language parser – this time for a variant of the *symbolic expression* (s-expression) notation.

S-expressions were invented for the Lisp programming language and are used for representing tree structures in a simplified manner. A compressed ANTLR grammar for the extended version of s-expressions developed for this thesis is depicted in Figure 37.

```
sExpr : sequence ;

sequence : '(' item ('.' item)* ')'
         | '()'
         ;

item    : sequence
        | string
        ;

string  : DELIMITED_STRING
        | NON_DELIMITED_STRING
        ;

DELIMITED_STRING        : '"' (ALPHANUMERIC | SPECIAL
                               |ESCAPED_RESERVED_TOK)* '"' ;
NON_DELIMITED_STRING    : (ALPHANUMERIC | SPECIAL
                               |ESCAPED_RESERVED_TOK)+ ;
ALPHANUMERIC            : [A-Za-z0-9] ;
SPECIAL                 : [#~!@$%^&*_\-?`=<>|{}[\]/:;,+] ;
ESCAPED_RESERVED_TOK    : '\\.' | '\\(' | '\\)' | '\\"' ;
```

Figure 37. ANTLR grammar for s-expressions.

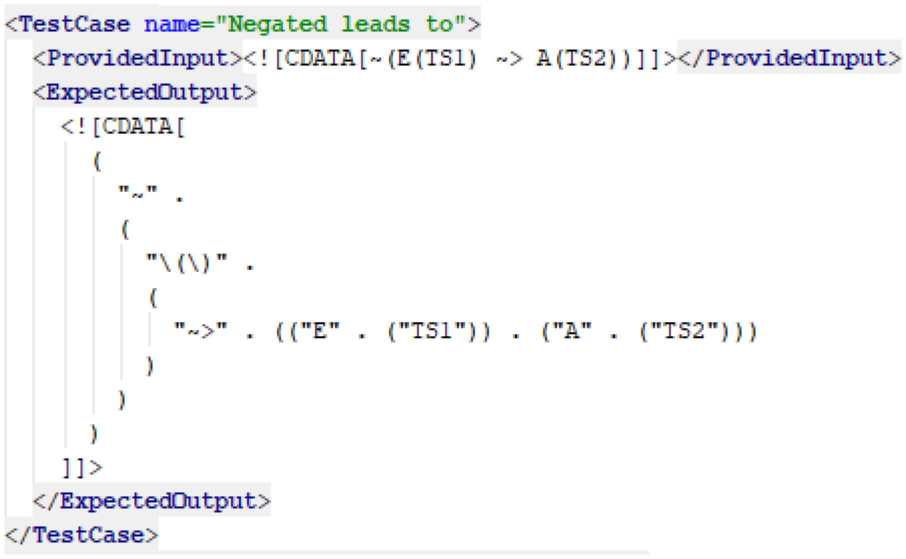
A sentence in the language specified above is a list of items which allows for recursive nesting. For example: `(“x” . (“y” . “z”))`. When an s-expression is parsed, the output object structure is a tree whose nodes can either be lists of other nodes or simple strings. The similarity between the language parser components implemented as part of the TDL^{TP} interpreter and the s-expression parser supplied for testing is obvious: both produce tree structures as output.

In our testing strategy, the tree structure of ASTs is exploited. An expected output for an input string is specified as an s-expression. Additionally, mapping utilities are provided in order to convert the output AST of a parser under test into an equivalent s-expression’s AST. Testing becomes the simple task of comparing whether the tree parsed from the expectation s-expression is equivalent to the s-expression derived from the targeted parser’s output.

The chosen approach has three major flaws:

1. Whoever writes the test needs to know how a test input string maps to the meta-tokens encoded in the AST-to-s-expression transformer classes used in the approach.
2. It is arduous to manually specify large s-expressions as expected outputs.
3. A code defect in a transformer class may lead to the detection of errors that do not exist.

Despite these shortcomings, a major argument for using the s-expression strategy is that it facilitates simple and reliable automation. An example test case specification from a configuration file produced as part of this work is depicted in Figure 38.



```
<TestCase name="Negated leads to">
  <ProvidedInput><![CDATA[~(E(TS1) ~> A(TS2))]]></ProvidedInput>
  <ExpectedOutput>
    <![CDATA[
      (
        "~" .
        (
          "\(\)" .
          (
            "~>" . (("E" . ("TS1")) . ("A" . ("TS2")))
          )
        )
      )
    ]]>
  </ExpectedOutput>
</TestCase>
```

Figure 38. Example s-expression-based language parser test case.

Using the s-expression strategy discussed in this section, a total of 109 automated test cases were implemented for the TDL^{TP} interpreter. 39 of those are for the TDL^{TP} expression parser, and 70 for the UPPAAL system language parser. Additionally, 39 code generation tests[1] were supplied for the TDL^{TP} grammar implementation described in Section 4.4.3. This led to the discovery of a handful of programming defects, which were subsequently resolved.

[1] For code generation, the test input is specified as an s-expression. General transformer utilities were developed to convert s-expressions into a TDL^{TP} ASTs. The expected output is provided as a TDL^{TP} expression, so the verification step in the test is essentially a string comparison.

In summary, while automated tests are necessary for verifying TDL^{TP} interpreter functionality, time limitations had to be taken into account. To ensure that the interpreter handles at least part of the user's input correctly, a collection of tests for the language components involved were implemented. These tests are automatically executed during the build process of the artifact.

6.2 Manual Integration Tests

It is not enough to simply follow a set of good programming guidelines and assume that this will lead to the correct operation of the resultant software artifact. The implementation needs to be verified by tests. As mentioned in previous sections, there was no time for complete test automation, so a set of manual integration tests was specified and executed against the interpreter's user interface. The total number of these test cases is 95. All of them are tabulated in Appendix 10.

In general, a manual integration test for the interpreter consists of an input model, a TDL^{TP} expression, and a generally stated output expectation. To illustrate this, we introduce test case *TC-TSExpr-LP-6* from Appendix 10.

The first test input in *TC-TSExpr-LP-6* is an artificial UPPAAL SUT model annotated with two trapsets: TS$_1$ and TS$_2$. This model is depicted in Figure 39.

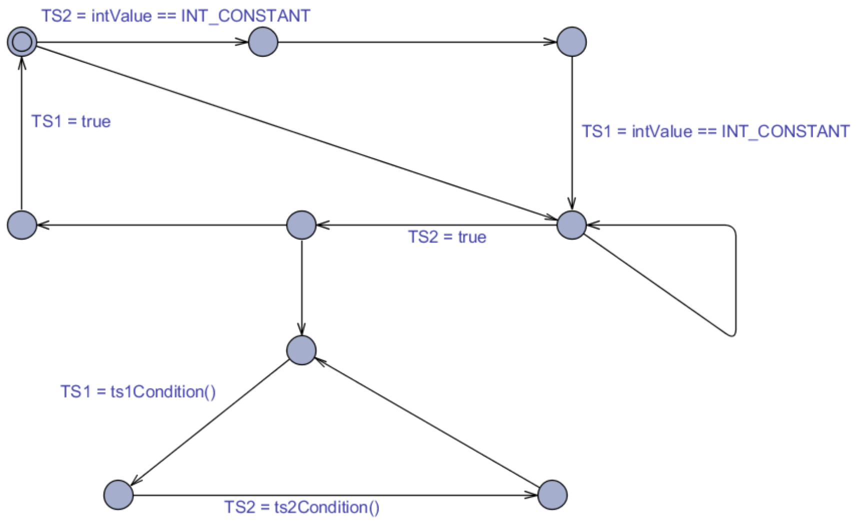

TS2 = intValue == INT_CONSTANT
TS1 = true
TS1 = intValue == INT_CONSTANT
TS2 = true
TS1 = ts1Condition()
TS2 = ts2Condition()

Figure 39. Example manual test case model input.

The trapsets in the input model each contain conditional and non-conditional traps. Trap conditionality is explained in Section 2.3.3.

The TDL^{TP} expression input used in conjunction with the model is provided in Figure 40.

```
A(TS1 ; TS2)
```

Figure 40. Example manual test case TDL^{TP} expression input.

The expression contains a universal trapset quantification operation (`A`) applied to the linked trapset of TS_1 and TS_2 (`TS1 ; TS2`).

As output, we expect the interpreter to produce a test model where the linked trapset expression has been applied to the input model according to Mapping M3 from Section 5 of [1] with conditional traps handled appropriately. Figure 41 presents the output of a manual test run executed using the inputs for *TC-TSExpr-LP-6*.

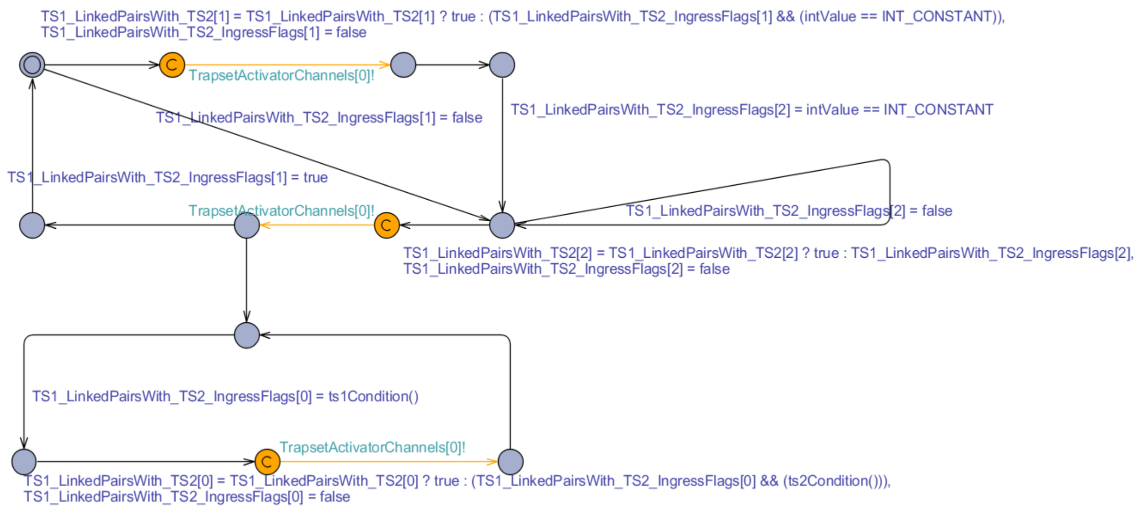

TS1_LinkedPairsWith_TS2[1] = TS1_LinkedPairsWith_TS2[1] ? true : (TS1_LinkedPairsWith_TS2_IngressFlags[1] && (intValue == INT_CONSTANT)),
TS1_LinkedPairsWith_TS2_IngressFlags[1] = false
C
TrapsetActivatorChannels[0]!
TS1_LinkedPairsWith_TS2_IngressFlags[1] = false
TS1_LinkedPairsWith_TS2_IngressFlags[2] = intValue == INT_CONSTANT
TS1_LinkedPairsWith_TS2_IngressFlags[1] = true
TrapsetActivatorChannels[0]!
C
TS1_LinkedPairsWith_TS2_IngressFlags[2] = false
TS1_LinkedPairsWith_TS2[2] = TS1_LinkedPairsWith_TS2[2] ? true : TS1_LinkedPairsWith_TS2_IngressFlags[2],
TS1_LinkedPairsWith_TS2_IngressFlags[2] = false
TS1_LinkedPairsWith_TS2_IngressFlags[0] = ts1Condition()
TrapsetActivatorChannels[0]!
C
TS1_LinkedPairsWith_TS2[0] = TS1_LinkedPairsWith_TS2[0] ? true : (TS1_LinkedPairsWith_TS2_IngressFlags[0] && (ts2Condition())),
TS1_LinkedPairsWith_TS2_IngressFlags[0] = false

Figure 41. Example manual test run output model.

The model depicted above has been slightly edited in UPPAAL's GUI to make the auxiliary variable assignments and channel synchronizations injected by the interpreter more discernible.

To determine the result of a manual test execution, the output model is inspected both visually and, if applicable, using the UPPAAL model-checker (discussed in Section 2.2.3). If the auxiliary test constructs in the model match expectations, and the test stopwatch automaton can reach its *pass* location during model-checking, then we know that the interpreter can handle conditional traps for the linked trapset operation correctly[1].

In conclusion, several manual test cases were documented and executed against the TDL^{TP} interpreter. A single user-friendliness issue, at least 4 significant logical errors, and a handful of code defects were discovered as the result of these executions. Problems discovered during testing were subsequently resolved.

[1] To an extent. There may of course be unforeseen configurations of inputs for which the interpreter will produce an erroneous test model. At least the basic use case for the linked trapset operator has been verified in the example test case.

7 Summary

The goal of this thesis was to implement an interpreter for the Test Purpose Specification Language TDL^{TP}. To achieve this, a collection of modules was assembled and composed into a software artifact which can accept TDL^{TP} expressions combined with UPPAAL SUT models as input and produce test models as output. The artifact was supplemented with a simple command-line interface in order to facilitate its inclusion in an MBT workflow.

The functionality of the interpreter was validated with a set of automated component tests and manually executed integration tests. Through these means it was verified to a degree that the interpreter functions according to the theory of TDL^{TP}.

During development, a number of ambiguities in the theory surfaced and were documented in this thesis[1]. These open questions represent opportunities for future theoretical work in the field of model-based testing in general and UPPAAL MBT specifically. The internals of the artifact were designed with this in mind – thanks to modularity and the use of intuitive software interfaces, improvements and extensions can be included with a relatively minimal development effort.

On the topic of future work, three primary dimensions for enhancement deserve to be mentioned: user-friendliness, internal design, and test coverage. We discuss them below.

User-friendliness. SUT models provided as input to the interpreter are annotated with trap variables which connect them to corresponding TDL^{TP} test purpose specifications. At this point the annotations are supplied by the user via UPPAAL's graphical interface, which means that the input definition process is somewhat repetitive. Since a custom parser for UPPAAL models was implemented as part of this thesis, it would be possible to implement a supplementary front-end tool for the interpreter which is specifically

[1] Section 5.3 and 5.2.2. Appendices 5 – 6.

designed for annotating a SUT model with traps. The inclusion of such a tool would make the interpreter more appealing to use.

Internal design. Though the components in the interpreter were refactored several times with the goal of improving their design and reusability, improvements are still possible. For example, the grammar parser for TDL^{TP} is housed in the same component as the code generator for the language. The input for the former is the output for the latter, and vice-versa. However, due to technical reasons, their logic is isolated within the component, which leads to partial duplication. Some generally applicable approach could be developed which would bridge code generation and grammar parsing for the interpreter in order to eliminate this duplication.

Test Coverage. Due to time limitations, most of the validation effort was manual in nature. Fully automated test coverage is a welcome addition to the project and basic scaffolding which could facilitate this has already been implemented.

In conclusion, we believe the objective of this thesis has been successfully executed. The final implementation is extensible and fulfils its purpose. The result was validated as adequately as possible and entry points for improvement were supplied.

References

[1] J. Vain, E. Halling, A. Boyarchuk and O. Illiashenko, "Test Scenario Specification Language for Model-based Testing," *International Journal of Computing,* p. 1 – 14, 2019.

[2] J. Lindholm, "Model-Based Testing," University of Helsinki, 2006.

[3] L. Apfelbaum and J. Doyle, "Model Based Testing," in *Software Quality Week Conference,* 1997.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson and W. Yi, "UPPAAL – a Tool Suite for Automatic Verification of Real-time Systems," in *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control: Verification and Control,* New Jersey, 1996.

[5] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science,* vol. 126, no. 2, p. 185 – 235, 1994.

[6] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson and A. Skou, "Testing Real-time Systems Using UPPAAL," in *Formal methods and testing*, Springer-Verlag, 2008, p. 77 – 117.

[7] M. Utting, A. Pretschner and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software Testing, Verification & Reliability,* vol. 22, no. 5, p. 297 – 312, 2012.

[8] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science(, Springer-Verlag, 2005.

[9] A. David, K. G. Larsen and G. Behrmann, *A Tutorial on UPPAAL 4.0,* Department of Computer Science, Aalborg University, 2006.

[10] J. Bengtsson and W. Yi, "Timed Automata: Semantics, Algorithms and Tools," *Lecture Notes in Computer Science,* vol. 3089, p. 87 – 124, 2004.

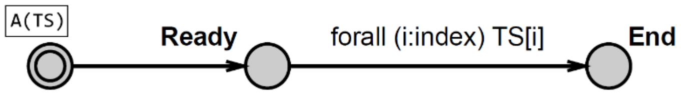
[11] J. Vain, A. Anier and E. Halling, "Provably Correct Test Development for Timed Systems," in *Databases and Information Systems VIII : Selected Papers from the Eleventh International Baltic Conference, Baltic DB&IS* , vol. 270, Amsterdam, IOS Press, 2014, p. 289 – 302.

[12] T. Parr, Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages, Pragmatic Bookshelf, 2009.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., 1995.

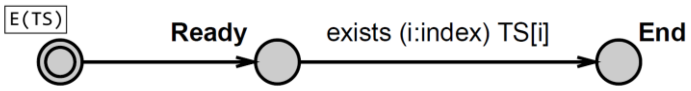
Appendix 1 – Trapset Quantifier Recognizer Template

Mapping M4 in Section 5.2 of [1] specifies a recognizer automaton for the TDL^{TP} *universal trapset quantification* operator (‘`A`’). Figure 5 in [1], which depicts this automaton, is reproduced below.



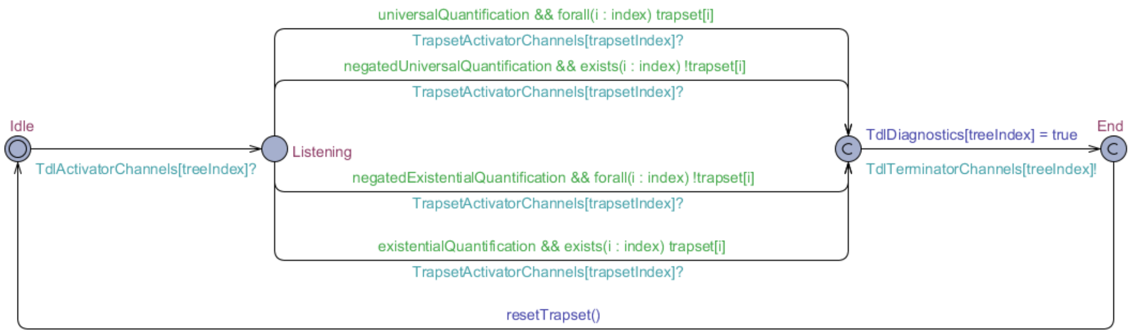
Recognizer automaton for universal trapset quantification [1, p. 7].

Mapping M5 in Section 5.2 of [1] specifies a recognizer automaton for the TDL^{TP} *existential trapset quantification* operator (‘`E`’). Figure 6 in [1], which depicts this automaton, is reproduced below.



Recognizer automaton for existential trapset quantification [1, p. 7].

The trapset quantifier recognizer template developed as part of this thesis based on Mappings M4 – M5 is presented below.


universalQuantification && forall(i : index) trapset[i]
TrapsetActivatorChannels[trapsetIndex]?
negatedUniversalQuantification && exists(i : index) !trapset[i]
TrapsetActivatorChannels[trapsetIndex]?
Idle
TdlActivatorChannels[treeIndex]?
Listening
negatedExistentialQuantification && forall(i : index) !trapset[i]
TrapsetActivatorChannels[trapsetIndex]?
existentialQuantification && exists(i : index) trapset[i]
TrapsetActivatorChannels[trapsetIndex]?
TdlDiagnostics[treeIndex] = true
TdlTerminatorChannels[treeIndex]!
End
resetTrapset()

Trapset quantifier recognizer template.

As evident from the figure above, existential and universal quantification were composed into a single parameterized template. The parameters for this template are presented in the table below.

Trapset quantifier template parameters.

Parameter	**Type**	**Description**
`const bool universal`	Constant Boolean.	Whether the recognizer should quantify universally or existentially.
`const bool negated`	Constant Boolean.	Whether the quantifier is negated.
`const TdlTreeIndex treeIndex`	Constant integer.	The index of the quantifier in the AST of the TDL^{TP} expression.
`const TrapsetIndex trapsetIndex`	Constant integer.	The index of the trapset operand of the quantifier in the AST of the TDL^{TP} expression.
`const int trapsetSize`	Constant integer.	The size of the trapset over which the recognizer quantifies.
`bool &trapset[0]`	Reference to Boolean array.	A reference to an array of flags where each flag represents a distinct trap in the test model. The size of this array is specified as 0 in the template, which is not a valid array size in UPPAAL. The size value will be corrected by the scenario composer before the template is added to the test model.

The `treeIndex` parameter is used to attach instances of the quantifier recognizer to the recognizer tree. The emission synchronization `TdlActivatorChannels[treeIndex]?` and the reception synchronization `TdlTerminatorChannels[treeIndex]!` instrument the quantifier template with input and output facilities, respectively. The parent recognizer of the quantifier in the tree will use these synchronizations to activate the quantifier recognizer and retrieve input from it.

The `trapsetIndex` parameter is used to index into an array of broadcast channels reserved for trapsets – `TrapsetActivatorChannels`. Whenever a trap from a given trapset is visited, a synchronization signal will be broadcast on the corresponding trapset activator channel. The recognizer assigned to the trapset in question will receive this signal (via the reception synchronization `TrapsetActivatorChannels[trapsetIndex]?`) and subsequently test whether the quantification condition has been fulfilled. If it has, the quantifier will return an output signal to its parent recognizer.

Some of the transition labels in the template contain references to variables and functions declared in the declarations section for the recognizer template. These are presented below.

Trapset quantifier template local declarations.

Declaration	**Type**	**Description**
`const bool negatedUniversalQuantification = universal && negated;`	Constant Boolean.	Whether the recognizer should check for negated universal quantification.
`const bool negatedExistentialQuantification = !universal && negated;`	Constant Boolean.	Whether the recognizer should check for negated existential quantification.
`const bool universalQuantification = universal && !negated;`	Constant Boolean.	Whether the recognizer should check for universal quantification.
`const bool existentialQuantification = !universal && !negated;`	Constant Boolean.	Whether the recognizer should check for existential quantification.
`typedef int[0, trapsetSize - 1] index;`	Custom bounded integer type.	Used for iteration in `resetTrapset`.

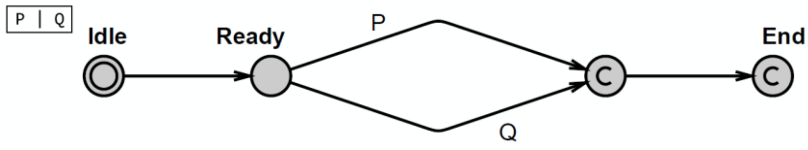
`void resetTrapset() {`
`  for (i : index) {`
`    trapset[i] = false;`
`  }`
`}`
	Void function.	Resets the state of the trapset flag array assigned to the recognizer after output has been returned to the parent recognizer.

The `const` modifier used in the declarations above helps UPPAAL reduce the state space of the test model during trace generation. For example, if the variable `universalQuantification` is constantly false, it will not be possible to take a transition whose guard conjunction refers to this variable. Therefore, the transition does not need to be considered when calculating possible state paths for the recognizer.

One issue with the trapset quantifier template provided in this appendix is that is not completely parameterized – the trapset array parameter needs to be modified by the scenario composer. Consequently, a test model may contain more than one quantifier template under different names. This is due to the syntax limitations of UPPAAL – there is no way to declare a reference to an array variable of unspecified size as the parameter of a template.

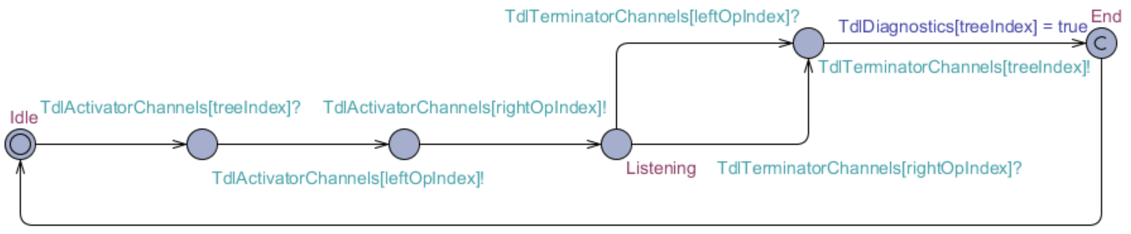
Appendix 2 – Disjunction Recognizer Template

Mapping M7 in Section 5.2 of [1] specifies a recognizer automaton for the TDL^{TP} *disjunction* operator (‘|’). Figure 8 in [1], which depicts this automaton, is reproduced below.



Recognizer automaton for disjunction [1, p. 7].

The recognizer template developed as part of this thesis based on Mapping M7 is presented below.


TdlTerminatorChannels[leftOpIndex]?
TdlDiagnostics[treeIndex] = true
End
C
TdlTerminatorChannels[treeIndex]!
Idle
TdlActivatorChannels[treeIndex]?
TdlActivatorChannels[rightOpIndex]!
TdlActivatorChannels[leftOpIndex]!
Listening
TdlTerminatorChannels[rightOpIndex]?

Disjunction recognizer template.

Parameters for the template are presented in the table below.

Disjunction recognizer template parameters.

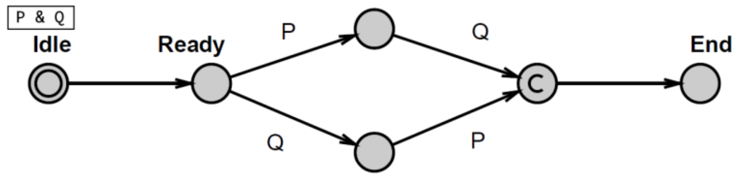
Parameter	**Type**	**Description**
`const TdlTreeIndex treeIndex`	Constant integer.	The index of the operator in the AST of the TDL^{TP} expression.
`const TdlTreeIndex leftOpIndex`	Constant integer.	The index of the left child of the operator in the AST of the TDL^{TP} expression.
`const TdlTreeIndex rightOpIndex`	Constant integer.	The index of the right child of the operator in the AST of the TDL^{TP} expression.

The `treeIndex` parameter used in the template is analogous to the one used for the quantifier template described in Appendix 1.

Parameters `leftOpIndex` and `rightOpIndex` are used to communicate with the child recognizers of the disjunction recognizer.

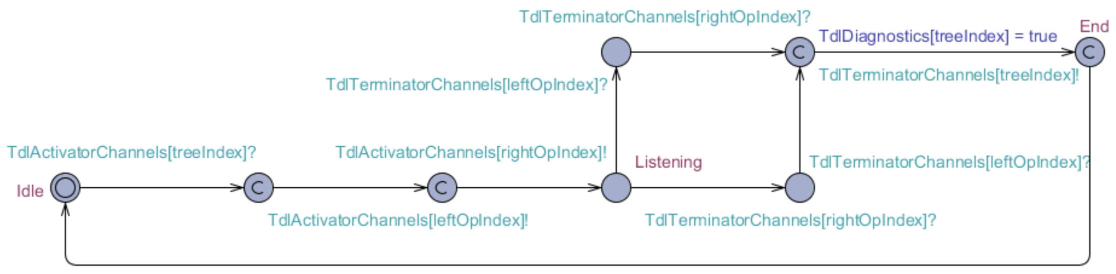
Appendix 3 – Conjunction Recognizer Template

Mapping M6 in Section 5.2 of [1] specifies a recognizer automaton for the TDL^{TP} *conjunction* operator (‘&’). Figure 8 in [1], which depicts this automaton, is reproduced below.



Recognizer for conjunction [1, p. 7].

The recognizer template developed as part of this thesis based on Mapping M7 is presented below.

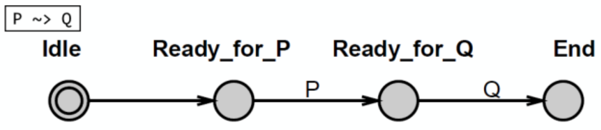

TdlTerminatorChannels[rightOpIndex]?
TdlDiagnostics[treeIndex] = true
End
TdlTerminatorChannels[treeIndex]!
TdlTerminatorChannels[leftOpIndex]?
TdlActivatorChannels[treeIndex]?
TdlActivatorChannels[rightOpIndex]!
Listening
TdlTerminatorChannels[leftOpIndex]?
Idle
TdlActivatorChannels[leftOpIndex]!
TdlTerminatorChannels[rightOpIndex]?

Conjunction recognizer template.

Template parameters for conjunction are identical to the parameters declared for the disjunction recognizer in Appendix 2.

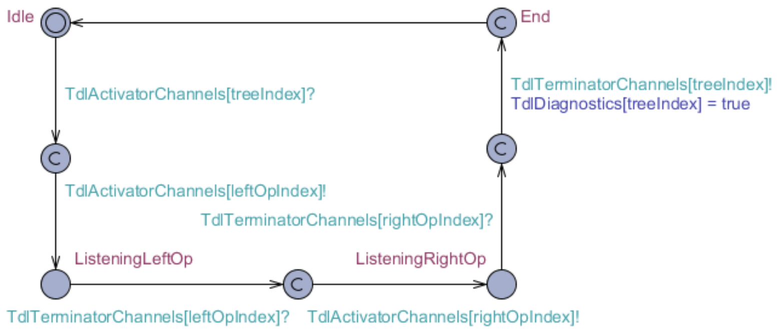
Appendix 4 – Leads To Recognizer Template

Mapping M8 in Section 5.2 of [1] specifies a recognizer automaton for the TDL^{TP} *leads to* operator (‘~>’). Figure 9 in [1], which depicts this automaton, is reproduced below.



Recognizer automaton for leads to [1, p. 7].

The recognizer template developed as part of this thesis based on Mapping M8 is provided below.

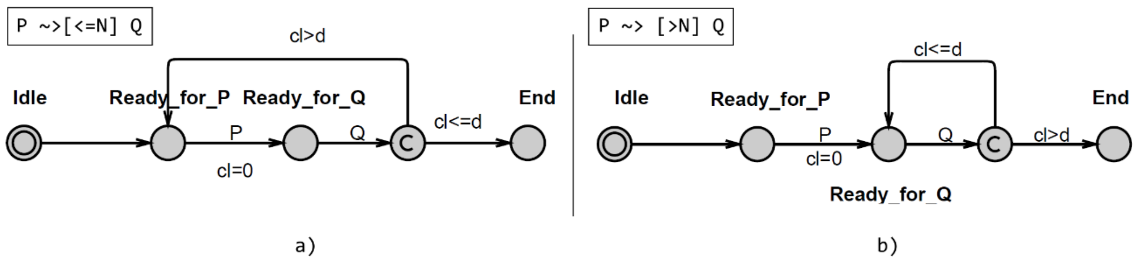

Idle
End
TdlActivatorChannels[treeIndex]?
TdlTerminatorChannels[treeIndex]!
TdlDiagnostics[treeIndex] = true
TdlActivatorChannels[leftOpIndex]!
TdlTerminatorChannels[rightOpIndex]?
ListeningLeftOp
ListeningRightOp
TdlTerminatorChannels[leftOpIndex]?
TdlActivatorChannels[rightOpIndex]!

Leads to recognizer template.

Template parameters for leads to are identical to the parameters declared for the disjunction recognizer in Appendix 2.

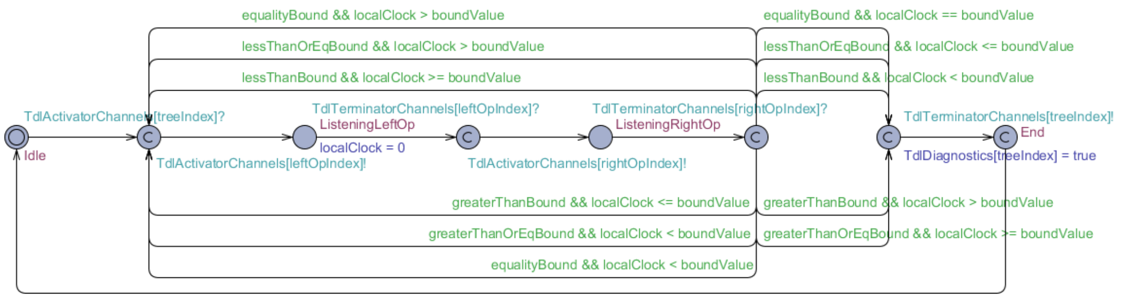
Appendix 5 – Time-Bounded Leads To Recognizer Template

Mapping M9 in Section 5.2 of [1] specifies an automaton for the TDL^{TP} *time-bounded leads to* operator (‘`~> [(<|<=|=|>=|>) N]`’). Figures 10a and 10b in [1], which depict bounded leads to with conditions ‘`<= N`’ and ‘`> N`’ respectively, are reproduced below.



Recognizers for time-bounded leads to with time constraint `<= N` (a), and time constraint `> N` (b) [1, p. 7].

The recognizer template developed as part of this thesis based on Mapping M9 is provided below.



equalityBound && localClock > boundValue
lessThanOrEqBound && localClock > boundValue
lessThanBound && localClock >= boundValue
equalityBound && localClock == boundValue
lessThanOrEqBound && localClock <= boundValue
lessThanBound && localClock < boundValue
TdlActivatorChannels[treeIndex]?
Idle
TdlTerminatorChannels[leftOpIndex]?
ListeningLeftOp
localClock = 0
TdlActivatorChannels[leftOpIndex]!
TdlTerminatorChannels[rightOpIndex]?
ListeningRightOp
TdlActivatorChannels[rightOpIndex]!
TdlTerminatorChannels[treeIndex]!
End
TdlDiagnostics[treeIndex] = true
greaterThanBound && localClock <= boundValue
greaterThanOrEqBound && localClock < boundValue
equalityBound && localClock < boundValue
greaterThanBound && localClock > boundValue
greaterThanOrEqBound && localClock >= boundValue

Time-bounded leads to recognizer template.

Parameters for this template are a superset of the parameters defined for the disjunction recognizer template in Appendix 2. Additional parameters for time-bounded leads to are provided in the following table.

Additional parameters for the time-bounded leads to recognizer template.

Parameter	Type	Description		
`const BoundType boundType`	Constant integer.	`BoundType` is a custom bounded integer type whose values range from 1 – 5. As shown below, each value maps to an inequality operator.		
		Constant	**Value**	**Inequality**
		`BOUND_EQ`	1	Equals.
		`BOUND_GT`	2	Greater than.
		`BOUND_GTE`	3	Greater than or equal to.
		`BOUND_LT`	4	Less than.
		`BOUND_LTE`	5	Less than or equal to.
`const BoundValue boundValue`	Constant integer.	The value part of the constraint specified by the combination of `boundType` and `boundValue`. Example: `5` in ‘`< 5`’.		

Some of the transition labels in the template contain references to variables declared in the declarations section for the recognizer template. The corresponding declarations are presented in the table below.

Time-bounded leads to template local declarations.

Declaration	Type	Description
`const bool lessThanBound = (boundType == BOUND_LT);`	Constant Boolean.	Whether the bound inequality is ‘less than’.
`const bool lessThanOrEqBound = (boundType == BOUND_LTE);`	Constant Boolean.	Whether the bound inequality is ‘less than or equal to’.
`const bool greaterThanBound = (boundType == BOUND_GT);`	Constant Boolean.	Whether the bound inequality is ‘greater than’.
`const bool greaterThanOrEqBound = (boundType == BOUND_GTE);`	Constant Boolean.	Whether the bound inequality is ‘greater than or equal to’.
`const bool equalityBound = (boundType == BOUND_EQ);`	Constant Boolean.	Whether the bound inequality is ‘equal to’.
`clock localClock;`	Clock.	Used for determining whether the time bound has been satisfied.

The `const` modifier used in some of the declarations above helps UPPAAL reduce the state space of the test model during trace generation. For example, if the variable `equalityBound` is constantly false, all transitions which use this variable in their guard conjunction can be ignored when calculating available transitions from the recognizer's current location.

It was discovered during the implementation of the interpreter that the bounded leads to recognizer specified in [1, p. 7] does not function according to the semantics of bounded leads to provided in Definition 12 of [1]. This definition is reproduced below.

'Time-bounded leads to' $[\![SE_1 \rightsquigarrow_{[\otimes\, n]}\ SE_2]\!]$ means that SE_2 must occur after SE_1 and the time instance of the occurrence of SE_2 (measured relative to the occurrence of SE_1) satisfies the constraint $\otimes\, n$ where $\otimes\ \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$:

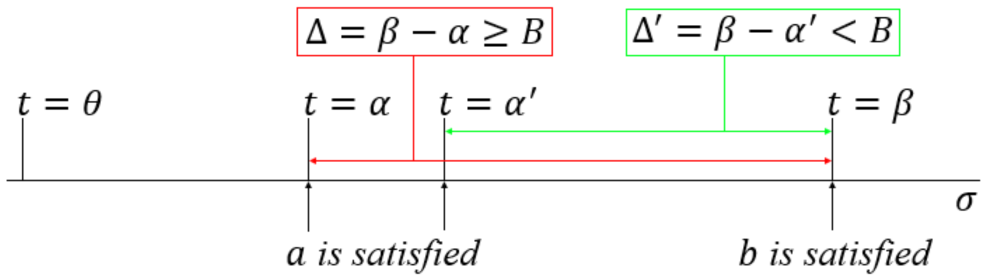
$[\![SE_1 \rightsquigarrow_{[\otimes\, n]}\ SE_2]\!]$ if and only if $\forall \sigma \exists k, l \geq k \in \mathbb{N} : [\![SE_1]\!]_{\sigma^k} \Rightarrow^{\otimes\, n} [\![SE_2]\!]_{\sigma^l}$ [1, p. 5].

According to this definition, the bounded leads to recognizer should return a success signal to its parent when the occurrence instances of its operands satisfy the specified time constraint.

Let us assume a test model T generated from a SUT model M according to the TDL^{TP} expression $\epsilon = A(TS_1) \rightsquigarrow_{[< B]}\ E(TS_2)$, where the constraint value $B > 0$. Let σ be a trace in the state space of M. At time step α, let us assume subexpression $a = A(TS_1)$ is satisfied in σ. At time step $\beta = \alpha + \Delta$, where $\Delta > B$, let us assume subexpression $b = E(TS_2)$ is satisfied in σ. According to the logic of TDL^{TP}, the recognizer R assigned to the bounded leads to expression in ϵ is activated at time $\theta = 0$.

R will start measuring the time distance between subexpressions a and b at time step α. Since the measured time distance ($\Delta = \beta - \alpha$) is greater than B, the recognizer will not return a success signal. At this point the definition of bounded leads to is satisfied by the recognizer.

Now let us assume subexpression a is also satisfied in σ at time step $\alpha' = \beta - \Delta'$ so that $\alpha' > \alpha$ and $\Delta' > 0$. Additionally, assume that the time difference $\Delta' = \beta - \alpha' < B$. This is depicted below.



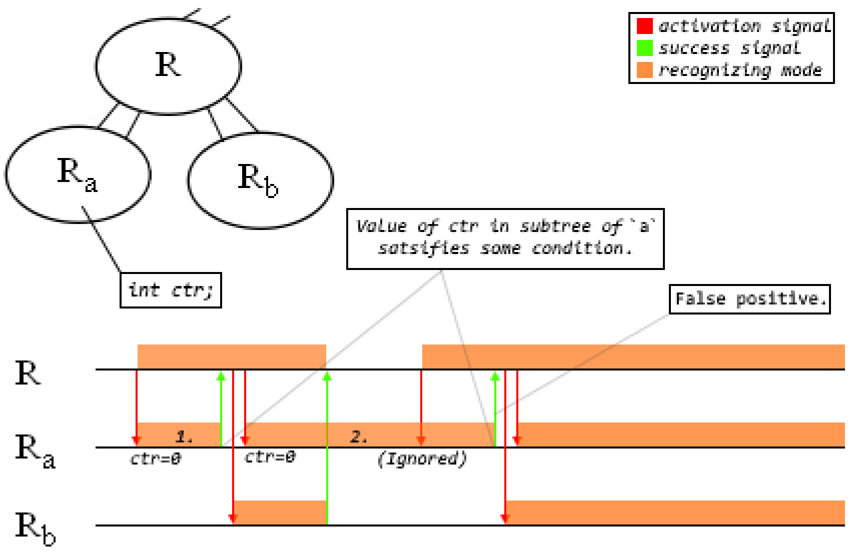
Trace example for σ.

If the recognizer assigned to the leads to operator in ϵ would start measuring the time distance between subexpressions a and b at time step α', it would in fact return a success signal at time step β since the measured time Δ' satisfies the bound $\Delta' < B$.

The problem here is that R will inevitably begin to measure time starting at time step α because that is the moment when the child recognizer for a first returns a success signal. Therefore, the implementation guidelines for bounded leads to specified in [1] do not match the definition of the operator.

One solution would be to reset the clock in R used for measuring the time distance between subexpressions a and b every time subexpression a is satisfied (`localClock = 0`). This would require the recognizer process to reactivate the corresponding operand recognizer after every occurrence of a in the trace.

The proposed solution has a major defect. Whenever R returns a success signal for ϵ, the operand recognizer for a (R_a) may still be in its recognizing mode (since the last time it was reactivated by R). R_a's recognizing mode may therefore continue beyond the next activation of R. If a were a more complex subexpression whose corresponding recognizer subtree contained processes with clocks or counters, the next time R is activated, it may occur that the recognizer for a signals success to R prematurely. This is depicted in the following figure.



Corner-case for bounded leads to recognizer R with operand recognizers R_a and R_b.

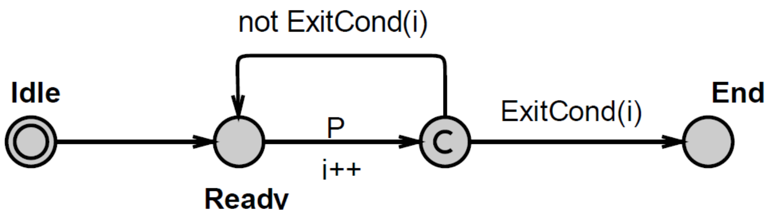
In the figure above, a time-bounded leads to subtree rooted at recognizer R sends an activating signal to its first operand recognizer, R_a, which enters its recognizing mode (1). After R_a returns a success signal, R activates its second operand recognizer, R_b, and immediately moves R_a back to its recognizing mode (2) (so that it could reset its clock when applicable). When R_b returns a success signal, R exits its recognizing mode and similarly returns a success signal to its parent. R_a, however, continues recognizing, which means it will ignore any subsequent activation signals, and returns results collected since its previous activation.

To solve the issue described above, an option would be to implement a recursive reset mechanism for recognizers. Namely, when R receives a success signal from R_b, it should send a reset signal to R_a, thus forcing the corresponding recognizer subtree to exit its recognizing mode and become available for future activations. However, this would require us to add externally accessible reset transitions to almost every location pair in every recognizer. Not only would this increase the state space of the test model, it would also be difficult to ensure that the recognizer functions correctly due to the amount of transitions involved.

The problems mentioned here imply the need for future revisions of the theory of TDL^{TP}. Solving these issues is outside of the scope of this thesis. Therefore, the bounded leads to recognizer template implementation described at the beginning of this appendix corresponds to the original guidelines provided in [1].

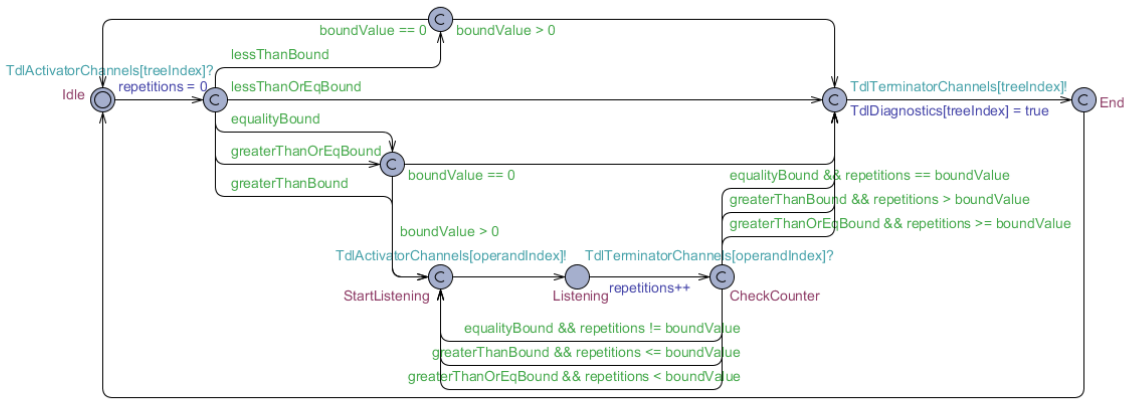
Appendix 6 – Conditional Repetition Recognizer Template

Mapping M10 in Section 5.2 of [1] specifies an automaton for the TDL^{TP} *conditional repetition* operator (‘`#[(<|<=|=|>=|>)  N]`’). Figure 11 in [1], which depicts this automaton, is reproduced below.



Recognizer for leads to [1, p. 8]

The recognizer template developed as part of this thesis based on Mapping M10 is provided below.



boundValue == 0
boundValue > 0
lessThanBound
TdlActivatorChannels[treeIndex]?
repetitions = 0
Idle
lessThanOrEqBound
TdlTerminatorChannels[treeIndex]!
TdlDiagnostics[treeIndex] = true
End
equalityBound
greaterThanOrEqBound
greaterThanBound
boundValue == 0
equalityBound && repetitions == boundValue
greaterThanBound && repetitions > boundValue
greaterThanOrEqBound && repetitions >= boundValue
boundValue > 0
TdlActivatorChannels[operandIndex]!
TdlTerminatorChannels[operandIndex]?
StartListening
Listening
repetitions++
CheckCounter
equalityBound && repetitions != boundValue
greaterThanBound && repetitions <= boundValue
greaterThanOrEqBound && repetitions < boundValue

Conditional repetition recognizer template implementation.

The set of template parameters specified for the conditional repetition recognizer are similar to the ones specified for the time-bounded leads to recognizer template discussed in Appendix 5. The only difference is that the bound parameters define a constraint on the number of repetitions of state configurations satisfying the subexpression represented by the recognizer’s operand. Repetitions are counted using the integer variable `repetitions` declared in the template’s local declarations section.

It was discovered during the implementation of the interpreter that the conditional repetition recognizer specified in [1, p. 7] does not function according to the semantics provided in Definition 13 of [1]. This definition is reproduced below.

“'*Conditional repetition*'. Let k enumerate the occurrences of $[\![SE]\!]$, then

$[\![\#SE \ \circledast n]\!]$ if and only if $\leadsto \cdots \leadsto$ $[\![SE]\!]^k$ [1, p. 5].

where index variable k satisfies constraint $\circledast\ n$, $\circledast \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$” [1, p. 5].

While the presentation of the definition is relatively opaque, per inspection of the recognizer automaton in Figure 14 from Section 7 of [1], it became apparent that a recognizer for conditional repetition should emit a success signal when the number of success signals received from its operand satisfies the constraint in the expression.

There are several issues with the recognizer automaton specified in Figure 14 of [1] in relation to Definition 13. We describe some of them below.

***Greater than or equal to* repetition constraints implicitly reduce to equality constraints.** For example, if the constraint is specified as ‘`>= 5`’, then since the recognizer returns immediately when the condition is satisfied (i.e. `repetitions` is equal to 5), it is essentially implementing the repetition constraint ‘`= 5`’.

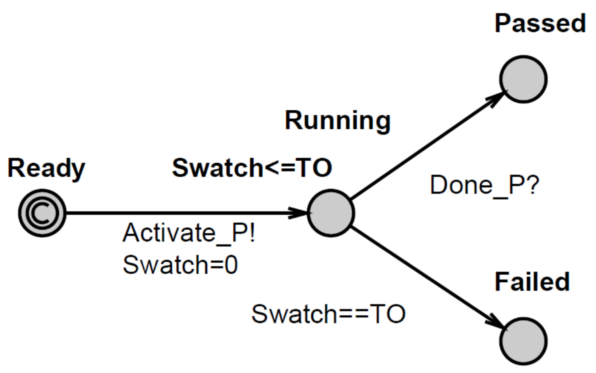
***Greater than* repetition constraints implicitly reduce to equality constraints.** For example, if the constraint is specified as ‘`> 5`’, then since the recognizer returns immediately when the condition is satisfied (i.e. `repetitions` is equal to 6), it is essentially implementing the repetition constraint ‘`= 6`’.

***Less than* & *less than or equal to* constraints are either trivially satisfied or not supported by the architecture of the recognizer tree.** When determining whether some state configuration occurs *less than N* times, the most obvious course of action for the recognizer is to return immediately. This is because the repetition count at the time of activation is 0, so the condition is trivially satisfied. Otherwise, if we alter the recognizer for ‘less than’-constraints so that a success signal is returned if and only if the repetition count occurs less than *N* times during the entire test run, then because of the timeout encoded in the test stopwatch, a failure result is inevitable.

The issues described above should be solved in future work on the theory of TDL^{TP}. As the topic of this thesis is constrained to implementing the logic detailed in the current state of the theory, resolving these issues is out of scope. The recognizer template introduced at the beginning of this appendix is structured in a manner that allows for future extensions.

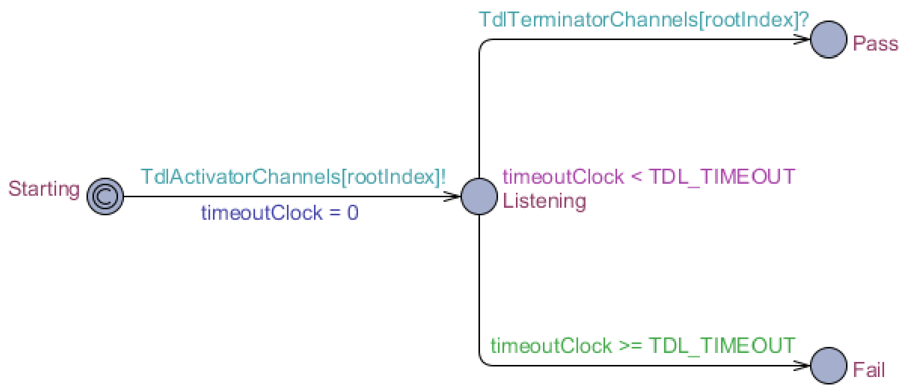
Appendix 7 – Test Stopwatch Template

The test stopwatch automaton is attached to the automaton at the root of the recognizer tree in the test model via broadcast channels. Figure 15 in Section 8 of [1] presents a depiction of this automaton:



Test stopwatch automaton [1, p. 10]

The stopwatch template injected into the test models produced by the TDL^{TP} interpreter is presented below.

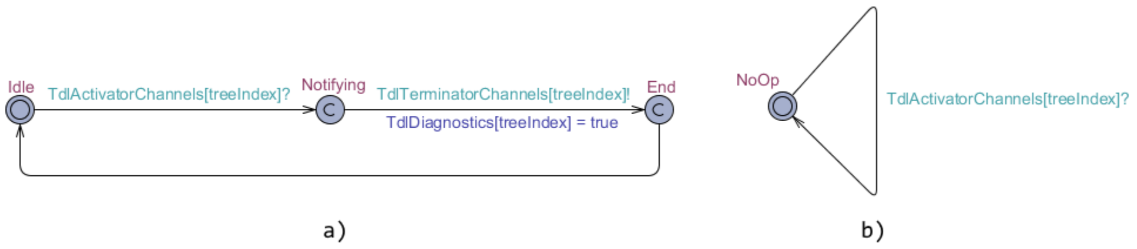

TdlTerminatorChannels[rootIndex]?
Pass
Starting
TdlActivatorChannels[rootIndex]!
timeoutClock = 0
timeoutClock < TDL_TIMEOUT
Listening
timeoutClock >= TDL_TIMEOUT
Fail

Test stopwatch template implementation.

Because the two templates are practically identical, further discussion is not needed.

Appendix 8 – Boolean Literal Recognizer Templates

As described in Section 5.2.2, trapset quantifiers can be replaced with Boolean literals under certain conditions. In order to support future expansions of the TDL^{TP} interpreter, basic recognizers were implemented for these literals. They are presented in the figure below.


Idle
TdlActivatorChannels[treeIndex]?
Notifying
TdlTerminatorChannels[treeIndex]!
TdlDiagnostics[treeIndex] = true
End
a)
NoOp
TdlActivatorChannels[treeIndex]?
b)

Boolean literal recognizer templates: (a) recognizer for *true*; (b) recognizer for *false*.

The semantics embodied by these recognizers is intuitive but trivial. A recognizer for the Boolean true literal returns a success signal immediately after activation. On the other hand, a recognizer for the Boolean false literal never returns a success signal.

Appendix 9 – User Guide

In this appendix we provide instructions for using the TDL^{TP} interpreter.

Prerequisites:

- (*optional*) UPPAAL Academic version 4.0.14;
- Java version 8 runtime.

Installation. The latest release is available in the *Releases* subfolder found in the project's development repository. Links to this repository are provided in Appendix 11.

Execution. The artifact is a Java-based command-line interface. In order to invoke it, the user must provide the root Java binary with a path to the artifact as follows: `java –jar <path to interpreter> <options>`.

The options which the interpreter is capable of accepting are presented below.

Interpreter options.

Short Option	**Long Option**	**Description**
`-e`	`--expression`	*Required.* Test purpose specification as a TDL^{TP} expression. Can be provided as a simple string or as a path to a plain text file.
`-m`	`--model`	*Required.* Path to the input UPPAAL XML model file.
`-o`	`--output`	*Required.* Path where the resultant model is to be stored. If omitted, results will be sent to standard output.
`-t`	`--traces`	*Optional.* Enables the printing of error traces.

Interpreter options (cont'd).

Short Option	**Long Option**	**Description**
`-u`	`--uppaal`	*Optional*. Path to UPPAAL's JAR file. If provided in conjunction with the `-o` option, the resultant model will be opened in UPPAAL after interpretation.
`-v`	`--verbose`	*Optional*. Enables the printing of simple progress messages unless `-o` is omitted.
`-h`	`--help`	Prints instructions.

At the time of writing, the interpreter can parse model files for the latest academic version of UPPAAL (4.0.14) – older versions are not supported.

Appendix 10 – Manual Integration Tests

In this appendix we tabulate the manual integration tests discussed in Section 6.2. The input files for these tests are available in the repository for this project (a link is available in Appendix 11).

The table below lists test cases for the absolute complement trapset operator.

Absolute complement test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-TSExpr-AC-1	`A(!TS)`	Trapset `TS` not mapped to any transitions.	Resultant trapset maps to every transition in test model.
TC-TSExpr-AC-2	`A(!TS)`	Trapset `TS` mapped to every transition.	Expression reduces to true. User receives warning message.
TC-TSExpr-AC-3	`A(!TS)`	Trapset `TS` mapped to at least one transition.	Resultant trapset follows definition of absolute complement.
TC-TSExpr-AC-4	`A(!TS)`	At least one trap in `TS` is conditional.	Resultant trapset contains transitions marked with conditional traps (negated).
TC-TSExpr-AC-5	`A(!TS1) & A(!TS2)`	`TS1` and `TS2` overlap completely.	No error. Rules applied according to subexpressions in conjunction.

The table below lists test cases for the linked pairs trapset operator.

Linked pairs test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-TSExpr-LP-1	`A(TS1; TS2)`	`TS1` maps to no transitions. `TS2` maps to at least one transition.	Reduction to Boolean true. User receives warning message.
TC-TSExpr-LP-2	`A(TS1; TS2)`	`TS1` maps to at least one transition. `TS2` maps to no transitions.	
TC-TSExpr-LP-3	`A(TS1; TS2)`	`TS1` maps to entire model. `TS2` maps to some transitions.	Resultant trapset follows definition of linked pairs.
TC-TSExpr-LP-4	`A(TS1; TS2)`	`TS1` maps to some transitions. `TS2` maps to entire model.	
TC-TSExpr-LP-5	`A(TS1; TS2)`	`TS1` and `TS2` map to some transitions but not all.	Trapset produced in resultant model follows the definition of the linked pairs operation.
TC-TSExpr-LP-6	`A(TS1; TS2)`	`TS1` and `TS2` map to some transitions conditionally.	Conditions are present in test model for both ingress transitions and egress transitions.
TC-TSExpr-LP-7	`A(TS1; TS2)`	`TS1` and `TS2` both map to entire model.	No error. Resultant trapset follows definition of linked pairs.
TC-TSExpr-LP-8	`A(TS1; TS2)`	Duplicate labels exist in model for `TS1` or `TS2`.	An error is reported to the user.
TC-TSExpr-LP-9	`A(TS1; TS2)`	`TS1` and `TS2` map to looping transitions.	Flags are set/reset appropriately on loops.

The following table lists test cases for the relative complement operator.

Relative complement test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-TSExpr-RC-1	`A(TS1 \ TS2)`	`TS1` maps to no transitions. `TS2` maps to at least one transition.	Reduction to Boolean true. User receives warning message.
TC-TSExpr-RC-2	`A(TS1 \ TS2)`	`TS1` maps to at least one transition. `TS2` maps to no transitions.	Output model trapset equivalent to `A(TS1)`.
TC-TSExpr-RC-3	`A(TS1 \ TS2)`	`TS1` maps to entire model. `TS2` maps to some transitions.	Output model trapset equivalent to `A(!TS2)`.
TC-TSExpr-RC-4	`A(TS1 \ TS2)`	`TS1` maps to some transitions. `TS2` maps to entire model.	Reduction to Boolean true. User receives warning message.
TC-TSExpr-RC-5	`A(TS1 \ TS2)`	`TS1` and `TS2` map to some transitions but not all.	Resultant trapset follows definition of relative complement.
TC-TSExpr-RC-6	`A(TS1 \ TS2)`	`TS1` and `TS2` map to some transitions conditionally.	Conditional traps are present in test model (negated).
TC-TSExpr-RC-7	`A(TS1 \ TS2)`	`TS1` and `TS2` both map to entire model.	Reduction to Boolean true. User receives warning message.
TC-TSExpr-RC-8	`A(TS1 \ TS2)`	Duplicate labels exist in model for `TS1` or `TS2`.	Error is reported to user.

The following table lists test cases for trapset quantifiers.

Trapset quantifier test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-TSQuant-UN-1	`A(TS)`	Trapset `TS` not mapped to any transitions.	Reduction to Boolean true. User receives warning message.
TC-TSQuant-UN-2	`A(TS)`	Trapset `TS` mapped to every transition.	Resultant trapset is mapped to every transition.
TC-TSQuant-UN-3	`A(TS)`	Trapset `TS` mapped to at least one transition.	Resultant trapset is equivalent to trapset defined in input model.
TC-TSQuant-EX-1	`E(TS)`	Trapset `TS` not mapped to any transitions.	Reduction to Boolean false. User receives warning message.
TC-TSQuant-EX-2	`E(TS)`	Trapset `TS` mapped to every transition.	Resultant trapset is mapped to every transition.
TC-TSQuant-EX-3	`E(TS)`	Trapset `TS` mapped to at least one transition.	Resultant trapset is equivalent to trapset defined in input model.

The table below lists test cases for the conjunction operator.

Conjunction test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-C-1	`E(TS1) & E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to Boolean false. User receives warning message.
TC-LogOp-C-2	`E(TS2) & E(TS1)`		
TC-LogOp-C-3	`A(TS1) & E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to `E(TS2)`. Trapset `TS1` not present in model.
TC-LogOp-C-4	`E(TS2) & A(TS1)`		
TC-LogOp-C-5	`~(E(TS2) & E(TS1))`	`TS1` maps to some transitions. `TS2` maps to some transitions.	Normalized to `~E(TS2) | ~E(TS1)` per negation replacement rule for conjunction. Applied to test model according to definition of disjunction.
TC-LogOp-C-6	`~(E(TS1) & E(TS2))`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to Boolean true. User receives warning message.
TC-LogOp-C-7	`~(E(TS2) & E(TS1))`		
TC-LogOp-C-8	`~(A(TS1) & E(TS2))`	`TS1` maps to no transitions, `TS2` maps to some transitions.	Reduction to `~E(TS2)`. Trapset `TS1` not present in model.
TC-LogOp-C-9	`~(E(TS2) & A(TS1))`		

The table below lists test cases for the conditional repetition operator.

Conditional repetition test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-CR-1	`#[>=100] E(TS)`	Trapset `TS` is not mapped to any transitions.	Reduction to Boolean false. User receives warning message.
TC-LogOp-CR-2	`#[>=100] A(TS)`		Reduction to Boolean true. User receives warning message.
TC-LogOp-CR-3	`~#[>=100] A(TS)`	Trapset `TS` is mapped to some transitions.	Reduction to Boolean true (because bound condition is negated). User receives warning message.
TC-LogOp-CR-4	`~#[>=100] E(TS)`	Trapset `TS` not mapped to any transitions.	Reduction to Boolean true. User receives warning message.
TC-LogOp-CR-5	`~#[>=100] A(TS)`		Reduction to Boolean false. User receives warning message.
TC-LogOp-CR-6	`#[>=100] A(TS)`	Trapset `TS` is mapped to some transitions.	No error. Test model matches definition of conditional repetition.

The table below lists test cases for the disjunction operator.

Disjunction test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-D-1	`E(TS1) | E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to `E(TS2)`. Trapset `TS1` not present in model.
TC-LogOp-D-2	`E(TS2) | E(TS1)`		
TC-LogOp-D-3	`A(TS1) | E(TS2)`		Reduction to Boolean true. User receives warning message.
TC-LogOp-D-4	`E(TS2) | A(TS1)`		
TC-LogOp-D-5	`~(E(TS2) | E(TS1))`	`TS1` maps to some transitions. `TS2` maps to some transitions.	Normalized to `~E(TS2) & ~E(TS1)` per negation replacement rule for conjunction. Applied to test model according to definition of conjunction.
TC-LogOp-D-6	`~(E(TS1) | E(TS2))`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to `~E(TS2)` and `TS1` trapset not present in model.
TC-LogOp-D-7	`~(E(TS2) | E(TS1))`		
TC-LogOp-D-8	`~(A(TS1) | E(TS2))`		Reduction to Boolean false. User receives warning message.
TC-LogOp-D-9	`~(E(TS2) | A(TS1))`		
TC-LogOp-D-10	`E(TS2) | E(TS1)`	`TS1` and `TS2` map to some transitions.	No error. Test model matches definition of disjunction.

The following table lists test cases for the equivalence operator.

Equivalence test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-E-1	`E(TS1) <=> E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to `~E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-E-2	`E(TS2) <=> E(TS1)`		
TC-LogOp-E-3	`A(TS1) <=> E(TS2)`		Reduction to `E(TS2)` and `TS1` trapset not present in model.
TC-LogOp-E-4	`E(TS2) <=> A(TS1)`		
TC-LogOp-E-5	`~(E(TS2) <=> E(TS1))`	`TS1` maps to some transitions. `TS2` maps to some transitions.	Reduction to `(E(TS2) & ~E(TS1)) | (E(TS1) & ~E(TS2))` and application according to the definitions of disjunction and conjunction.
TC-LogOp-E-6	`~(E(TS1) <=> E(TS2))`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to `E(TS2)` and `TS1` trapset not present in model.
TC-LogOp-E-7	`~(E(TS2) <=> E(TS1))`		
TC-LogOp-E-8	`~(A(TS1) <=> E(TS2))`		Reduction to `~E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-E-9	`~(E(TS2) <=> A(TS1))`		
TC-LogOp-E-10	`E(TS2) <=> E(TS1)`	`TS1` and `TS2` map to some transitions.	Reduction to `(~E(TS2) | E(TS1)) & (~E(TS1) | E(TS2))` and application to input model according to definitions of disjunction and conjunction.

The following table lists test cases for the implication operator.

Implication test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-I-1	`E(TS1) => E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to Boolean true. User receives warning message.
TC-LogOp-I-2	`E(TS2) => E(TS1)`		Reduction to `~E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-I-3	`A(TS1) => E(TS2)`		Reduction to `E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-I-4	`E(TS2) => A(TS1)`		Reduction to Boolean true. User receives warning message.
TC-LogOp-I-5	`~(E(TS2) => E(TS1))`	`TS1` maps to some transitions. `TS2` maps to some transitions.	Reduction to `E(TS2) & ~E(TS1)`. Result applied to input model according to definition of conjunction.
TC-LogOp-I-6	`~(E(TS1) => E(TS2))`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to Boolean false. User receives warning message.
TC-LogOp-I-7	`~(E(TS2) => E(TS1))`		Reduction to `E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-I-8	`~(A(TS1) => E(TS2))`		Reduction to `~E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-I-9	`~(E(TS2) => A(TS1))`		Reduction to Boolean false. User receives warning message.
TC-LogOp-I-10	`E(TS2) => E(TS1)`	`TS1` and `TS2` map to some transitions.	Reduction to `~E(TS2) | E(TS1)`. Applied appropriately.

The following table lists test cases for the leads to operator.

Leads to test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-LT-1	`E(TS1) ~> E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to Boolean false. User receives warning message.
TC-LogOp-LT-2	`E(TS2) ~> E(TS1)`		
TC-LogOp-LT-3	`A(TS1) ~> E(TS2)`		Reduction to `E(TS2)` and trapset `TS1` not present in model.
TC-LogOp-LT-4	`E(TS2) ~> A(TS1)`		
TC-LogOp-LT-5	`~(E(TS2) ~> E(TS1))`	`TS1` maps to some transitions. `TS2` maps to some transitions.	Negation of time-bounded leads to is not supported. User receives error message.
TC-LogOp-LT-6	`~(E(TS1) ~> E(TS2))`	`TS1` maps to no transitions. `TS2` maps to some transitions.	
TC-LogOp-LT-7	`~(E(TS2) ~> E(TS1))`		
TC-LogOp-LT-8	`~(A(TS1) ~> E(TS2))`		
TC-LogOp-LT-9	`~(E(TS2) ~> A(TS1))`		
TC-LogOp-LT-10	`E(TS2) ~> E(TS1)`	`TS1` and `TS2` map to some transitions.	Applied to input model according to definition of leads to.

The following table lists test cases for the bounded leads to operator.

Time-bounded leads to test cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-TBLT-1	`E(TS1) ~>[>=100] E(TS2)`	`TS1` maps to no transitions. `TS2` maps to some transitions.	Reduction to Boolean false. User receives warning message.
TC-LogOp-TBLT-2	`E(TS2) ~>[>=100] E(TS1)`		
TC-LogOp-TBLT-3	`A(TS1) ~>[>=100] E(TS2)`		Reduction to `True ~>[>= 100] E(TS2)`. Applied to model according to definition of bounded leads to.
TC-LogOp-TBLT-4	`E(TS2) ~>[>=100] A(TS1)`		Reduction to `E(TS2) ~>[>= 100] True`. Applied to model according to definition of bounded leads to.
TC-LogOp-TBLT-5	`~(E(TS2) ~>[>=100] E(TS1))`	`TS1` maps to some transitions. `TS2` maps to some transitions.	Negation of time-bounded leads to is not supported. User receives error message.
TC-LogOp-TBLT-6	`~(E(TS1) ~>[>=100] E(TS2))`	`TS1` maps to no transitions. `TS2` maps to some transitions.	
TC-LogOp-TBLT-7	`~(E(TS2) ~>[>=100] E(TS1))`		
TC-LogOp-TBLT-8	`~(A(TS1) ~>[>=100] E(TS2))`		
TC-LogOp-TBLT-9	`~(E(TS2) ~>[>=100] A(TS1))`		
TC-LogOp-TBLT-10	`E(TS2) ~>[>=100] E(TS1)`	`TS1` and `TS2` map to some transitions.	Applied to input model according to definition of time-bounded leads to.

The following table lists symmetrical test cases produced for the discarded subtree corner-case discussed in Section 5.4.

Reduction corner cases.

Identifier	**Input expression**	**Input model**	**Output expectation**
TC-LogOp-X-1	`E(TS2) & ((E(TS1) | E(TS2)) | (E(TS1) & E(TS2)))`	`TS1` is not mapped to any transitions. `TS2` is mapped to some transitions.	Reduction to `E(TS2) & E(TS2)`. Application according to rules for conjunction.
TC-LogOp-X-2	`E(TS2) & ((E(TS1) & E(TS2) | (E(TS1) | E(TS2))))`	`TS1` is not mapped to any transitions. `TS2` is mapped to some transitions.	

As mentioned in Section 6.2, all of these tests were executed against the TDL^{TP} interpreter, thereby verifying a significant portion of its functionality.

Appendix 11 – Repository Links

Source code for the interpreter (and the issue tracking system used during development) is located on Tallinn University of Technology's GitLab instance: https://gitlab.cs.ttu.ee/Tanel.Prikk/iapb.

In case the link above has expired, the repository has been mirrored to GitHub: https://github.com/tanelprikk/ee.taltech.cs.mbt.tdl.interpreter.

For redundancy, the repository has also been mirrored to gitlab.com: https://gitlab.com/tanelprikk/ee.taltech.cs.mbt.tdl.interpreter.