VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AN EFFICIENT FINITE TREE AUTOMATA LIBRARY

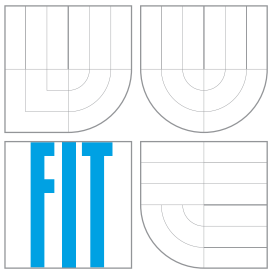
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ONDŘEJ LENGÁL

BRNO 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EFEKTIVNÍ KNIHOVNA PRO PRÁCI S KONEČNÝMI STROMOVÝMI AUTOMATY

AN EFFICIENT FINITE TREE AUTOMATA LIBRARY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ONDŘEJ LENGÁL
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. TOMÁŠ VOJNAR, Ph.D.
SUPERVISOR

BRNO 2010

Abstrakt

Mnoho současných počítačových systémů používá dynamické datové či řídicí struktury předem neomezené velikosti. Tyto datové struktury mají často charakter stromů nebo se dají zakódovat jako stromy s některými dodatečnými ukazateli nad stromovou kostrou. Této skutečnosti využívají některé v současné době intenzivně studované techniky formální verifikace, které reprezentují nekonečně mnoho stavů konečným stromovým automatem. Nicméně v současnosti neexistuje efektivní a flexibilní implementace knihovny pro stromové automaty, která by byla pro tyto techniky vhodná. Cílem této diplomové práce je takovouto knihovnu poskytnout. Předložený text nejdříve popisuje základy teorie konečných stromových automatů a regulárních stromových jazyků. Dále jsou prozkoumány existující implementace knihoven pro stromové automaty a různé verifikační techniky pro systémy se stromovou strukturou. Poté se text zaobírá návrhem reprezentace stromového automatu a algoritmů provádějících standardní jazykové operace nad touto reprezentací, načež následuje popis implementace knihovny. Prostřednictvím provedených experimentů ukazujeme, že knihovna může konkurovat ostatním dostupným knihovnám pro práci se stromovými automaty, přičemž její výkon v určitých oblastech je řádově vyšší.

Abstract

Numerous computer systems use dynamic control and data structures of unbounded size. These data structures have often the character of trees or they can be encoded as trees with some additional pointers. This is exploited by some currently intensively studied techniques of formal verification that represent an infinite number of states using a finite tree automaton. However, currently there is no tree automata library implementation that would provide an efficient and flexible support for such methods. Thus the aim of this Master's Thesis is to provide such a library. The present paper first describes the theoretical background of finite tree automata and regular tree languages. Then it surveys the current implementations of tree automata libraries and studies various verification techniques, outlining requirements for the library. Representation of a finite tree automaton and algorithms that perform standard language operations on this representation are proposed in the next part, which is followed by description of library implementation. Through a series of experiments it is shown that the library can compete with other available tree automata libraries, in certain areas being even significantly superior to them.

Klíčová slova

stromové automaty, formální verifikace, abstraktní regulární stromový model checking, binární rozhodovací diagramy, multiterminálové binární rozhodovací diagramy

Keywords

tree automata, formal verification, abstract regular tree model checking, binary decision diagrams, multi-terminal binary decision diagrams

Citace

Ondřej Lengál: An Efficient Finite Tree Automata Library, diplomová práce, Brno, FIT VUT v Brně, 2010

An Efficient Finite Tree Automata Library

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Tomáše Vojnara, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Lengál
26. května 2010

Poděkování

Děkuji touto cestou vedoucímu práce, doc. Ing. Tomáši Vojnarovi, Ph. D., za odborný přínos, ochotu ke konzultacím, motivaci a trpělivost. Dále bych chtěl poděkovat Mgr. Lukáši Holíkovi, Ing. Jiřímu Šimáčkovi a Mgr. Adamu Rogalewiczovi, Ph. D., za poskytnuté konzultace. Mé díky patří i všem, kteří mě podporovali: Honzovi, Liborovi, Marcelce, Markovi, Petrovi, Martinovi a hlavně mé rodině.

I think that I shall never see
A poem lovely as a tree.
A tree whose hungry mouth is prest
Against the sweet earth's flowing breast;
A tree that looks at God all day,
And lifts her leafy arms to pray;
A tree that may in summer wear
A nest of robins in her hair;
Upon whose bosom snow has lain;
Who intimately lives with rain.
Poems are made by fools like me,
But only God can make a tree.

— Joyce Kilmer, *Trees*

Contents

Chapter 1

Introduction

Donald Knuth, the pioneer of the analysis of algorithms, says that computer scientists love *trees* more than anybody else [1]. Indeed, trees play a crucial role in computer science. They recur in many of its fields, from the representation of programs in the form of abstract syntax trees [2], through the use for fast data retrieval in search trees [3], to tree topologies of computer networks. It is not surprising that trees are often a natural way to represent a model of many types of systems including safety-critical systems

Software errors in safety-critical systems may cause severe losses of money and, in the worst case, even human lives (the Ariane 5 failure is perhaps the best-known case of an expensive software failure [4]). There are several means which help to avoid software bugs in such systems, one of them being verification based on formal mathematical methods, *formal verification.*

Formal verification of computer systems has gained in popularity in recent years. One of the reasons of this increased interest is the fact that testing of systems, which do not need to be very large, can never cover 100 % of cases in an acceptable time (even for systems with finite state spaces; infinite state systems *per se* cannot be completely tested). A popular approach to formal verification is *model checking* (introduced in early 1980s by E. M. Clarke, E. A. Emerson, J. P. Queille and J. Sifakis), a method based on checking whether a given system conforms to given specification by systematically searching the state space of the system. However, in the real world, there exist systems with state spaces that are infinite, though they often have regular structure, e.g. systems with unbounded queues or stacks. As one of the approaches to handle infinite-state systems where states have a linear (or effectively linearizable) structure, *regular model checking* has been proposed [5]. Regular model checking is based on the following ideas: configurations of the systems being verified are represented as finite words over finite alphabet, transitions are represented as relations over words. Then finite (word) automata over the alphabet can be used to represent sets of configurations of the system and finite (word) transducers can be used to express the transition relation.

However, there are also systems that do not have a linear structure which would enable natural encoding of their configuration into finite words. A special case of these are systems with tree-like structure, such as parametrised tree networks or heaps. Moreover, it turns out that many more general graph structures that cannot be easily linearized can be effectively encoded using trees (see e.g. [6]). For such cases, it is convenient to generalise the method to *regular tree model checking* [7], where finite tree automata, a generalisation of finite automata to trees, and finite tree transducers are used.

Nonetheless when used for reachability analysis, regular model checking in general may

suffer from problems with infinite number of configurations as the transducers may generate ever new configurations. Therefore various acceleration techniques that ensure finiteness of the method for many real world problems have been proposed. These methods may need to perform sophisticated operations upon finite tree automata (transducers). In order to conduct the operations in verification of non-trivial systems in an acceptable time, smart data structures and algorithms must be used. However, currently there is no efficient tree automata library that would be suitable for such operations (although MONA, which is discussed in Section 3.2, includes a fairly sophisticated deterministic finite tree automata implementation).

The aim of this work is to design an efficient library that would be suitable for sophisticated tree model checking techniques while being flexible enough to be used even for methods which have not yet been developed. The library focuses on an efficient representation of finite tree automata that work with large alphabets. Unlike most other tree automata libraries, we use symbolic representation to encode transition functions of tree automata. An exception in this sense is MONA which also uses symbolic representation. Moreover, unlike MONA, our library allows to handle *nondeterministic* finite tree automata, which turns out to be crucial for the efficiency of many verification approaches. We have developed a set of algorithms that conduct standard language operations on symbolically represented nondeterministic finite tree automata, as well as algorithms that perform several non-standard operations, such as reduction according to *downward simulation* or inclusion checking based on *antichains*. A prototype of the library has been implemented and evaluated through a series of experiments.

The text is divided into several chapters. Chapter 2 introduces terms, trees, finite tree automata and regular tree languages, while Chapter 3 discusses available libraries that support work with tree automata. In Chapter 4, various formal verification techniques using tree automata are studied and requirements for the library are outlined. Chapter 5 describes the proposed tree automata representation and algorithms for standard operations that work with this representation. This is followed by a description of the implementation of the library in Chapter 6. Chapter 7 gives experimental results. Finally, Chapter 8 summarizes the work and outlines its possible further development.

Chapter 2

Theoretical Background

This chapter introduces standard definitions, which were taken from [8]. Theorems are presented without proofs as they can be found in the same source. First, terms over a ranked alphabet and trees are defined, followed by a description of tree automata and an analysis of closure properties of regular tree languages. Then the concept of tree automata minimisation is introduced, and decision problems for tree automata languages are discussed. Finally, a definition of tree transducers concludes the chapter.

2.1 Terms and Trees

This section introduces terms over ranked alphabet and trees.

2.1.1 Terms

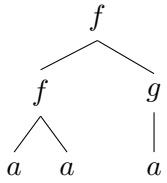
A *ranked alphabet* is a couple $(\mathcal{F}, Arity)$ where $\mathcal{F}$ is a finite set of symbols and $Arity$ is a mapping $Arity : \mathcal{F} \to \mathbb{N}$ ($\mathbb{N}$ denotes the set of non-negative integer numbers). $Arity(f)$, where $f \in \mathcal{F}$, is the *arity* of f. The set of symbols of arity p is denoted by $\mathcal{F}_p$. We assume that the set $\mathcal{F}_0$ (the set of constants) is nonempty. Furthermore, we use parenthesis and commas for a short declaration of symbols with arity, such as $f(,)$ for a binary symbol f.

Let $\mathcal{X}$ be a set of constants called *variables* such that $\mathcal{X} \cap \mathcal{F}_0 = \emptyset$. We define a set of n variables as $\mathcal{X}_n$. The set $T(\mathcal{F}, \mathcal{X})$ of *terms* over the ranked alphabet $\mathcal{F}$ and the set of variables $\mathcal{X}$ is the smallest set defined by:

- $\mathcal{F}_0 \subseteq T(\mathcal{F}, \mathcal{X})$ and
- $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$ and
- if $p \geq 1$, $f \in \mathcal{F}_p$ and $t_1, \ldots, t_p \in T(\mathcal{F}, \mathcal{X})$, then $f(t_1, \ldots, t_p) \in T(\mathcal{F}, \mathcal{X})$.

$T(\mathcal{F}, \emptyset)$ is abbreviated as $T(\mathcal{F})$. Terms in $T(\mathcal{F})$ are called *ground terms*. A term $t \in T(\mathcal{F}, \mathcal{X})$ is *linear* if each variable occurs at most once in t.

Example 1. Let $\mathcal{F} = \{a, g(), f(,)\}$ be a ranked alphabet. The ground term $t = f(f(a, a), g(a))$ can be represented in a graphical way as:



2.1.2 Trees

Let E be a set of labels and $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ be a prefix-closed set. Mapping $f : \mathcal{P}os(t) \to E$ is called a finite ordered *tree* t. A term $t \in T(\mathcal{F}, \mathcal{X})$ can then be viewed as a finite ordered ranked tree with its leaves labeled with variables or constants and its internal nodes labeled with symbols of positive arity, with out-degree equal to the arity of the label. Term $t \in T(\mathcal{F}, \mathcal{X})$ can then be defined as a partial function $t : \mathbb{N}^* \to \mathcal{F} \cup \mathcal{X}$ (with domain $\mathcal{P}os(t)$) that satisfies the following properties:

(i) $\mathcal{P}os(t)$ is nonempty and prefix-closed,

(ii) $\forall p \in \mathcal{P}os(t)$, if $t(p) \in \mathcal{F}_n, n \geq 1$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\}$,

(iii) $\forall p \in \mathcal{P}os(t)$, if $t(p) \in \mathcal{X} \cup \mathcal{F}_0$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \emptyset$.

In the following we confuse trees and terms.

2.1.3 Substitutions

A *substitution* σ is a mapping $\sigma : \mathcal{X} \to T(\mathcal{F}, \mathcal{X})$ (and a *ground substitution* is a mapping $\sigma : \mathcal{X} \to T(\mathcal{F})$) where there are only finitely many variables which are not mapped to themselves. The *domain* of a substitution σ is the subset of variables $x \in \mathcal{X}$ such that $\sigma(x) \neq x$. The substitution $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ is the identity of $\mathcal{X} \setminus \{x_1, \ldots, x_n\}$ and maps $x_i \in \mathcal{X}$ on $t_i \in T(\mathcal{F}, \mathcal{X})$, for every index $1 \leq i \leq n$. The following extends substitutions to $T(\mathcal{F}, \mathcal{X})$:

$$\forall f \in \mathcal{F}_n, \forall t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X}) \qquad \sigma\left(f(t_1, \ldots, t_n)\right) = f\left(\sigma(t_1), \ldots, \sigma(t_n)\right). \tag{2.1}$$

We confuse a substitution and its extension to $T(\mathcal{F}, \mathcal{X})$. Postfix notation is often used for substitutions: $t\sigma$ is the result of applying σ to the term t.

2.1.4 Contexts

A linear term $C \in T(\mathcal{F}, \mathcal{X}_n)$ is called a *context* and the expression $C[t_1, \ldots, t_n]$ for $t_1, \ldots, t_n \in T(\mathcal{F})$ denotes the term in $T(\mathcal{F})$ obtained from C by replacing variable x_i by t_i for each $1 \leq i \leq n$, i.e. $C[t_1, \ldots, t_n] = C\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$. $\mathcal{C}^n(\mathcal{F})$ denotes the set of contexts over $(x_1, \ldots, x_n)$.

Contexts with a single variable are denoted as $\mathcal{C}(\mathcal{F})$. A context is trivial if it is reduced to a variable. Given a context $C \in \mathcal{C}(\mathcal{F})$, we denote the trivial context by C^0, C^1 is equal to C and, for $n > 1$, $C^n = C^{n-1}[C]$ is a context in $\mathcal{C}(\mathcal{F})$.

2.2 Regular Tree Languages and Finite Tree Automata

This section introduces various kinds of finite tree automata.

2.2.1 Nondeterministic Finite Tree Automata

A (bottom-up) *nondeterministic finite tree automaton* (NFTA) over $\mathcal{F}$ is a 4-tuple $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, where Q is a finite set of states ($Q \cap \mathcal{F} = \emptyset$), $Q_f \subseteq Q$ is a set of final states and Δ is a set of transition rules

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)), \tag{2.2}$$

where $n \in \mathbb{N}$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$ and $x_1, \ldots, x_n \in \mathcal{X}$. The *move relation* $\to_{\mathcal{A}}$ is defined by: let $t, t' \in T(\mathcal{F} \cup Q)$,

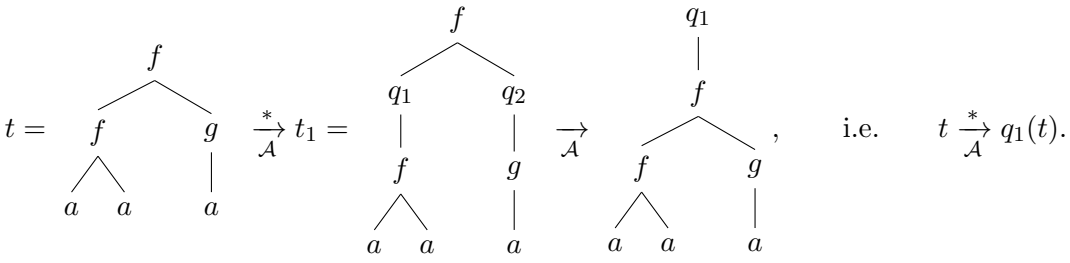
$$t \underset{\mathcal{A}}{\to} t' \Leftrightarrow \begin{cases} \exists C \in \mathcal{C}(\mathcal{F} \cup Q), \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\ \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta, \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))], \\ t' = C[q(f(u_1, \ldots, u_n))]. \end{cases} \tag{2.3}$$

$\to^*_{\mathcal{A}}$ is the reflexive and transitive closure of $\to_{\mathcal{A}}$. A ground term $t \in T(\mathcal{F})$ is *accepted* by an NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ if there exists $q \in Q_f$ such that

$$t \underset{\mathcal{A}}{\overset{*}{\to}} q(t). \tag{2.4}$$

The set of all ground terms accepted by NFTA $\mathcal{A}$ (the *language* of $\mathcal{A}$) is denoted as $\mathcal{L}(\mathcal{A})$. A set $\mathcal{L}$ of ground terms is *regular* if there exists such NFTA $\mathcal{A}$ that $\mathcal{L} = \mathcal{L}(\mathcal{A})$. If two (or more) NFTA accept the same tree language, they are *equivalent*.

Example 2. Consider the ground term $t = f(f(a, a), g(a))$ from Example 1. Let $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be an NFTA and $t_1 \in T(\mathcal{F} \cup Q)$, $t_1 = f(q_1(f(a, a)), q_2(g(a)))$ be a partially processed term t by $\mathcal{A}$, $t \to^*_{\mathcal{A}} t_1$. Assume that $f(q_1(x_1), q_2(x_2)) \to q_1(f(x_1, x_2)) \in \Delta$; then the following sequence of transitions is possible:



If $q_1 \in Q_f$, then $t \in \mathcal{L}(\mathcal{A})$.

An NFTA $\mathcal{A}$ is *complete* if there is at least one rule

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(f(x_1, \ldots, x_n)) \in \Delta \tag{2.5}$$

for all $n \geq 0$, $f \in \mathcal{F}_n$, and $q_1, \ldots, q_n \in Q$. A state $q \in Q$ is *accessible* if there exists a ground term t such that $t \to^*_{\mathcal{A}} q(t)$. An NFTA is *reduced* when all of its states are accessible.

The set of transition rules can also be defined as the set of rules of an alternative form: $f(q_1, \ldots, q_n) \to q$. A move relation can be defined as before, except that instead of preserving the structure of the term, the NFTA $\mathcal{A}$ replaces subtrees with its states. A term t is then accepted by an NFTA $\mathcal{A}$ if

$$t \xrightarrow[\mathcal{A}]{*} q \tag{2.6}$$

where $q \in Q_f$.

2.2.2 Nondeterministic Finite Tree Automata with ε-rules

The definition of *NFTA with ε-rules* is similar to the definition of NFTA, except for the set of transition rules which may now also contain ε-rules of the form $q \to q'$, i.e. the state is changed without processing an input symbol.

Theorem 1. *If $\mathcal{L}$ is accepted by an NFTA with ε-rules, then $\mathcal{L}$ is accepted by an NFTA without ε-rules.*

2.2.3 Deterministic Finite Tree Automata

A *deterministic finite tree automaton* (DFTA) is an NFTA where there are no two rules with the same left-hand side (and no ε-rules) in Δ. It is *unambiguous*, i.e. there is at most one run for every ground term, which means that there is at most one state $q \in Q$ such that $t \to^*_{\mathcal{A}} q$.

Theorem 2. *Let $\mathcal{L}$ be a regular set of ground terms. Then there exists a DFTA that accepts $\mathcal{L}$.*

2.3 Closure properties

2.3.1 Union

Theorem 3. *The class of regular tree languages is closed under union.*

Let us have the following two complete NFTAs (an NFTA can always be made complete by adding missing transitions that all point to a *sink* nonaccepting state): $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$. Now let us construct NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ that accepts $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$, where $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_2 \cup Q_1 \times Q_{f2}$, and $\Delta = \Delta_1 \times \Delta_2$ where

$$\begin{aligned} \Delta_1 \times \Delta_2 \quad = \quad & \{f((q_1, q'_1), \ldots, (q_n, q'_n)) \to (q, q') \mid \\ & f(q_1, \ldots, q_n) \to q \in \Delta_1, f(q'_1, \ldots, q'_n) \to q' \in \Delta_2\}. \end{aligned} \tag{2.7}$$

This construction preserves determinism, i.e. if $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic, then $\mathcal{A}$ is deterministic too.

2.3.2 Complementation

Theorem 4. *The class of regular tree languages is closed under complementation.*

Let $\mathcal{L}(\mathcal{A})$ be a regular tree language and $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ be a complete DFTA. Then an NFTA $\mathcal{A}' = (Q, \mathcal{F}, Q \setminus Q_f, \Delta)$ accepts the complement of set $\mathcal{L}$ in $T(\mathcal{F})$.

2.3.3 Intersection

Theorem 5. *The class of regular tree languages is closed under intersection.*

Closure under intersection follows directly from closure under union and complementation using De Morgan's law:

$$\mathcal{L}_1 \cap \mathcal{L}_2 = \overline{\overline{\mathcal{L}_1} \cup \overline{\mathcal{L}_2}} \tag{2.8}$$

where $\overline{\mathcal{L}}$ denotes the complement of set $\mathcal{L}$ in $T(\mathcal{F})$. The construction that preserves determinism follows: Let $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$ be NFTA. Consider NFTA $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ such that $Q = Q_1 \times Q_2$, $Q_f = Q_{f1} \times Q_{f2}$ and $\Delta = \Delta_1 \times \Delta_2$. It holds that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

2.4 Minimisation of Tree Automata

An equivalence relation $\equiv$ on $T(\mathcal{F})$ is a *congruence* on $T(\mathcal{F})$ if for every $f \in \mathcal{F}_n$

$$u_i \equiv v_i, 1 \leq i \leq n \Rightarrow f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n). \tag{2.9}$$

It is of *finite index* when there are only finitely many $\equiv$-classes. An equivalent definition is that a congruence is an equivalence relation closed under context, i.e. for all contexts $C \in \mathcal{C}(\mathcal{F})$, if $u \equiv v$, then $C[u] \equiv C[v]$. Assume $\mathcal{L}$ is a regular tree language, then $\equiv_{\mathcal{L}}$ on $T(\mathcal{F})$ is defined by: $u \equiv_{\mathcal{L}} v$ if for all contexts $C \in \mathcal{C}(\mathcal{F})$,

$$C[u] \in \mathcal{L} \Leftrightarrow C[v] \in \mathcal{L}. \tag{2.10}$$

Myhill-Nerode Theorem for Tree Languages. *The following three statements are equivalent:*

(i) $\mathcal{L}$ *is a regular tree language,*

(ii) $\mathcal{L}$ *is the union of some equivalence classes of a congruence of finite index,*

(iii) the relation $\equiv_{\mathcal{L}}$ *is a congruence of finite index.*

An interesting point of the proof of the theorem above is the proof of *(iii)* $\Rightarrow$ *(i)*:

Proof. Let Q_{min} be the finite set of equivalence classes of $\equiv_{\mathcal{L}}$. Let us define the transition relation Δ_{min} as the smallest set such that

$$f([u_1], \ldots, [u_n]) \to [f(u_1, \ldots, u_n)] \in \Delta_{min} \tag{2.11}$$

for all $f \in \mathcal{F}$, $u_1, \ldots, u_n \in T(\mathcal{F})$, where $[u]$ denotes the equivalence class of term u. The definition of Δ_{min} is consistent because $\equiv_{\mathcal{L}}$ is a congruence. Let $Q_{min_f} = \{[u] \mid u \in \mathcal{L}\}$. The DFTA $\mathcal{A}_{min} = (Q_{min}, \mathcal{F}, Q_{min_f}, \Delta_{min})$ accepts the tree language $\mathcal{L}$. □

It can be proved that $\mathcal{A}_{min}$ is minimum (in the number of states) and unique up to a renaming of states.

2.5 Top-down Tree Automata

A nondeterministic *top-down* finite tree automaton (top-down NFTA) over $\mathcal{F}$ is a 4-tuple $\mathcal{A} = (Q, \mathcal{F}, I, \Delta)$, where Q is a finite set of states, $I \subseteq Q$ is a set of initial states and Δ is a set of transition rules

$$q(f(x_1, \ldots, x_n)) \to f(q_1(x_1), \ldots, q_n(x_n)), \tag{2.12}$$

where $n \geq 0$, $f \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$, $x_1, \ldots, x_n \in \mathcal{X}$. The move relation is easily deduced from the move relation for bottom-up NFTA.

The tree language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of ground terms t for which there is an initial state $q \in I$ such that

$$q(t) \xrightarrow[\mathcal{A}]{*} t. \tag{2.13}$$

Note that the expressive power of bottom-up and top-down nondeterministic finite tree automata is the same. However, top-down DFTA are strictly less powerful than top-down NFTA.

2.6 Decision Problems and their Complexity

This section summarises some decision problems of regular tree languages and their complexity in the context of RAM machines. Note the increased complexity with respect to regular word languages, which implies an even stronger need for a very careful design and various heuristic optimizations of working with finite tree automata.

- The *fixed membership* problem (determining whether a certain ground term is accepted by a fixed finite tree automaton, i.e. the automaton *is not* the input of the decision procedure) is **ALOGTIME**-complete.
- The *uniform membership* problem (determining whether a certain ground term is accepted by a given finite tree automaton, i.e. the automaton *is* also the input of the decision procedure) can be decided in linear time for DFTA and in polynomial time for NFTA.
- The *emptiness* problem (determining whether the language accepted by given finite tree automaton is empty) is decidable in linear time.
- The *intersection non-emptiness* problem (determining whether there is at least one ground term accepted by each finite tree automaton from a given finite sequence of tree automata) is **EXPTIME**-complete.
- The *finiteness* problem (determining if the language of a given finite tree automaton is finite) is decidable in polynomial time.
- The *complement emptiness* problem (determining whether a given finite tree automaton accepts every ground term) can be decided in polynomial time for DFTA and it is **EXPTIME**-complete for NFTA.
- The *equivalence* problem (determining whether two given finite tree automata accept the same language) is decidable.
- The *singleton set* problem (determining whether a given finite tree automaton accepts only a single ground term) is decidable in polynomial time.

2.7 Tree Transducers

2.7.1 Bottom-up Tree Transducers

A *nondeterministic bottom-up tree transducer* (NBUTT) is a 5-tuple $U = (Q, \mathcal{F}, \mathcal{F}', Q_f, \Delta)$, where Q is a set of states ($Q \cap \mathcal{F} = \emptyset$, $Q \cap \mathcal{F}' = \emptyset$), $\mathcal{F}$ and $\mathcal{F}'$ are finite nonempty sets of input symbols and output symbols, $Q_f \subseteq Q$ is a set of final states and Δ is a set of transduction rules of the following two types:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(u), \tag{2.14}$$

where $f \in \mathcal{F}_n$, $u \in T(\mathcal{F}', \mathcal{X}_n)$, $q, q_1, \ldots, q_n \in Q$, and $x_1, \ldots, x_n \in \mathcal{X}_n$, and

$$q(x_1) \to q'(u) \qquad (\varepsilon\text{-rule}), \tag{2.15}$$

where $u \in T(\mathcal{F}', \mathcal{X}_1)$, $q, q' \in Q$, and $x_1 \in \mathcal{X}_1$.

Let $t, t' \in T(\mathcal{F} \cup \mathcal{F}' \cup Q)$. The move relation $\to_U$ is defined as:

$$t \underset{U}{\to} t' \Leftrightarrow \begin{cases} \exists f(q_1(x_1), \ldots, q_n(x_n)) \to q(u) \in \Delta, \\ \exists C \in \mathcal{C}(\mathcal{F} \cup \mathcal{F}' \cup Q), \\ \exists u_1, \ldots, u_n \in T(\mathcal{F}'), \\ t = C[f(q_1(u_1), \ldots, q_n(u_n))], \\ t' = C[q(u\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\})]. \end{cases} \tag{2.16}$$

The reflexive and transitive closure of $\to_U$ is $\to_U^*$. The relation induced by U (also denoted as U) is:

$$U = \left\{(t, t') \mid t \to_U^* q(t'), t \in T(\mathcal{F}), t' \in T(\mathcal{F}'), q \in Q_f\right\}. \tag{2.17}$$

A transducer is *ε-free* if there is no ε-rule in Δ. If all transduction rules are linear (no variable occurs twice in the right-hand side), then the transducer is *linear*. It is *nonerasing* if, for each rule, at least one symbol from $\mathcal{F}'$ occurs in the right-hand side. In a *complete* (or *non-deleting*) transducer, for every rule $f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$, for every $x_i, (1 \leq i \leq n)$, x_i occurs at least once in u. An ε-free transducer where there are no two rules with the same left-hand side is called *deterministic* (DBUTT).

2.7.2 Top-down Tree Transducers

A *nondeterministic top-down tree transducer* (NTDTT) is a 5-tuple $D = (Q, \mathcal{F}, \mathcal{F}', Q_i, \Delta)$, where Q is a set of states ($Q \cap \mathcal{F} = \emptyset$, $Q \cap \mathcal{F}' = \emptyset$), $\mathcal{F}$ and $\mathcal{F}'$ are finite nonempty sets of input and output symbols, $Q_i \subseteq Q$ is a set of initial states and Δ is a set of transduction rules of the following two types:

$$q(f(x_1, \ldots, x_n)) \to u[q_1(x_{i_1}), \ldots, q_p(x_{i_p})], \tag{2.18}$$

where $f \in \mathcal{F}_n$, $u \in \mathcal{C}^p(\mathcal{F}')$, $q, q_1, \ldots, q_p \in Q$, $x_{i_1}, \ldots, x_{i_p} \in \mathcal{X}_n$, and

$$q(x) \to u[q_1(x), \ldots, q_p(x)] \qquad (\varepsilon\text{-rule}), \tag{2.19}$$

where $u \in \mathcal{C}^p(\mathcal{F}')$, $q, q_1, \ldots, q_p \in Q$, $x \in \mathcal{X}$.

Let $t, t' \in T(\mathcal{F} \cup \mathcal{F}' \cup Q)$. The move relation $\rightarrow_D$ is defined as:

$$t \underset{D}{\longrightarrow} t' \Leftrightarrow \begin{cases} \exists q(f(x_1, \ldots, x_n)) \rightarrow u[q_1(x_{i_1}), \ldots, q_p(x_{i_p})] \in \Delta, \\ \exists C \in \mathcal{C}(\mathcal{F} \cup \mathcal{F}' \cup Q), \\ \exists u_1, \ldots, u_n \in T(\mathcal{F}), \\ t = C[q(f(u_1, \ldots, u_n))], \\ t' = C[u[q_1(v_1), \ldots, q_p(v_p)]] \qquad \text{where } v_j = u_k \text{ if } x_{i_j} = x_k. \end{cases} \tag{2.20}$$

The reflexive and transitive closure of $\rightarrow_D$ is $\rightarrow_D^*$. The relation induced by D (also denoted as D) is:

$$D = \left\{ (t, t') \mid q(t) \rightarrow_D^* t', t \in T(\mathcal{F}), t' \in T(\mathcal{F}'), q \in Q_i \right\}. \tag{2.21}$$

ε-free, linear, non-erasing, complete, deterministic (DTDTT) top-down tree transducers are the same as in the bottom-up case.

Chapter 3

Existing Tree Automata Libraries

This chapter describes several implementations of finite tree automata libraries focusing on a couple of the most interesting from our point of view: Timbuk and MONA.

3.1 Timbuk

Timbuk [9] is a collection of tools for achieving proofs of reachability over *term rewriting systems* and for manipulating tree automata. This system is written in OCaml, a popular functional programming language. Version 2.2 of Timbuk was surveyed; although newer version 3.0 is currently available, this version has abandoned the tree automata library present in earlier versions as the tool now focuses on reachability analysis and equational approximations of term rewriting systems. This library is a free software (available under the GNU LGPLv2 [10] licence) distributed for free, therefore it was possible to study the implementation.

The tree automaton is implemented as a tuple of lists: a list of symbols (an alphabet), a list of state operators, a list of states, a list of final states, a list of transitions and a list of prioritary transitions. The supported operations on tree automata are the standard ones: intersection, union, language emptiness, deletion of inaccessible states, determinisation and others. Since states and transitions are represented as lists, the aforementioned operations are implemented in a straightforward way. The library is able to construct a tree automaton directly from a given term rewriting system.

3.2 MONA

MONA [11] is a tool (released free of charge under the GNU GPLv2 licence [12]) that implements decision procedures for the *weak second-order theory of one or two successors* (WS1S/WS2S). These types of logic are notable for the following reasons:

WS1S Büchi claims in [13] that WS1S has an expressive power equivalent to regular expressions, i.e. it can be used to denote the class of regular languages.

WS2S According to [14] (who further refers to Thatcher and Wright [15]), WS2S is equivalent to the class of regular tree languages.

Indeed, MONA uses finite automata and finite tree automata for determining the truth status of formulae in WS1S and WS2S, respectively. Independently of MONA, Glenn and Gasarch [16] also implemented an automaton-based decision procedure for WS1S.

MONA was actively developed for six years, but since 2002 no further progress of the tool has appeared and only bugfixes have been applied. However, Klarlund *et al* [17] boldly claim that the developers of MONA tried many approaches to deal with common problems and tuned the tool to give the best performance. The most important feature mentioned is symbolic representation of transition functions of automata by *multi-terminal binary decision diagrams* (MTBDD), which are a generalisation of *reduced ordered binary decision diagrams* (ROBDD, often abbreviated just as BDD, see [18] for further details). The idea of generalising BDDs to MTBDDs is by assigning multiple values to the sink nodes of the diagram (i.e. generalising function f represented by BDD, $f : \{0,1\}^n \to \{0,1\}$, to function g represented by MTBDD, $g : \{0,1\}^n \to \mathbb{D}$, where $\mathbb{D}$ is an arbitrary domain such that it contains *bottom* element $\bot \in \mathbb{D}$).

Due to the fact that BDDs are only a compact representation of formulae in propositional logic with Boolean variables $x_1, \ldots, x_n$, Boolean formulae can be used for their description. A BDD $f : \{0,1\}^n \to \{0,1\}$ maps to the Boolean formula

$$\sum_{(a_1,\ldots,a_n)\in\{0,1\}^n} \left(\prod_{a_i=0} \neg x_i \cdot \prod_{a_i=1} x_i \cdot f(a_1, \ldots, a_n) \right) . \tag{3.1}$$

The mapping for MTBDDs is analogous, however a few preconditions need to be imposed on domain $\mathbb{D}$:

(i) the product of $x \in \{0,1\}$ and $d \in \mathbb{D}$ is defined as

$$x \cdot d \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if} \quad x = 0 \\ d & \text{if} \quad x = 1 \end{cases} , \tag{3.2}$$

(ii) the addition operation on $\mathbb{D}$ needs to ensure that for $d \in \mathbb{D}$ it holds that

$$d + \bot = \bot + d = d \ . \tag{3.3}$$

Then we define the mapping from MTBDD $g : \{0,1\}^n \to \mathbb{D}$ to the Boolean formula

$$\sum_{(a_1,\ldots,a_n)\in\{0,1\}^n} \left(\prod_{a_i=0} \neg x_i \cdot \prod_{a_i=1} x_i \cdot g(a_1, \ldots, a_n) \right) . \tag{3.4}$$

Example 3. This example shows in Figure 3.1 the structure of the following decision diagrams:

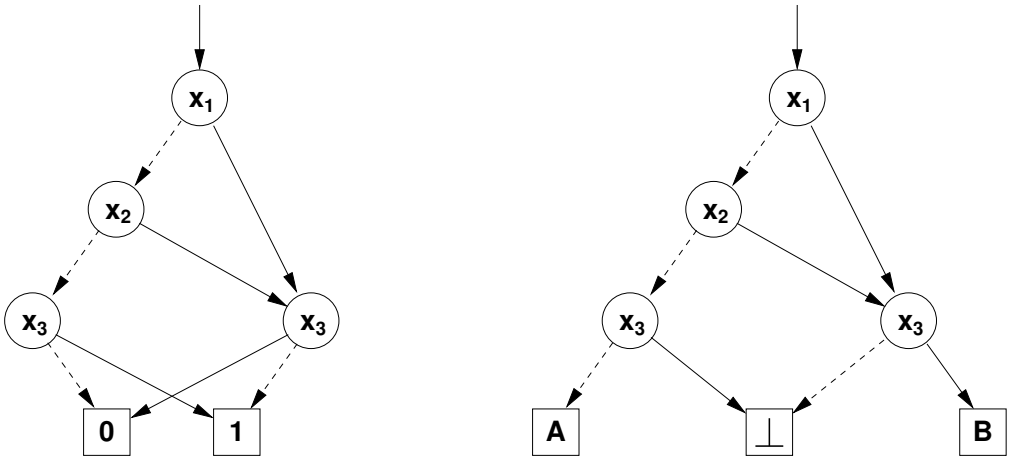
a) a BDD representing formula:

$$x_1 \neg x_3 + \neg x_1 x_2 \neg x_3 + \neg x_1 \neg x_2 x_3 \ , \tag{3.5}$$

b) an MTBDD representing formula:

$$\neg x_1 \neg x_2 \neg x_3 A + x_1 x_3 B + \neg x_1 x_2 x_3 B \ . \tag{3.6}$$

Note that, e.g., the expression $x_1 x_3$ represents the expression $x_1(x_2 + \neg x_2)x_3$ which fully expands to $x_1 x_2 x_3 + x_1 \neg x_2 x_3$.



a) BDD b) MTBDD

Figure 3.1: Examples of the structure of BDD and MTBDD.

When MTBDDs are used for transition table representation, every symbol of the input alphabet Σ is assigned a binary string, i.e. there exists an encoding function $enc : \Sigma \rightarrow \{0,1\}^n$, where $n = \lceil \lg |\Sigma| \rceil$. The values of sink nodes (set $\mathbb{D}$ from previously mentioned function g) is the state set of the automaton. Such an MTBDD can be used to describe the transition relation of the automaton for a single state: the input symbol is encoded by function *enc* into a sequence of binary digits $(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ correspond to the Boolean variables of the MTBDD. The assignment to the variables denotes the path that is to be taken in the diagram and determines the sink node (i.e. the next state of the automaton). Such an MTBDD may either exist for every state of the automaton, or preferably *shared* MTBDD is used. This is another generalisation which merges all diagrams into a single one with multiple root nodes (each corresponding to a different state) and changes the tree-like structure of an MTBDD into a *directed acyclic graph* (DAG). This solution yields a compact representation of the transition function even for large input alphabets.

Another concept introduced by MONA developers is *guided tree automaton* [19], which is supposed to tackle state space blow-up. Bottom-up tree automata often suffer from the problems of the way they work: while the automaton traverses the tree from its leaves to the root, it does not have any information about the position in the tree. The guided tree automaton provides a *guide*, an additional top-down tree automaton that labels tree nodes by assigning state spaces to them, making them aware of their position in the tree. This assignment is done before the actual automaton starts working. When it does, it operates faster, since every state space has its own state set and transition table. The guide needs to be either provided by the programmer, or it can be synthesized automatically for certain domains (e.g. WSRT logic used for description of recursive data types, which is implemented in MONA).

Another noteworthy optimization applied in MONA is so-called *eager minimisation*: whenever the structure of an automaton is modified, a Myhill-Nerode minimisation is performed. Although originally not expected, this strategy yields very good results. Despite the fact that formulae are often represented in the form of trees (at least during syntactic analysis of the formula), MONA uses DAGs for their representation. Common subexpressions are identified and collapsed, thus saving both space and time.

Although MONA only supports work with deterministic finite (tree) automata, there are formal verification techniques (such as [20]) that can efficiently work directly with nondeterministic finite (tree) automata, thus avoiding possible time and space exponential blow-up caused by automata determinisation. After the construction of a finite tree automaton, MONA tries to find both a satisfying example and a counterexample. Therefore there is no efficient support for sophisticated manipulation with automata which may be required by some verification methods.

3.3 Other Libraries

Java library Lethal [21] supports numerous operations on tree automata, like checking whether some properties (determinism, completeness, ...) hold for a given automaton, or standard operations on languages (such as union, intersection, complement or difference). The implementation appears to be quite naïve, with a primary focus on education. However, as the only studied library, Lethal also implements tree transducers and hedge automata (a modification of tree automata for unranked trees).

Binary Tree Automata Library [22] is a Caml library for tree automata. The implementation provides only basic functions and is close to Timbuk (see section 3.1), although it uses hash tables for a transition table representation and language-provided sets for state sets.

A simple implementation of a tree automata library in ELAN can be found in [23]. Also this library provides only basic functionality.

Chapter 4

Analysis

This chapter starts with an introduction to abstract regular tree model checking and an analysis of potential use of the library and collects requirements for the library.

4.1 Abstract Regular Tree Model Checking

The basic idea of *regular tree model checking* is to decide the emptiness of the language

$$\tau^* \left(\mathcal{L}(\mathit{Init})\right) \cap \mathcal{L}(\mathit{Bad}), \tag{4.1}$$

where *Init* is a tree automaton denoting the set of initial states of the system, *Bad* is a tree automaton expressing the set of states violating the safety properties of the system, and τ is a linear tree transducer representing the transition relation of the system. Because an iterative computation of $\tau^* \left(\mathcal{L}(\mathrm{Init})\right)$ may not terminate, several acceleration methods have been proposed. One of them is *abstract regular tree model checking* [24], which is an acceleration technique based on the *abstract-check-refine* paradigm. *Abstraction* α is a function from the set of all tree automata $\mathcal{M}_\mathcal{F}$ over ranked alphabet $\mathcal{F}$ to its subset $\mathcal{A}_\mathcal{F}$, $\mathcal{A}_\mathcal{F} \subseteq \mathcal{M}_\mathcal{F}$, such that $\forall M \in \mathcal{M}_\mathcal{F} : \mathcal{L}(M) \subseteq \mathcal{L}(\alpha(M))$.

4.1.1 Abstraction Based on Languages of Finite Height

Abstraction based on languages of finite height, which was introduced in [24], defines two states of a tree automaton as equivalent if their languages up to a given height n are identical. The implementation can be done similar to the Myhill-Nerode minimisation, except that the procedure stops after n iterations.

4.1.2 Abstraction Based on Predicate Languages

Given a set of *predicate* tree automata $\mathcal{P} = (P_1, \ldots, P_n)$, *abstraction based on predicate languages* (introduced also in [24]) defines two states of a tree automaton equivalent if their languages have a nonempty intersection with exactly the same subset of languages represented by tree automata from $\mathcal{P}$. This can be done by labeling every state with predicates that have a nonempty intersection with the language of the automaton and collapsing states with identical labeling.

4.2 Requirements

An important requirement for the library is to enable a direct work with nondeterministic tree automata without determinising the automaton first. This is convenient for avoiding state explosion connected with automaton determinisation in some verification techniques (see [20]). The following standard operations are necessary to be implemented in the library:

- creating a finite tree automaton denoting the union of languages of given finite tree automata,
- creating a finite tree automaton denoting the intersection of languages of given finite tree automata,
- creating a finite tree automaton denoting the complement of the language of a given finite tree automaton,
- determinisation of a finite tree automaton,
- minimisation of a finite tree automaton,
- determining emptiness of language of a finite tree automaton,
- reducing the size of a given nondeterministic finite tree automaton without determinisation, and
- determining inclusion of languages of given finite tree automata while avoiding determinisation of any automaton.

The library also needs to implement tree transducers at least in their *structure-preserving* form. Certain techniques [24, 25, 26, 27, 28] need to efficiently traverse all states of the automaton in order to, for instance, compute the abstraction of the automaton. Support for this is also necessary.

Chapter 5

Design

This chapter starts with a description of the representation that we propose for the transition function of nondeterministic finite tree automata. This is followed by a description of algorithms for operations on finite tree automata that use this representation. Relabelling tree transducers and operations with them are described at the end of the chapter.

5.1 Representation of a Finite Tree Automaton

The performance of operations on a nondeterministic finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is mostly affected by the choice of the data structure for representing the transition function Δ. Two major approaches are possible:

Explicit representation This approach represents the transition function of a tree automaton by enumerating all transitions in a data structure used for a representation of the set.

Symbolic representation This method is a popular approach in model checking that is based on a representation of the transition function using Boolean formulae. The exact form of the representation varies depending on the application, however the most popular data structure used for representing Boolean formulae is the BDD.

The analysis in Chapter 3 showed that *symbolic representation* using MTBDDs is a very promising approach. Therefore we chose MTBDDs for representation of transition function. To recap, MTBDD is a data structure that stores mapping $g : \{0,1\}^n \to \mathbb{D}$, where $\mathbb{D}$ is an arbitrary set.

Our design attempts to tackle the problem of large alphabets by using a shared MTBDD such that the domain of the MTBDD, i.e. the sequence of Boolean variables $\{0,1\}^n$, represents binary encodings of symbols from $\mathcal{F}$ according to some encoding function $enc : \mathcal{F} \to \{0,1\}^n$, where $n \leq \lceil \lg |\mathcal{F}| \rceil$ (note that n may be smaller than $\lceil \lg |\mathcal{F}| \rceil$ because when arity of symbols is implicit, more symbols with different arity may map to one assignment of Boolean variables; in conflicting cases, we will denote symbol $f \in \mathcal{F}_p$ as f_p). Using function enc to encode symbols from $\mathcal{F}$, MTBDD may represent function $g : \mathcal{F} \to \mathbb{D}$. Before we proceed, let us first define the set of *super-states* $S(\Delta)$ of the transition function Δ as

$$\begin{aligned} S(\Delta) \quad = \quad & \{(q_1, \ldots, q_p) \mid p \geq 0, \\ & f(q_1, \ldots, q_p) \to D \in \Delta, f \in \mathcal{F}_p, D \subseteq Q, D \neq \emptyset\} \end{aligned} \tag{5.1}$$

or in case of a complete automaton with a sink state q_{sink}:

$$\begin{aligned} S(\Delta) \quad = \quad & \{(q_1, \ldots, q_p) \mid p \geq 0, \\ & f(q_1, \ldots, q_p) \to D \in \Delta, f \in \mathcal{F}_p, D \subseteq Q, D \neq \{q_{sink}\}\}\,. \end{aligned} \tag{5.2}$$

Let $S_n(\Delta)$ be the set of super-states of Δ of arity n. Note that an empty sequence () represents *initial super-state*, i.e. the super-state from which transitions over leaf nodes are possible. We also extend the definition of membership relation $\in$ to super-states in the following way:

$$q \in (q_1, \ldots, q_n) \overset{\text{def}}{\Leftrightarrow} \exists 1 \leq i \leq n : q = q_i \tag{5.3}$$

Using the previous definition of super-states and definition of Δ (see Equation 2.2), we may alternatively define the transition function of an automaton $\mathcal{A}$ as a mapping $\Delta^\bullet$ in the following way:

$$\begin{aligned} \Delta^\bullet \quad : \quad & S \to (\mathcal{F} \to 2^Q) \\ & (q_1, \ldots, q_p) \mapsto \{(f, D) \mid f(q_1, \ldots, q_p) \to D \in \Delta\} \end{aligned} \tag{5.4}$$

This means that we may represent the transition function Δ of a tree automaton $\mathcal{A}$ as a data structure that associates each super-state with an MTBDD that is indexed using a binary encoding of symbols from $\mathcal{F}$ and has subsets of Q in its sink nodes. When shared MTBDD is used, each super-state is mapped to a root of a given MTBDD. In case $\Delta^\bullet$ is not *total*, we make it total by assigning MTBDD where all symbols map to a sink state $\{q_{sink}\}$ to each super-state that has no image in $\Delta^\bullet$, which yields a complete automaton. We further confuse Δ and $\Delta^\bullet$.

Example 4. Consider the nondeterministic finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, $Q = \{q_1, q_2, q_3\}$, $\mathcal{F} = \{a, b_0, b_2, c_0, c_1, d_1\}$, and $\Delta = \{b_0 \to \{q_1, q_2\}, c_0 \to \{q_2\}, d_1(q_2) \to \{q_3\}, b_2(q_1, q_3) \to \{q_1, q_2\}, c_1(q_3) \to \{q_1, q_2\}\}$ (Q_f is not important at this point). A shared MTBDD corresponding to Δ is in Figure 5.1.

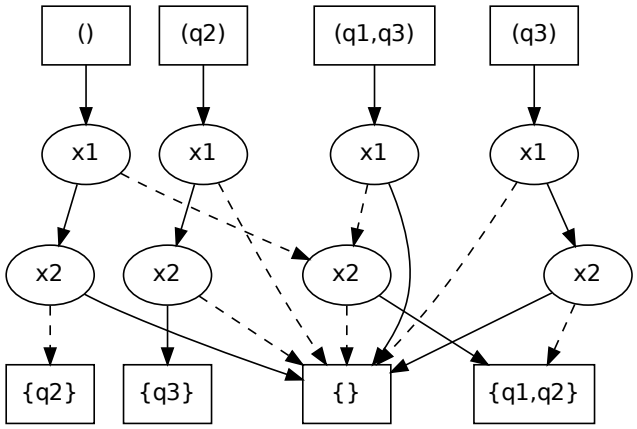


Figure 5.1: A representation of Δ by a shared MTBDD. Encoding of symbols from $\mathcal{F}$: a: `00`, b: `01`, c: `10`, d: `11`. Dashed lines represent `0` value of given variable, solid lines represent value `1`.

5.2 Operations on MTBDDs

This section describes algorithms that perform operations on finite tree automata with a transition function represented by an MTBDD. These algorithms manipulate MTBDDs using the following two sufficient functions:

Apply The standard *Apply* function that performs a given binary operation `op` on all respective sink nodes (this means sink nodes accessible over the same symbol) of two input MTBDDs (`lhs` and `rhs` for left-hand side MTBDD and right-hand side MTBDD respectively) and returns the resulting MTBDD.

$$\begin{aligned} \mathit{Apply} \quad &: \quad (\mathcal{F} \to 2^Q) \to (\mathcal{F} \to 2^Q) \to (2^Q \to 2^Q \to 2^Q) \to (\mathcal{F} \to 2^Q) \\ &\quad \mathit{Apply}\ \texttt{lhs}\ \texttt{rhs}\ \texttt{op} = \lambda x\ .\ \texttt{op}\ (\texttt{lhs}\ x)\ (\texttt{rhs}\ x) \end{aligned} \tag{5.5}$$

MonadicApply The monadic version of *Apply* function that performs a given unary operation `op` on all sink nodes of input MTBDD `tf` and returns the resulting MTBDD.

$$\begin{aligned} \mathit{MonadicApply} \quad &: \quad (\mathcal{F} \to 2^Q) \to (2^Q \to 2^Q) \to (\mathcal{F} \to 2^Q) \\ &\quad \mathit{MonadicApply}\ \texttt{tf}\ \texttt{op} = \lambda x\ .\ \texttt{op}\ (\texttt{tf}\ x) \end{aligned} \tag{5.6}$$

Note that λ-calculus is used for definitions and applications of functions that work with MTBDDs in order to make them more comprehensible. In case the result of the *Apply* operation is not stored (the operation is performed solely for the side effect of `op`), `op` does not need to return a value. Further, we assume that transition functions for all automata are stored in a single shared MTBDD.

5.2.1 Insertion of a Transition

Inserting a transition into an MTBDD-represented transition function of finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is done by creating a new MTBDD with only given transition and merging it with the original MTBDD representing Δ by substituting the sink node at position given by the symbol of the transition with the new value as described in Algorithm 1. In order to create a new MTBDD with a given transition, an additional function is necessary:

CreateMTBDD This function creates an MTBDD which maps a single symbol to a single set of states.

$$\begin{aligned} \mathit{CreateMTBDD} \quad &: \quad \mathcal{F} \to 2^Q \to (\mathcal{F} \to 2^Q) \\ &\quad \mathit{CreateMTBDD}\ \texttt{k}\ \texttt{D} = \lambda x\ .\ \textbf{if}\ x = \texttt{k}\ \textbf{then}\ \texttt{D}\ \textbf{else}\ \{q_{sink}\} \end{aligned} \tag{5.7}$$

Algorithm 1: Transition insertion

Input: Transition function Δ_{IN}
Transition $f(q_1, \ldots, q_n) \to D$ to be inserted
Output: $\Delta_{OUT} = (\Delta_{IN} \setminus \{f(q_1, \ldots, q_n) \to E \mid E \subseteq Q\}) \cup \{f(q_1, \ldots, q_n) \to D\}$

```
1 begin
2     tmp := CreateMTBDD f D;
3     sp := (q_1, ..., q_n);
4     Δ_OUT := Δ_IN;
5     Δ_OUT sp := Apply (Δ_IN sp) tmp (λX Y . if Y = {q_sink} then X else Y);
6     return Δ_OUT;
7 end
```

5.2.2 Retrieval of a Transition

The algorithm that *retrieves a transition* (i.e. for a given super-state $(q_1, \ldots, q_n)$ and a symbol f returns D such that $f(q_1, \ldots, q_n) \to D \in \Delta$) from an MTBDD-represented transition function Δ first creates a *projection BDD* and makes a projection of the MTBDD representing the transition function for given super-state according to given symbol of the input alphabet. A projection BDD $p : \mathcal{F} \to \{0, 1\}$ is a BDD over the same set of Boolean variables as the transition function MTBDD which identifies the nodes that are to be excluded from the MTBDD with value 0 and the others with value 1. After the projection is done, *MonadicApply* collects the sink nodes of the resulting MTBDD. The algorithm, which is described in Algorithm 2, needs the following two additional functions for working with projection BDDs:

CreateProjection This function creates a projection BDD for symbol k.

$$\begin{aligned} \mathit{CreateProjection} \quad &: \quad \mathcal{F} \to (\mathcal{F} \to \{0,1\}) \\ &\quad\; \mathit{CreateProjection}\ \mathtt{k} = \lambda x\ .\ \textbf{if}\ x = \mathtt{k}\ \textbf{then}\ 1\ \textbf{else}\ 0 \end{aligned} \tag{5.8}$$

Project Makes a projection of MTBDD `lhs` using a projection BDD `rhs` and returns the resulting MTBDD.

$$\begin{aligned} \mathit{Project} \quad &: \quad (\mathcal{F} \to 2^Q) \to (\mathcal{F} \to \{0,1\}) \to (\mathcal{F} \to 2^Q) \\ &\quad\; \mathit{Project}\ \mathtt{lhs}\ \mathtt{rhs} = \lambda x\ .\ \textbf{if}\ (\mathtt{rhs}\ x) = 1\ \textbf{then}\ (\mathtt{lhs}\ x)\ \textbf{else}\ \{q_{sink}\} \end{aligned} \tag{5.9}$$

Algorithm 2: Transition retrieval

Input: Transition function Δ
Symbol f and super-state $(q_1, \ldots, q_n)$
Output: $D = \{E \mid f(q_1, \ldots, q_n) \to E \in \Delta\}$

```
1 begin
2     states := ∅;
3     tmp := CreateProjection f;
4     proj := Project (Δ (q_1, ..., q_n)) tmp;
5     MonadicApply proj (collect states);
6     return states;
7 end
```

Function `collect(`*states, leaf*`)`

```
1 begin
2     states := states ∪ leaf;
3 end
```

5.2.3 Language Union

The task of the operation of *language union* is, for two input tree automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$, to create a tree automaton $\mathcal{A}_\cup = (Q_\cup, \mathcal{F}, Q_{f\cup}, \Delta_\cup)$ such that $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$. Although the algorithm presented in Section 2.3.1 preserves determinism, we chose to use a more simple approach that does not create a product automaton but rather reuses transition functions of input automata as much as possible (and may introduce nondeterminism when input automata are deterministic).

The idea of this construction is to create such an automaton that makes nondeterministic transitions over leaf symbols to either $\mathcal{A}_1$ or $\mathcal{A}_2$ and then continues its run in the target automaton. Assume without loss of generality that $Q_1 \cap Q_2 = \emptyset$, then $Q_\cup = Q_1 \cup Q_2$, $Q_{f\cup} = Q_{f1} \cup Q_{f2}$, and

$$\begin{aligned} \Delta_\cup \;=\; & (\Delta_1 \setminus \{f \to D_1 \mid f \in \mathcal{F}_0, D_1 \subseteq Q_1\}) \cup \\ & (\Delta_2 \setminus \{f \to D_2 \mid f \in \mathcal{F}_0, D_2 \subseteq Q_2\}) \cup \\ & \{f \to D \mid f \in \mathcal{F}_0, D = D_1 \cup D_2, f \to D_1 \in \Delta_1, f \to D_2 \in \Delta_2\}\,. \end{aligned} \tag{5.10}$$

Computations of $Q_\cup$ and $Q_{f\cup}$ are trivial. Since all transitions are stored in a single MTBDD the computation of $\Delta_\cup$ needs only one *Apply* operation:

$$\Delta_\cup\ ()\ :=\ \mathit{Apply}\ (\Delta_1\ ())\ (\Delta_2\ ())\ (\lambda X\ Y\ .\ X \cup Y)\,. \tag{5.11}$$

Figure 5.2 shows the process of construction of the transition function for the union automaton. The procedure is described in Algorithm 3.

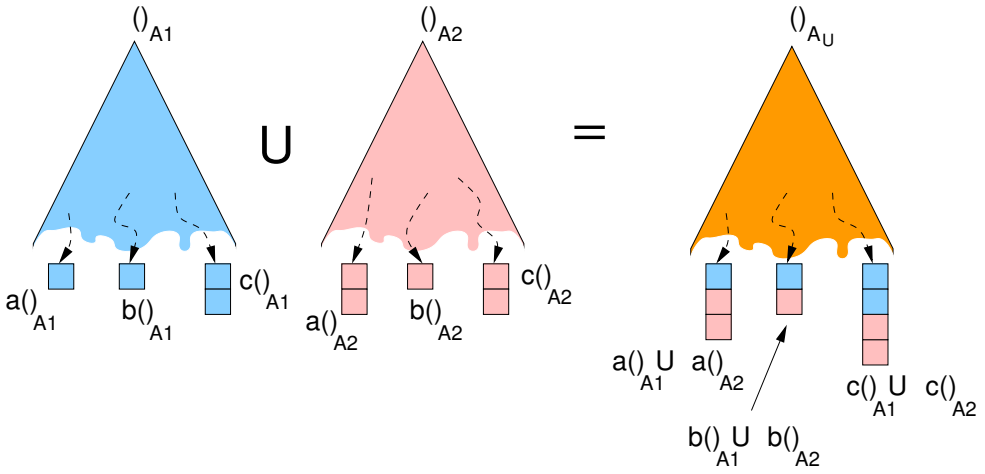


Figure 5.2: Construction of automaton $\mathcal{A}_\cup$ such that $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

5.2.4 Language Intersection

The requirements on the *language intersection* operation are very similar to language union: given two finite tree automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$ construct a finite tree automaton $\mathcal{A}_\cap = (Q_\cap, \mathcal{F}, Q_{f\cap}, \Delta_\cap)$ such that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

The construction is done by creating a *product automaton* (a tree automaton with state set that is the Cartesian product of state sets of input automata) $\mathcal{A}_\cap$ which simulates

Algorithm 3: Union automaton construction

Input: Input automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$
Output: $\mathcal{A}_\cup = (Q_\cup, \mathcal{F}, Q_{f\cup}, \Delta_\cup)$ such that $\mathcal{L}(\mathcal{A}_\cup) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$

```
1 begin
2     Q_∪ := Q_1 ∪ Q_2;
3     Q_f∪ := Q_f1 ∪ Q_f2;
4     Δ_∪ := Δ_1 ∪ Δ_2;
5     Δ_∪ () := Apply (Δ_1 ()) (Δ_2 ()) (λX Y . X ∪ Y);
6     return A_∪ = (Q_∪, F, Q_f∪, Δ_∪);
7 end
```

parallel run of both input automata:

$$\mathcal{A}_\cap = (Q_1 \times Q_2, \mathcal{F}, Q_{f1} \times Q_{f2}, \Delta_\cap) \tag{5.12}$$

where

$$\begin{aligned} \Delta_\cap \quad = \quad & \{f\,((q_{11}, q_{21}), \ldots, (q_{1n}, q_{2n})) \to (q_1, q_2) \mid f \in \mathcal{F}_n, \\ & f(q_{11}, \ldots, q_{1n}) \to q_1 \in \Delta_1, f(q_{21}, \ldots, q_{2n}) \to q_2 \in \Delta_2\} \end{aligned} \tag{5.13}$$

such that $\mathcal{A}_\cap$ contains only reachable states and transitions. Detection of reachable states is done by starting from initial super-states of automata, analysing all transitions from reachable super-states and collecting states that may be reached in this way until the algorithm has no unanalysed state. A super-state $(q_1, \ldots, q_n)$ is reachable if $\forall 1 \leq i \leq n : q_i$ is reachable. Due to the fact that we work with complete automata (with sink state $q_{sink} \notin Q_f$), whenever we reach a product state (q_1, q_{sink}) or (q_{sink}, q_2), we may stop generating further states (this is because q_{sink} has only transitions to q_{sink} so no accepting state can be reached from such state). Figure 5.3 shows the first step of construction of the product automaton. The construction process is described in Algorithm 4.

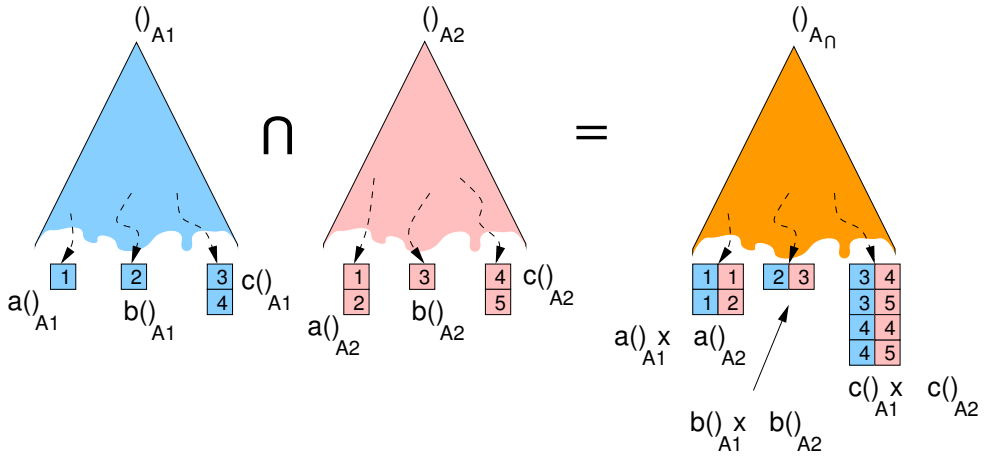


Figure 5.3: Construction of automaton $\mathcal{A}_\cap$ such that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

Algorithm 4: Intersection automaton construction

Input: Input automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$
Output: $\mathcal{A}_\cap = (Q_\cap, \mathcal{F}, Q_{f\cap}, \Delta_\cap)$ such that $\mathcal{L}(\mathcal{A}_\cap) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$

```
1  begin
2      Q_∩ := Q_f∩ := Δ_∩ := ∅;
3      newStates := empty queue;
4      Δ_∩ () := Apply (Δ_1 ()) (Δ_2 ()) (intersect newStates);
5      while newStates is not empty do
6          (q_a, q_b) := newStates.dequeue();
7          if (q_a, q_b) ∉ Q_∩ then
8              Q_∩ := Q_∩ ∪ {(q_a, q_b)};
9              if q_a = q_sink ∨ q_b = q_sink then continue;
10             if q_a ∈ Q_f1 ∧ q_b ∈ Q_f2 then Q_f∩ := Q_f∩ ∪ {(q_a, q_b)};
11             foreach n ∈ ℕ such that S_n(Δ_1) ≠ ∅ ∧ S_n(Δ_2) ≠ ∅ do
12                 foreach (q_11, ..., q_1n) ∈ S_n(Δ_1) such that q_a ∈ (q_11, ..., q_1n) do
13                     foreach (q_21, ..., q_2n) ∈ S_n(Δ_2) such that q_b ∈ (q_21, ..., q_2n) do
14                         if ∀1 ≤ i ≤ n : (q_1i, q_2i) ∈ Q_∩ then
15                             sp1 := (q_11, ..., q_1n);
16                             sp2 := (q_21, ..., q_2n);
17                             Δ_∩ ((q_11, q_21), ..., (q_1n, q_2n)) :=
                               Apply (Δ_1 sp1) (Δ_2 sp2) (intersect newStates);
18                         endif
19                     endfch
20                 endfch
21             endfch
22         endif
23     endw
24     return A_∩ = (Q_∩, F, Q_f∩, Δ_∩);
25 end
```

Function `intersect(`*newStates, lhs, rhs*`)`

```
1  begin
2      productSet := lhs × rhs;
3      foreach (q_a, q_b) ∈ productSet do
4          newStates.enqueue((q_a, q_b));
5      endfch
6      return productSet;
7  end
```

5.2.5 Determinisation

The *determinisation* operation takes an input finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and transforms it into a deterministic finite tree automaton $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{fd}, \Delta_d)$ such that $\mathcal{L}(\mathcal{A}_d) = \mathcal{L}(\mathcal{A})$.

The determinisation algorithm that is described in Algorithm 5 works with *macrostates*. A macrostate $M \subseteq Q$ is a state in the deterministic automaton that represents all states which might have been accessed during the run of the nondeterministic automaton over

the same sequence of symbols. The algorithm starts from the initial super-state, creates a new macrostate for each sink node of the MTBDD for the initial super-state and proceeds with finding all super-states $(q_1, \ldots, q_n)$ such that there exist macrostates $M_1, \ldots, M_n : \forall 1 \leq i \leq n : q_i \in M_i$. For each such super-state an MTBDD with union of sets at sink nodes of all MTBDDs that can be accessed by combinations of states in given macrostates is created; new macrostates are retrieved as sets of states from sink node of this MTBDD. This guarantees that only reachable states are present in the result automaton.

Algorithm 5: Automaton determinisation

Input: Input automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
Output: Deterministic automaton $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{fd}, \Delta_d)$, $\mathcal{L}(\mathcal{A}_d) = \mathcal{L}(\mathcal{A})$

```
1  begin
2      Q_d := Q_fd := Δ_d := ∅;
3      newStates := empty queue;
4      Δ_d () := MonadicApply (Δ ()) (collectSets newStates);
5      while newStates is not empty do
6          s := newStates.dequeue();
7          if s ∉ Q_d then
8              Q_d := Q_d ∪ {s};
9              if ∃q_f ∈ s such that q_f ∈ Q_f then Q_fd := Q_fd ∪ {s};
10             foreach n ∈ ℕ such that S_n(Δ) ≠ ∅ do
11                 foreach (q_1, ..., q_n) ∈ S_n(Δ) such that ∃1 ≤ i ≤ n : q_i ∈ s do
12                     foreach s_1, ..., s_n ∈ Q_d such that q_1 ∈ s_1, ..., q_n ∈ s_n, s_i = s
                       do
                           /* Create empty MTBDD                                  */
13                         tmp := ∅;
14                         foreach (p_1, ..., p_n) ∈ S_n(Δ) such that p_1 ∈ s_1, ..., p_n ∈ s_n
                           do
15                             tmp := Apply tmp (Δ (p_1, ..., p_n)) (λX Y . X ∪ Y);
16                         endfch
17                         Δ_d (s_1, ..., s_n) :=
                           MonadicApply tmp (collectSets newStates);
18                     endfch
19                 endfch
20             endfch
21         endif
22     endw
23     return A_d = (Q_d, F, Q_fd, Δ_d);
24 end
```

Function `collectSets(`*newStates, tf*`)`

```
1  begin
2      newStates.enqueue(tf);
3      return {tf};
4  end
```

5.2.6 Language Complementation

Given a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, the task of *language complementation* is to construct automaton $\mathcal{A}_c$ such that $\mathcal{L}(\mathcal{A}_c) = \overline{\mathcal{L}(\mathcal{A})}$. This is done by first transforming $\mathcal{A}$ to a deterministic automaton $\mathcal{A}_d = (Q_d, \mathcal{F}, Q_{fd}, \Delta_d)$ by the procedure described in Section 5.2.5 and then complementing the set of accepting states: $\mathcal{A}_c = (Q_d, \mathcal{F}, Q_d \setminus Q_{fd}, \Delta_d)$.

5.2.7 Automaton Reduction

Reduction of a finite tree automaton is a generic operation that takes a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ and a quotient set $Q/\sim$ of some equivalence relation $\sim$ and returns a reduced finite tree automaton $\mathcal{A}_r = (Q_r, \mathcal{F}, Q_{fr}, \Delta_r)$ such that $Q_r = Q/\sim$, $Q_{fr} = \{D \in Q_r \mid \exists q \in D : q \in Q_f\}$, and

$$\begin{aligned} \Delta_r \;=\; & \{f(B_1, \ldots, B_n) \to B \mid \\ & f(q_1, \ldots, q_n) \to q \in \Delta, f \in \mathcal{F}, q_1 \in B_1, \ldots, q_n \in B_n, q \in B\}\,. \end{aligned} \tag{5.14}$$

Various methods can be used for obtaining the equivalence relation $\sim$, e.g. Myhill-Nerode minimisation (see Section 5.2.9) or downward simulation (see Section 5.2.11). Note that while the former approach can be used over deterministic finite tree automata only, the latter may be used to reduce the size of nondeterministic finite tree automata as well (in their case, however, the result is not a minimal nondeterministic finite tree automaton but reduced nondeterministic finite tree automaton only). The algorithm for reduction of a finite tree automaton is given in Algorithm 6.

Algorithm 6: Automaton reduction

Input: Input automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
Quotient set $Q/\sim$
Output: Reduced automaton $\mathcal{A}_r = (Q_r, \mathcal{F}, Q_{fr}, \Delta_r)$

```
1  begin
2      Q_r := Q/∼;
3      Q_fr := {[q]_∼ | q ∈ Q_f};
4      Δ_r := ∅;
5      foreach n ∈ ℕ such that S_n(Δ) ≠ ∅ do
6          foreach (q_1, ..., q_n) ∈ S_n(Δ) do
7              sp := ([q_1]_∼, ..., [q_n]_∼);
8              Δ_r sp := Apply (Δ_r sp) (Δ (q_1, ..., q_n)) (λX Y . X ∪ {[y]_∼ | y ∈ Y});
9          endfch
10     endfch
11     return A_r = (Q_r, F, Q_fr, Δ_r);
12 end
```

5.2.8 Pruning Unreachable States

The task of *pruning unreachable states* of a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is removal of states q (and corresponding transitions, which means removing MTBDDs for all super-states that contain q) for which there does not exist a tree $t \in T(\mathcal{F})$ such that $t \to^*_{\mathcal{A}} q$. The algorithm attempts to simulate the run of the automaton for all possible trees and collect states that can be reached. The description of the algorithm is in Algorithm 7.

Algorithm 7: Unreachable states pruning

Input: Input automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
Output: Automaton $\mathcal{A}_p = (Q_p, \mathcal{F}, Q_{fp}, \Delta_p)$ without unreachable states, such that $\mathcal{L}(\mathcal{A}_p) = \mathcal{L}(\mathcal{A})$

```
1  begin
2      Q_p := Q_fp := Δ_p := ∅;
3      reachStates := empty queue;
4      Δ_p () := MonadicApply (Δ ()) (collectReachable reachStates);
5      while reachStates is not empty do
6          q := reachStates.dequeue();
7          if q ∉ Q_p then
8              Q_p := Q_p ∪ {q};
9              if q ∈ Q_f then Q_fp := Q_fp ∪ {q};
10             foreach n ∈ ℕ such that S_n(Δ) ≠ ∅ do
11                 foreach (q_1, ..., q_n) ∈ S_n(Δ) such that q ∈ (q_1, ..., q_n) do
12                     if ∀1 ≤ i ≤ n : q_i ∈ Q_p then
13                         sp := (q_1, ..., q_n);
14                         Δ_p sp :=
                           MonadicApply (Δ sp) (collectReachable reachStates);
15                     endif
16                 endfch
17             endfch
18         endif
19     endw
20     return A_p = (Q_p, F, Q_fp, Δ_p);
21 end
```

Function `collectReachable`(*reachStates, leaf*)

```
1  begin
2      foreach q ∈ leaf do
3          reachStates.enqueue(q);
4      endfch
5      return leaf;
6  end
```

5.2.9 Minimisation

Automaton minimisation is an operation on a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ which returns deterministic finite tree automaton $\mathcal{A}_m = (Q_m, \mathcal{F}, Q_{fm}, \Delta_m)$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_m)$ and $\mathcal{A}_m$ is an automaton that has the least states from all deterministic finite tree automata that accept $\mathcal{L}(\mathcal{A})$. Existence of a minimum deterministic finite tree automaton is guaranteed by the proof of Myhill-Nerode Theorem (see Section 2.4).

The minimisation process starts with pruning unreachable states and determinising the input automaton. Once done, Algorithm 8 computes equivalence relation for congruence (used in Myhill-Nerode Theorem) $\sim$ on Q. This is done by refining the equivalence relation

from the start point with two initial classes: the accepting states and the non-accepting states. All super-states $(q_1, \ldots, q_n)$ are then searched and in case there exists an equivalence class $[q_i]_\sim$ such that when q_i is substituted in $(q_1, \ldots, q_n)$ for some other element from $[q_i]_\sim$ and the target class of transitions over respective symbols differs, then $\sim$ is refined.

In the following step, the quotient set of $\sim$ is passed to the reduction procedure (see Section 5.2.7) and obtaining the minimum automaton is straightforward.

Algorithm 8: Computation of $\sim$ equivalence over states

```
   Input: Deterministic automaton without unreachable states A = (Q, F, Q_f, Δ)
   Output: Equivalence relation ∼⊆ Q × Q
 1 begin
 2     eq := {(p, q) | p ∈ Q_f ⇔ q ∈ Q_f};
 3     prevEq := ∅;
 4     while eq ≠ prevEq do
 5         prevEq := eq;
 6         foreach n ∈ ℕ such that S_n(Δ) ≠ ∅ do
 7             foreach (q_1, ..., q_n) ∈ S_n(Δ) do
 8                 foreach 1 ≤ i ≤ n do
 9                     foreach q ∈ [q_i]_prevEq do
10                         sp_qi := (q_1, ..., q_{i-1}, q_i, q_{i+1}, ..., q_n);
11                         sp_q := (q_1, ..., q_{i-1}, q, q_{i+1}, ..., q_n);
12                         if sp_q ∈ S_n(Δ) then
13                             refined := false;
14                             Apply (Δ sp_qi) (Δ sp_q) (refineEq prevEq refined);
15                             if refined then eq := eq \ {(q, q_i), (q_i, q)};
16                         else
17                             eq := eq \ {(q, q_i), (q_i, q)};
18                         endif
19                     endfch
20                 endfch
21             endfch
22         endfch
23     endw
24     return ∼= eq;
25 end
```

Function `refineEq`(*prevEq, refined,* {*lhs*}, {*rhs*})

```
1 begin
2     if [lhs]_prevEq ≠ [rhs]_prevEq then
3         refined := true;
4     endif
5 end
```

5.2.10 Checking Language Emptiness

The problem of *determining emptiness* of a language is defined as given a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$, is $\mathcal{L}(\mathcal{A}) = \emptyset$? The algorithm for deciding language emptiness first

removes unreachable states from automaton $\mathcal{A}$ using the method described in Section 5.2.8. This constructs a finite tree automaton $\mathcal{A}_p = (Q_p, \mathcal{F}, Q_{fp}, \Delta_p)$ without unreachable states. It holds that language $\mathcal{L}(\mathcal{A}_p)$ is empty if and only if $Q_{fp} = \emptyset$ (i.e. there is no reachable final state in $\mathcal{A}_p$). Note that a slightly more efficient algorithm can determine that $\mathcal{L}(\mathcal{A}) \neq \emptyset$ immediately when the analysis of reachable states reaches state q such that $q \in Q_f$.

5.2.11 Downward Simulation Reduction

Downward simulation [26] $\preceq$ for a finite tree automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$ is a binary relation on Q such that if $q \preceq r$ and $f(q_1, \ldots, q_n) \to q \in \Delta$, then there are $r_1, \ldots, r_n$ such that $f(r_1, \ldots, r_n) \to r \in \Delta$ and $q_i \preceq r_i$ for each $1 \leq i \leq n$. Formally:

$$\begin{aligned} \forall f \in \mathcal{F} \quad : \quad & [q \preceq r \wedge f(q_1, \ldots, q_n) \to q \in \Delta] \Rightarrow \\ & [\exists r_1, \ldots, r_n \in Q : f(r_1, \ldots, r_n) \to r \in \Delta \wedge \forall 1 \leq i \leq n : q_i \preceq r_i] \end{aligned} \tag{5.15}$$

From the previous equation, the following can be inferred using *modus tollens*:

$$\begin{aligned} \forall f \in \mathcal{F} \quad : \quad & \neg [\exists r_1, \ldots, r_n \in Q : f(r_1, \ldots, r_n) \to r \in \Delta \wedge \forall 1 \leq i \leq n : q_i \preceq r_i] \Rightarrow \\ & [\neg (q \preceq r) \vee \neg (f(q_1, \ldots, q_n) \to q \in \Delta)] \end{aligned} \tag{5.16}$$

We further expand $\preceq$ relation to super-states:

$$(q_1, \ldots, q_n) \preceq (r_1, \ldots, r_n) \overset{\text{def}}{\Leftrightarrow} \forall 1 \leq i \leq n : q_i \preceq r_i \tag{5.17}$$

It can be proved that $\preceq$ is reflexive and transitive. It is possible to use downward simulation for reduction of the size of an automaton by identifying states that simulate each other and collapsing those states together. Even though an automaton obtained in this way is often not *minimum*, the reduction can be significant and computation is faster than minimisation which needs to first convert the automaton to deterministic one.

The algorithm for computation of downward simulation, described in Algorithm 9, starts with declaring $\preceq = Q \times Q$ and then for each super-state $(q_1, \ldots, q_n)$ finds all super-states $(r_1, \ldots, r_n)$ such that $(q_1, \ldots, q_n) \preceq (r_1, \ldots, r_n)$ and makes a new MTBDD with uniting the sink nodes of those. This union MTBDD represents all states r that can be reached using super-states $(r_1, \ldots, r_n)$ simulating super-state $(q_1, \ldots, q_n)$. Now for each state q accessible from $(q_1, \ldots, q_n)$ over symbol $f \in \mathcal{F}$ we check for each r such that $q \preceq r$ that r is in the union MTBDD accessible over f. In case it is not, according to Equation 5.16 the simulation relation $\preceq$ can be is refined by removing (q, r) from $\preceq$. This is repeated until $\preceq$ reaches the fixpoint.

As downward simulation is reflexive and transitive but generally not symmetric, symmetric closure of the relation needs to be performed in order to obtain equivalence relation. Reduction is then performed using the generic reduction procedure as described in Section 5.2.7.

5.2.12 Checking Language Inclusion Using Antichains

The *language inclusion* decision problem is to determine for two input finite tree automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$ whether it holds that $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$. The standard approach of checking language inclusion is by determinising $\mathcal{A}_2$, complementing it, and checking whether $\mathcal{L}(\mathcal{A}_1) \cap \overline{\mathcal{L}(\mathcal{A}_2)} = \emptyset$. In case the intersection is not empty, it means that there are some trees which are in $\mathcal{L}(\mathcal{A}_1)$ and not in $\mathcal{L}(\mathcal{A}_2)$ and therefore the

Algorithm 9: Downward simulation computation

Input: Input automaton $\mathcal{A} = (Q, \mathcal{F}, Q_f, \Delta)$
Output: Simulation relation $\preceq \subseteq Q \times Q$

```
1  begin
2      prevSim := ∅;
3      sim := Q × Q;
4      while prevSim ≠ sim do
5          prevSim := sim;
6          foreach n ∈ ℕ such that S_n(Δ) ≠ ∅ do
7              foreach (q_1, ..., q_n) ∈ S_n(Δ) do
                   /* Create empty MTBDD                                  */
8                  tmp := ∅;
9                  foreach (r_1, ..., r_n) ∈ S_n(Δ) such that ∀1 ≤ i ≤ n : (q_i, r_i) ∈ sim
                   do
10                     tmp := Apply tmp (Δ (r_1, ..., r_n)) (λX Y . X ∪ Y);
11                 endfch
12                 Apply (Δ (q_1, ..., q_n)) tmp (simulationRefinement sim);
13             endfch
14         endfch
15     endw
16     return ⪯= sim;
17 end
```

Function `simulationRefinement(`*sim, lhs, rhs*`)`

```
1  begin
2      foreach q ∈ lhs do
3          foreach r such that (q, r) ∈ sim do
4              if r ∉ rhs then sim := sim \ {(q, r)};
5          endfch
6      endfch
7  end
```

inclusion does not hold. Nevertheless complementation needs determinisation of $\mathcal{A}_2$ which is often very expensive. Therefore it is desirable to find approaches that do not need this operation.

One approach that avoids determinisation is based on *antichains* [20]. An antichain over $Q_1 \times 2^{Q_2}$ is a set $S \subseteq Q_1 \times 2^{Q_2}$ such that for every $(p, s), (p', s') \in S$ if $p = p'$ then $s \not\subset s'$. For $(p, s) \in S$, p denotes a state from $\mathcal{A}_1$ that is reachable over some tree and s denotes a set of states of automaton $\mathcal{A}_2$ that are reachable over the same tree. If such a pair (p, s) can be reached so that $p \in Q_{f1}$ and $\forall r \in s : r \notin Q_{f2}$, the inclusion $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ does not hold. The algorithm is given in Algorithm 10.

Algorithm 10: Antichain-based inclusion

Input: Input automata $\mathcal{A}_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\mathcal{A}_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$
Output: **true** if $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$, **false** otherwise

```
1  begin
2      prevAntichain := ∅;
3      antichain := ∅;
4      Apply (Δ1 ()) (Δ2 ()) (collectProducts antichain);
5      while antichain ≠ prevAntichain do
6          prevAntichain := antichain;
7          foreach (q, D) ∈ prevAntichain do
8              if q ∈ Qf1 ∧ ∀p ∈ D : p ∉ Qf2 then return false;
9          endfch
10         foreach n ∈ ℕ such that Sn(Δ1) ≠ ∅ do
11             foreach (q1, ..., qn) ∈ Sn(Δ1) such that
               ∀1 ≤ i ≤ n : ∃Ri ⊆ Q2 : (qi, Ri) ∈ prevAntichain do
12                 tmp := ∅;
13                 foreach (s1, ..., sn) ∈ Sn(Δ2) such that ∀1 ≤ i ≤ n : si ∈ Ri do
14                     tmp := Apply tmp (Δ2 (s1, ..., sn)) (λX Y . X ∪ Y);
15                 endfch
16                 Apply (Δ1 (q1, ..., qn)) tmp (collectProducts antichain);
17             endfch
18         endfch
19     endw
20     return true;
21 end
```

5.3 Transducers

This section starts with a description of the representation of *relabelling* (or sometimes called *structure-preserving*) tree transducers. These are transducers that do not change the structure of input trees but only change symbols in their nodes. The section continues by a definition of two operations that are necessary in regular tree model checking: performing a transduction step on a finite tree automaton and composition of transducers.

5.3.1 Representation of a Relabelling Tree Transducer

We represent only relabelling tree transducers that use the same alphabet $\mathcal{F}$ for both input and output, we therefore refer to transducer $\tau = (Q, \mathcal{F}, \mathcal{F}' = \mathcal{F}, Q_f, \Delta)$ by $\tau = (Q, \mathcal{F}, Q_f, \Delta)$. Relabelling tree transducers contain transduction rules of the following type:

$$f(q_1(x_1), \ldots, q_n(x_n)) \to q(g(x_1, \ldots, x_n)), \tag{5.18}$$

where $n \in \mathbb{N}$, $f, g \in \mathcal{F}_n$, $q, q_1, \ldots, q_n \in Q$ and $x_1, \ldots, x_n \in \mathcal{X}$, or using an alternative notation as

$$f(q_1, \ldots, q_n) \to q(g). \tag{5.19}$$

The representation of a transduction function Δ of a relabelling tree transducer is therefore very similar to the representation of a transition function of a finite tree automaton

Function `collectProducts(`*antichain,lhs, rhs*`)`

```
1 begin
2     foreach q ∈ lhs do
3         if ∄(q, E) ∈ antichain such that rhs ⊆ E then
4             antichain := (antichain \ {(q, F) | F ⊂ rhs}) ∪ {(q, rhs)};
5         endif
6     endfch
7 end
```

and can again be symbolic. We naturally expand the definition of a super-state $S(\Delta)$ to the transduction function. The transduction function Δ of a relabelling tree transducer τ may then be alternatively defined as a mapping $\Delta^\bullet$ in the following way:

$$\begin{aligned} \Delta^\bullet \quad : \quad & S \to \big(\mathcal{F} \to (\mathcal{F} \to 2^Q)\big) \\ & (q_1, \ldots, q_p) \mapsto \{(f, (g, D)) \mid D = \{q \mid f(q_1, \ldots, q_p) \to q(g) \in \Delta\}\}\,. \end{aligned} \tag{5.20}$$

However, since the composition of functions is associative, the formula in Equation 5.20 can be rewritten as

$$\begin{aligned} \Delta^\bullet \quad : \quad & S \to \big((\mathcal{F} \to \mathcal{F}) \to 2^Q\big) \\ & (q_1, \ldots, q_p) \mapsto \{((f, g), D) \mid D = \{q \mid f(q_1, \ldots, q_p) \to q(g) \in \Delta\}\} \end{aligned} \tag{5.21}$$

(we again confuse Δ and $\Delta^\bullet$). This means that we can represent a transduction function of a relabelling tree transducer using MTBDDs in the same way as a transition function of a finite tree automaton, provided we expand the function $enc : \mathcal{F} \to \{0,1\}^n$, which is defined in Section 5.1, to $enc_T : (\mathcal{F} \times \mathcal{F}) \to \{0,1\}^{2n}$ in the following way:

$$\begin{aligned} enc_T \quad : \quad & (\mathcal{F} \times \mathcal{F}) \to \{0,1\}^{2n} \\ & (a, b) \mapsto (a_1, \ldots, a_n, b_1, \ldots, b_n) \\ & \qquad \text{where} \qquad (a_1, \ldots, a_n) = enc(a) \quad \text{and} \quad (b_1, \ldots, b_n) = enc(b). \end{aligned} \tag{5.22}$$

Note that the actual ordering of $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ is not important provided that it remains consistent for enc_T. Another ordering which may be useful for some cases is for instance $(a_1, b_1, \ldots, a_n, b_n)$.

In case we denote the MTBDD for a super-state $s_\mathcal{A}$ of the transition function $\Delta_\mathcal{A}$ of a finite tree automaton $\mathcal{A}$ as

$$\sum_{\substack{f \in \mathcal{F} \\ enc(f) = (a_1, \ldots, a_n)}} \left(\prod_{a_i = 0} \neg x_i \cdot \prod_{a_i = 1} x_i \cdot s_\mathcal{A}(f) \right) \tag{5.23}$$

(see Section 3.2 for further details of this notation), we may represent MTBDD for a super-state s_τ of the transduction function Δ_τ of a relabelling tree transducer τ as

$$\sum_{\substack{(f,g) \in \mathcal{F} \times \mathcal{F} \\ enc_T(f,g) = \\ (a_1, \ldots, a_n, b_1, \ldots, b_n)}} \left(\prod_{a_i = 0} \neg x_i \cdot \prod_{a_i = 1} x_i \cdot \prod_{b_i = 0} \neg y_i \cdot \prod_{b_i = 1} y_i \cdot s_\tau(f, g) \right). \tag{5.24}$$

This representation works with an MTBDD extended by Boolean variables $y_1, \ldots, y_n$ (we assume that the representation of finite tree automata described in Section 5.1 uses variables $x_1, \ldots, x_n$). In order to support operations that work with both finite tree automata and relabelling tree transducers, the following two functions that work directly with the structure of MTBDDs are necessary:

TrimVariables This function receives an MTBDD M and x, which is a base name of Boolean variables $x_1, \ldots, x_n$ such that $x_1, \ldots, x_n$ are in M, and returns MTBDD M_{-x} that does not contain $x_1, \ldots, x_n$. Hence, $\mathit{TrimVariables}(M, x) = M_{-x}$. Since this operation may cause collisions (e.g. producing formula $y_1 y_2 A + y_1 y_2 B$ where $A \neq B, A \neq \bot, B \neq \bot$), they need to be properly handled by uniting colliding state sets (e.g. producing $y_1 y_2 (A \cup B)$ for the previous example). The following formula formally defines the function:

$$\mathit{TrimVariables}\left(\sum_{\substack{(f,g)\in\mathcal{F}\times\mathcal{F}\\ enc_T(f,g)=\\ (a_1,\ldots,a_n,b_1,\ldots,b_n)}}\left(\prod_{a_i=0}\neg x_i \cdot \prod_{a_i=1} x_i \cdot \prod_{b_i=0}\neg y_i \cdot \prod_{b_i=1} y_i \cdot J(f,g)\right), x\right)$$
$$= \sum_{\substack{g\in\mathcal{F}\\ enc(g)=(b_1,\ldots,b_n)}}\left(\prod_{b_i=0}\neg y_i \cdot \prod_{b_i=1} y_i \cdot \bigcup_{f\in\mathcal{F}} J(f,g)\right) \tag{5.25}$$

The implementation of $\mathit{TrimVariables}(M, x)$ can be done in the following way:

> Traverse M from the root to sink nodes and for each node k on the path such that k represents some x_i do the following: take both child nodes of k, k_0 and k_1, and set k to $k := \mathit{Apply}\ k_0\ k_1\ (\lambda X\ Y\ .\ X \cup Y)$.

RenameVariables A function that receives an MTBDD M and names of two Boolean variables x and y such that $x_1, \ldots, x_n$ are in M. The function renames all occurrences of x_i to y_i for each $1 \leq i \leq n$. The function is formally defined by the following formula:

$$\mathit{RenameVariables}\left(\sum_{\substack{f\in\mathcal{F}\\ enc(f)=(a_1,\ldots,a_n)}}\left(\prod_{a_i=0}\neg x_i \cdot \prod_{a_i=1} x_i \cdot J(f)\right), x, y\right)$$
$$= \sum_{\substack{f\in\mathcal{F}\\ enc(f)=(b_1,\ldots,b_n)}}\left(\prod_{b_i=0}\neg y_i \cdot \prod_{b_i=1} y_i \cdot J(f)\right) \tag{5.26}$$

The implementation of $\mathit{RenameVariables}(M, x, y)$ simply traverses M and renames all occurrences of x_i to y_i (assuming that $y_1, \ldots, y_n$ are not in M).

5.3.2 Performing a Transduction Step

The operation of *performing a transduction step* of a finite tree automaton $\mathcal{A} = (Q_\mathcal{A}, \mathcal{F}, Q_{f\mathcal{A}}, \Delta_\mathcal{A})$ according to the transduction denoted by a relabelling tree transducer $\tau = (Q_\tau, \mathcal{F}, Q_{f\tau}, \Delta_\tau)$ constructs a finite tree automaton $\mathcal{A}_w = (Q_w, \mathcal{F}, Q_{fw}, \Delta_w)$ such that $\mathcal{L}(\mathcal{A}_w) = \tau(\mathcal{L}(\mathcal{A}))$.

Informally, if $\mathcal{A}$ represents a set of configurations of a system and τ represents transitions in the system, then $\tau(\mathcal{L}(\mathcal{A}))$ is a finite tree automaton that represents the set of configurations of the system after one transition.

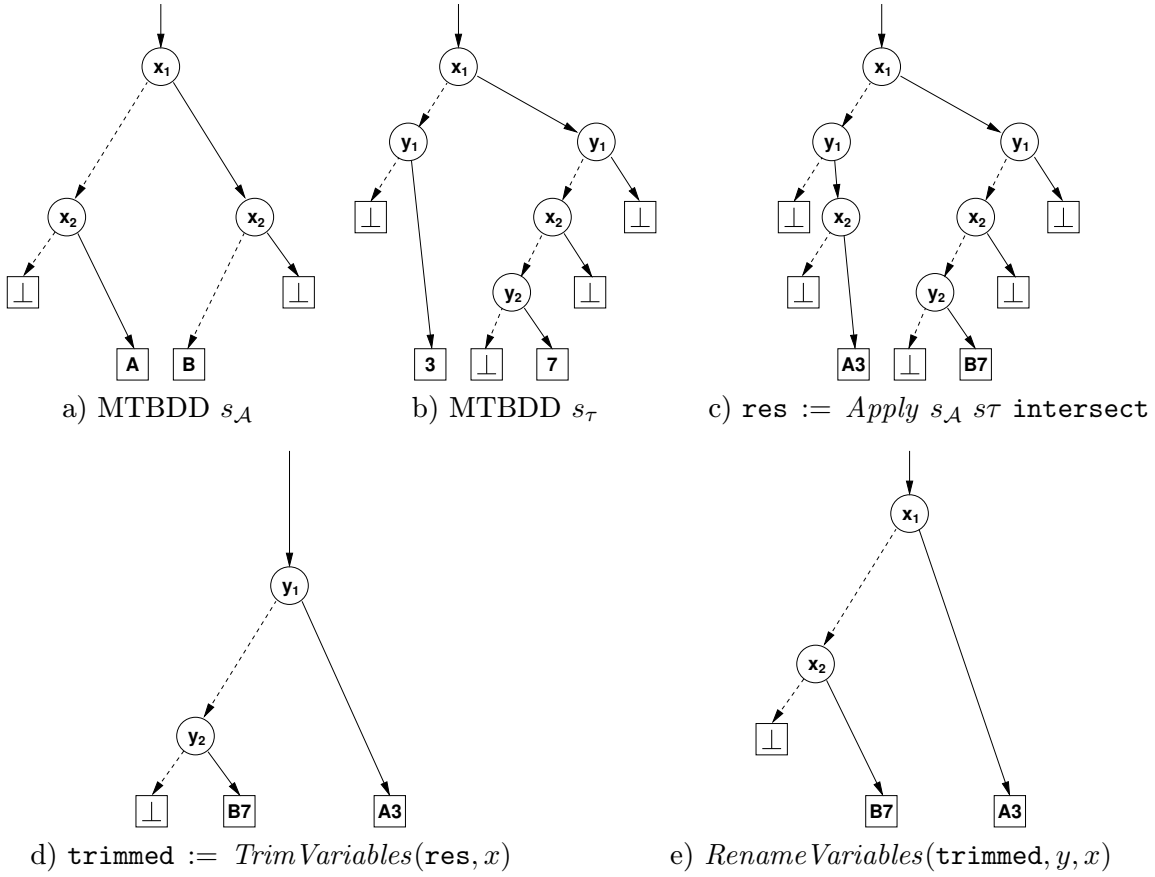
Algorithm 11: Performing transduction step

Input: Input automaton $\mathcal{A} = (Q_{\mathcal{A}}, \mathcal{F}, Q_{f\mathcal{A}}, \Delta_{\mathcal{A}})$
Relabelling tree transducer $\tau = (Q_{\tau}, \mathcal{F}, Q_{f\tau}, \Delta_{\tau})$
Output: Automaton $\mathcal{A}_w = (Q_w, \mathcal{F}, Q_{fw}, \Delta_w)$ such that $\mathcal{L}(\mathcal{A}_w) = \tau(\mathcal{L}(\mathcal{A}))$

```
1  begin
2    Q_w := Q_fw := Δ_w := ∅;
3    newStates := empty queue;
4    tmp := Apply (Δ_A ()) (Δ_τ ()) (intersect newStates);
5    tmp := TrimVariables(tmp, x);
6    Δ_w () := RenameVariables(tmp, y, x);
7    while newStates is not empty do
8      (q_a, q_b) := newStates.dequeue();
9      if (q_a, q_b) ∉ Q_w then
10       Q_w := Q_w ∪ {(q_a, q_b)};
11       if q_a = q_sink ∨ q_b = q_sink then continue;
12       if q_a ∈ Q_fA ∧ q_b ∈ Q_fτ then Q_fw := Q_fw ∪ {(q_a, q_b)};
13       foreach n ∈ ℕ such that S_n(Δ_A) ≠ ∅ ∧ S_n(Δ_τ) ≠ ∅ do
14         foreach (q_11, ..., q_1n) ∈ S_n(Δ_A) such that q_a ∈ (q_11, ..., q_1n) do
15           foreach (q_21, ..., q_2n) ∈ S_n(Δ_τ) such that q_b ∈ (q_21, ..., q_2n) do
16             if ∀1 ≤ i ≤ n : (q_1i, q_2i) ∈ Q_w then
17               sp1 := (q_11, ..., q_1n);
18               sp2 := (q_21, ..., q_2n);
19               tmp :=
                 Apply (Δ_A sp1) (Δ_τ sp2) (intersect newStates);
20               tmp := TrimVariables(tmp, x);
21               Δ_w ((q_11, q_21), ..., (q_1n, q_2n)) :=
                 RenameVariables(tmp, y, x);
22             endif
23           endfch
24         endfch
25       endfch
26     endif
27   endw
28   return A_w = (Q_w, F, Q_fw, Δ_w);
29 end
```

The algorithm for this operation is described in Algorithm 11. The algorithm assumes that MTBDDs for transition function $\Delta_{\mathcal{A}}$ are defined over Boolean variables $x_1, \ldots, x_n$ and that MTBDDs for transduction function Δ_{τ} are defined over Boolean variables $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, where $x_1, \ldots, x_n$ are used for input symbols of the transducer and $y_1, \ldots, y_n$ are used for output symbols. MTBDDs for the output automaton are again over Boolean variables $x_1, \ldots, x_n$. The algorithm works by traversing both the automaton and the transducer in parallel and performing relabelling of transitions which are in both (the algorithm may resemble the computation of intersection, it actually uses function `intersect()` which

is defined in Section 5.2.4). Figure 5.4 attempts to give the idea about how the algorithm works for a pair of super-states.



a) MTBDD $s_{\mathcal{A}}$ b) MTBDD s_{τ} c) `res` := *Apply* $s_{\mathcal{A}}$ $s\tau$ `intersect`

d) `trimmed` := *TrimVariables*(`res`, x) e) *RenameVariables*(`trimmed`, y, x)

Figure 5.4: An example of performing transduction step of transducer τ on automaton $\mathcal{A}$ for one pair of super-states s_{τ} and $s_{\mathcal{A}}$, such that $\texttt{01}(s_{\mathcal{A}}) \to A, \texttt{10}(s_{\mathcal{A}}) \to B$, and $\texttt{0X}(s_{\tau}) \to 3(\texttt{1X}), \texttt{10}(s_{\tau}) \to 7(\texttt{01})$. Ordering $(a_1, b_1, \ldots, a_n, b_n)$ is used.

5.3.3 Transducer Composition

Transducer composition is an operation that, when given two relabelling tree transducers $\tau_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\tau_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$, creates a relabelling tree transducer $\tau = (Q, \mathcal{F}, Q_f, \Delta)$ such that for all finite tree automata $\mathcal{A}$, it holds that $\tau(\mathcal{L}(\mathcal{A})) = \tau_2(\tau_1(\mathcal{L}(\mathcal{A})))$ (or $\tau = \tau_2 \circ \tau_1$).

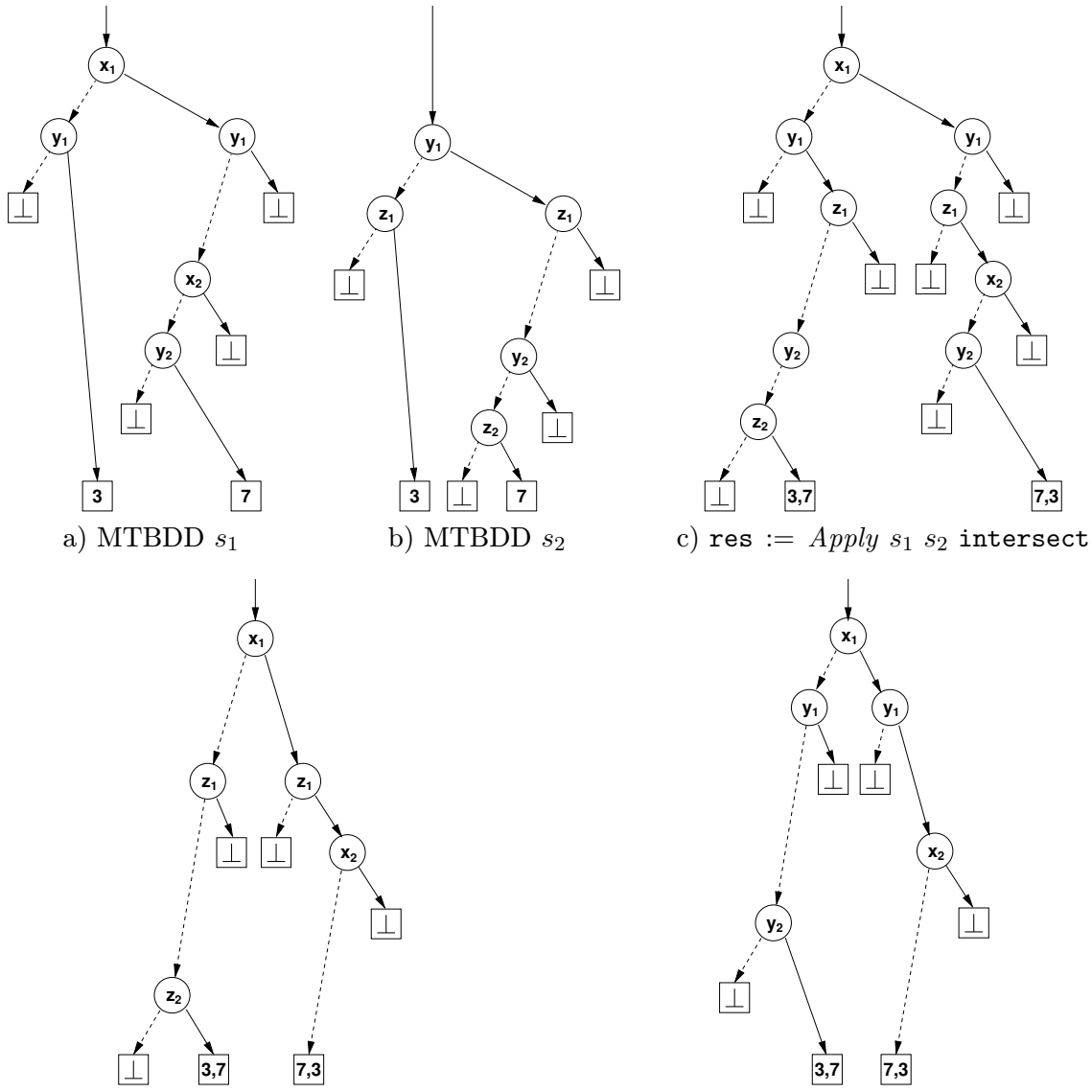
The algorithm described as Algorithm 12 assumes that MTBDDs for transduction functions Δ_1 and Δ_2 are over Boolean variables $x_1, \ldots, x_n$ (which encode the input symbol) and $y_1, \ldots, y_n$ (which encode the output symbol). The MTBDDs for the transduction function of the constructed transducer are again over Boolean variables $x_1, \ldots, x_n$ for the input and $y_1, \ldots, y_n$ for the output. However, the MTBDDs need to be able to also work with Boolean variables $z_1, \ldots, z_n$ as they are used inside the algorithm.

The algorithm is very similar to the algorithm that performs a transduction step on a finite tree automaton (see Section 5.3.2). Figure 5.5 gives an example of the operations carried out by the algorithm for one pair of super-states.

Algorithm 12: Transducer composition

Input: Input transducers $\tau_1 = (Q_1, \mathcal{F}, Q_{f1}, \Delta_1)$ and $\tau_2 = (Q_2, \mathcal{F}, Q_{f2}, \Delta_2)$
Output: Transducer $\tau = (Q, \mathcal{F}, Q_f, \Delta)$ such that $\tau = \tau_2 \circ \tau_1$

```
1  begin
2      Q := Q_f := Δ := ∅;
3      newStates := empty queue;
4      tmp := RenameVariables(Δ_2 (), y, z);
5      tmp := RenameVariables(tmp, x, y);
6      tmp := Apply (Δ_1 ()) tmp (intersect newStates);
7      tmp := TrimVariables(tmp, y);
8      Δ () := RenameVariables(tmp, z, y);
9      while newStates is not empty do
10         (q_a, q_b) := newStates.dequeue();
11         if (q_a, q_b) ∉ Q then
12             Q := Q ∪ {(q_a, q_b)};
13             if q_a = q_sink ∨ q_b = q_sink then continue;
14             if q_a ∈ Q_f1 ∧ q_b ∈ Q_f2 then Q_f := Q_f ∪ {(q_a, q_b)};
15             foreach n ∈ ℕ such that S_n(Δ_1) ≠ ∅ ∧ S_n(Δ_2) ≠ ∅ do
16                 foreach (q_11, ..., q_1n) ∈ S_n(Δ_1) such that q_a ∈ (q_11, ..., q_1n) do
17                     foreach (q_21, ..., q_2n) ∈ S_n(Δ_2) such that q_b ∈ (q_21, ..., q_2n) do
18                         if ∀1 ≤ i ≤ n : (q_1i, q_2i) ∈ Q then
19                             tmp := RenameVariables(Δ_2 (q_21, ..., q_2n), y, z);
20                             tmp := RenameVariables(tmp, x, y);
21                             sp := (q_11, ..., q_1n);
22                             tmp := Apply (Δ_1 sp) tmp (intersect newStates);
23                             tmp := TrimVariables(tmp, y);
24                             Δ ((q_11, q_21), ..., (q_1n, q_2n)) :=
                               RenameVariables(tmp, z, y);
25                         endif
26                     endfch
27                 endfch
28             endfch
29         endif
30     endw
31     return τ = (Q, F, Q_f, Δ);
32 end
```



a) MTBDD s_1 b) MTBDD s_2 c) `res` := *Apply* s_1 s_2 `intersect`

d) `trimmed` := *TrimVariables*(`res`, y) e) *RenameVariables*(`trimmed`, z, y)

Figure 5.5: An example of performing transducer composition of transducer τ on itself: $\tau \circ \tau$, for one super-state s_τ, such that $\texttt{0X}(s_\tau) \rightarrow 3(\texttt{1X}), \texttt{10}(s_\tau) \rightarrow 7(\texttt{01})$. We assume that $s_1 = s_\tau$ and $s_2 = \mathit{RenameVariables}(\mathit{RenameVariables}(s_\tau, y, z), x, y)$. Ordering $(a_1, b_1, c_1, \ldots, a_n, b_n, c_n)$ is used.

Chapter 6

Implementation

This chapter describes design and implementation of a prototype of the library. It starts with description of the implementation of the type of MTBDDs as defined in Section 5.1. This is followed by description of the object-oriented design of the implementation.

6.1 MTBDD Package

Since a smart and efficient implementation of MTBDDs is not trivial, it was decided that an existing library should be used instead of implementing an own BDD package. For this purpose, CUDD [29] (distributed free of charge under the new and simplified BSD licence [30]), which is a C library implementing shared BDDs, ADDs (algebraic decision diagrams) and ZDDs (zero-suppressed decision diagrams), has been chosen.

Using this library, we represent an MTBDD by an ADD [31], which is in fact an MTBDD that puts emphasis on performing algebraic operations (such as addition, multiplication, or computation of logarithm) on sets of floating point numbers represented by the diagram. Despite such broad range of operations we use the data structure only for storage and retrieval of data and performing *Apply* operation. Because CUDD only allows to store floating point numbers into the sink nodes of MTBDDs, we had to deal with the problem to substitute those floating point numbers for sets of states of an automaton (as described in Section 5.1). We solved this problem by patching the library so that sink nodes would contain pointers to sets stored in another data structure, which serves as a pool of sets of states. In order to make use of MTBDD's space reduction, it must hold that there are never two equal sets of states in the pool. This means that two pointers point to the same set of states if and only if they are equal.

As shared variation of MTBDD is used, a way to distinguish among individual MTBDDs in such structure needs to be defined. We use another data structure that provides mapping from the set of super-states of the transition function to roots of the shared MTBDD. The resulting wrapper over MTBDDs provided by CUDD is shown in Figure 6.1.

Due to the fact that many algorithms in Chapter 5 need to alter some data outside of MTBDD during an *Apply* operation, we also patched CUDD and support the following *Apply* operation: *Apply*`(lhs, rhs, op)`, where `lhs` and `rhs` are sets of states of the left and right MTBDD respectively, and `op` is a function object: an object that can be called like an ordinary function. Using `op` to pass pointers to data structures in main subroutines, we can avoid using global variables and thus making the code re-entrant and less prone to programming errors.

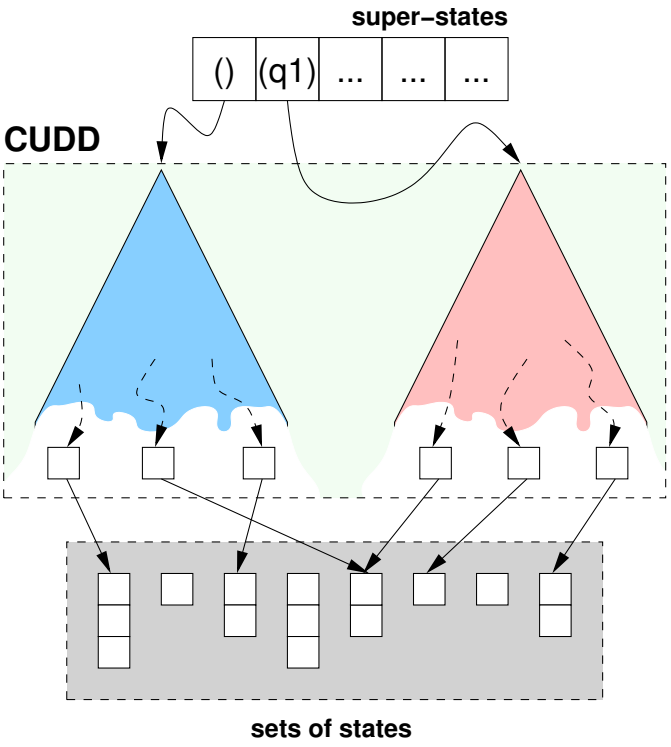


Figure 6.1: Wrapper over CUDD-provided MTBDDs.

6.2 Object-Oriented Design

C++ has been chosen as the implementation language because of its efficiency, good support, means for modular design and an extensive standard library. We employ C++'s support of object-oriented programming paradigm to create a generic and modular design, which is further described in this section.

In order to provide both modularity and good performance, policy-based design [32] is exploited in the object-oriented design of the library. This approach uses *policy classes*, which are classes that are not supposed to be instantiated (which can be enforced by making their constructor protected) or to only provide interface, but rather to provide a certain functionality when inherited by some class called the *host class*. Each policy class implements a particular interface called a *policy*. The host class is a class template, i.e. an incomplete class that does not name a type by itself but needs to be have its template arguments bound in order to do so, as shown in Figure 6.2. In addition to standard

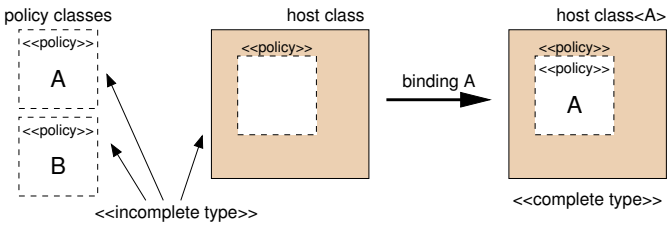


Figure 6.2: Binding of policy classes to host class.

template arguments the host class also defines its policies. Using multiple inheritance, several orthogonal policy classes may be inherited by host class. Due to the fact that policy classes of a host class are resolved statically during compile time, the compiler may perform certain optimizations, such as inlining of code, which is an advantage over using e.g. virtual methods.

6.2.1 MTBDD Wrapper

`CUDDFacade` is a class that was designed according to the façade design pattern [33]. It is used as the access point to CUDD library that provides very extensive and confusing API. `CUDDFacade` provides a clean and type-safe interface with only those operations which are necessary for the implementation of the library, while hiding the others. The class is compiled with all CUDD's object files into a single static library which is then further used by the tree automata library.

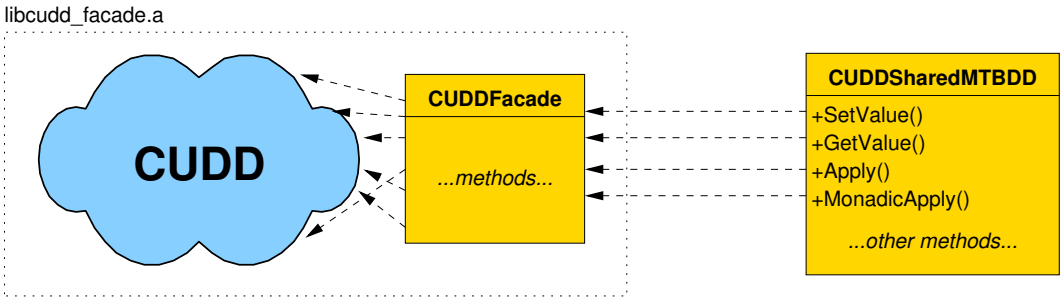


Figure 6.3: Interface to CUDD package.

`CUDDSharedMTBDD` is an object-oriented representation of a shared MTBDD as described in Section 6.1. The class uses `CUDDFacade` to access CUDD as depicted in Figure 6.3. `CUDDSharedMTBDD` is a class template with the following template parameters:

`RootType` Defines the type for accessing MTBDDs for individual super-states. This may be an arbitrary type, the prototype implementation uses `unsigned`.

`LeafType` The type for the sink node of the MTBDD. This may again be an arbitrary type, the prototype implementation uses a set of states. However, deterministic automata may store the target state directly in the sink node thus making the access time shorter.

`VariableAssignmentType` This template parameter determines the data type for representation of assignment to Boolean variables of the MTBDD, i.e. the data type for the symbol. To fully utilise the potential of MTBDDs, our representation of assignment to Boolean variables of MTBDD can assign 3 different values to a variable: `true` (`1`), `false` (`0`) and `don't care` (`X`). By using the `don't care` value, we may work with whole sets of transitions over various symbols as with a single transition (for example to work with four transitions over symbols encoded as `100`, `101`, `110`, and `111` at once, it is enough to use only one encoding `1XX`).

`RootAllocator` The implementation of this policy defines the mapping of roots of `RootType` to CUDD-related pointers to root nodes of corresponding MTBDDs.

`LeafAllocator` This policy determines the exact implementation of the mapping between the `unsigned` sink nodes stored in the patched CUDD data structures and those of type `LeafType`.

These template parameters allow high configurability of `CUDDSharedMTBDD`, for instance to be used for deterministic finite tree automata or for finite (word) automata transition functions.

`CUDDSharedMTBDD` also defines `AbstractApplyFunctorType` which is an abstract class of a function object with a single pure virtual method, which is overloaded operator `()`:

```
virtual LeafType operator()(const LeafType& lhs, const LeafType& rhs) = 0;
```

Classes that inherit this abstract function object need to implement the only method by defining a function for *Apply* operation. The *Apply* operation takes root nodes of two MTBDDs and an object of a class that implements the `AbstractApplyFunctorType` interface:

```
RootType Apply(const RootType& lhs, const RootType& rhs,
               AbstractApplyFunctorType* op);
```

6.2.2 Transition Function

The `MTBDDTransitionFunction` class represents transition functions of several automata using single MTBDD. This is because CUDD only allows executing *Apply* operation on MTBDD roots from the same shared MTBDD. When an automaton is being created, it *registers* to some `MTBDDTransitionFunction` and inserts all its transitions into this object.

A challenging issue that needs to be faced is the choice of data structure for storage of super-states, i.e. mapping of super-states to their respective MTBDDs. Storage of nullary and unary super-states is obvious. Since there is only one nullary super-state for each automaton, these super-states are stored in a single array indexed by automaton number. Unary super-states of each automaton are stored in a separate array indexed by the only state's number. The prototype implementation also deals with storage of binary super-states by using a 2-dimensional matrix indexed by the two states of the super-state. Due to the fact that space requirements for n-dimensional matrix grow exponentially and the utilisation drops with almost the same speed for real-world problems, more sophisticated data structures need to be found. Our prototype implementation uses for storage of super-states with arity greater than 2 a hash table with an arbitrarily long vector of states as the key and pointer to MTBDD as the value.

6.2.3 Tree Automaton

The `TreeAutomaton` class represents a finite tree automaton with a high-level interface. The interface allows the use of human-readable names of states and symbols and provides mapping to their inner representation. `TreeAutomaton` enables adding a state, transition or marking a state as final. It also supports importing and exporting a finite tree automaton to or from a file.

6.2.4 Automaton Import

In order to support direct user interface to the library, the library supports reading a finite tree automaton from a file. The reading interface is designed according to the builder

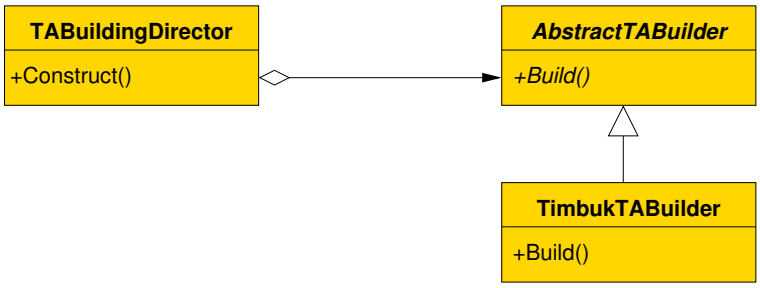


Figure 6.4: `TABuildingDirector` structure.

design pattern [33]. Building a new finite tree automaton from a file is done by creating an object of class `TABuildingDirector` and assigning an instance of class implementing the `AbstractTABuilder` interface to it. Then calling the `Construct()` method of `TABuildingDirector` (which further calls the `Build()` method of `AbstractTABuilder`) with a data stream that has a format recognized by the concrete builder constructs proper tree automaton. For testing purposes one concrete builder was implemented: `TimbukTABuilder` which accepts input data stream with description of automata in Timbuk-like format (see Section 3.1).

6.2.5 Automaton Export

This section deals with exporting description of a tree automaton into a human-readable format. In order to do so, the most difficult task is to extract transitions from symbolic representation into explicit. This operation needs to know the structure of the MTBDD used for representation of the transition function. This is achieved by using *test symbols* $(x_1, \ldots, x_n)$ that start with value `XXX...X` and for each Boolean variable $x_1, \ldots, x_n$ attempt to bind its value to both `0` and `1`. If it holds that the resulting MTBDDs for both bindings are the same, then the value of the variable is not important, it is left in `X` (`don't care`) and the procedure carries on to the following variables. In case the MTBDDs are not the same, the procedure splits into two branches and continues for both bindings. This continues until all variables have been either bound or left in `X`.

A simple script that converts a file in the output format into a graphical representation of the automaton for `dot` tool [34] has been created.

Unlike finite word automata, transition function and run of a finite tree automaton cannot be expressed simply as a labelled graph and *walk* (i.e. sequence of vertices and edges such that each vertex or edge may occur several times in the sequence) in the graph. Up to our knowledge there is no widely accepted standard graphical representation of finite tree automata, we therefore attempted to choose a simple and understandable representation resembling finite (word) automata as much as possible. As in finite (word) automata, states are represented by circles, final states by double circles. We introduce new type of vertices representing super-states which make the graph bipartite: an edge is either from a state to a super-state or from a super-state to a state (this may resemble a Petri net). Representation of super-states is by rectangles with *boxes*. The number of boxes determines the arity of the super-state. The labelling of edges from states to super-states denotes the position of the state in the super-state vector; the labelling of edges from super-states to states denotes the symbol over which the transition is to be made. An example of graphical representation of

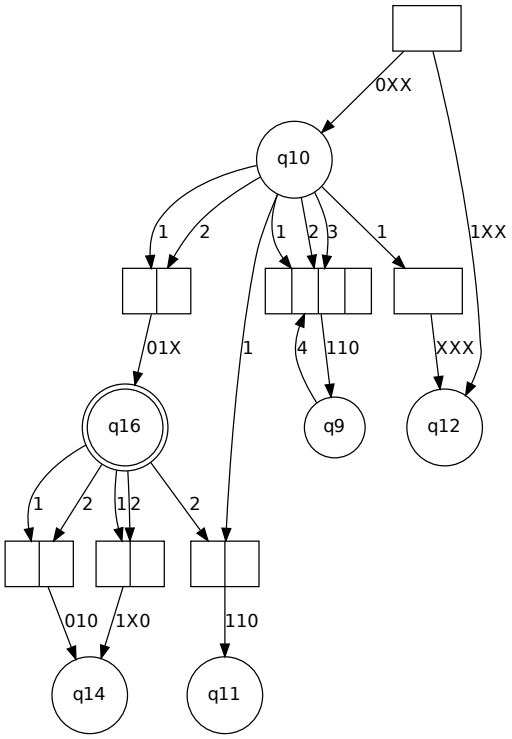


Figure 6.5: Example of graphical representation of a tree automaton.

a finite tree automaton is in Figure 6.5.

6.2.6 Operations

Operations on finite tree automata with transition function represented using an MTBDD are provided by `MTBDDOperation` class. The prototype implementation implements the following operations: language union, language intersection, reduction of an automaton according to some equivalence class, and computation of downward simulation. All these operations are performed only using the interface provided by `CUDDSharedMTBDD`.

Chapter 7

Evaluation

This chapter provides an evaluation of the prototype implementation (called libSFTA) of the library described in the previous chapters. The tests were run on a laptop with a dual-core Intel Core 2 Duo CPU at 1.80 GHz and 2 GiB of available memory with Debian Squeeze GNU/Linux installed. We measured performance of the following three finite tree automata operations: language union, language intersection, and automaton reduction according to the downward simulation relation.

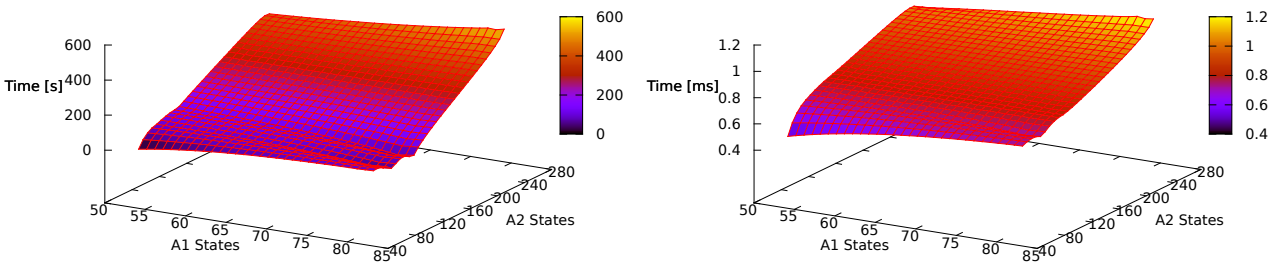
7.1 Language Union

The performance of the language union operation on two input finite tree automata was measured and compared to Timbuk, a tree automata library (described in Section 3.1) that performs operations on nondeterministic finite tree automata using an *explicit* representation of the transition function (note that the implemented library uses a *symbolic* representation). We made this choice because MONA immediately determinises input automata so the comparison would not be fair.

We performed the tests on binary tree automata over an alphabet with 130 symbols and various size of the state set obtained from tree model checking of real systems. It should be pointed out that the execution time for both libSFTA and Timbuk does not include the time necessary to load the automaton from a file. This should give more valid results, since building an MTBDD for a transition function is not a trivial operation (note that the comparison is fair from a practical point of view since within a verification framework automata are built internally, not loaded from a file). The results are given in Table 7.1 (in a graphical form in Figure 7.1). It can be seen that libSFTA significantly outperforms Timbuk in all cases.

Automata		Timbuk	libSFTA
A0053	A0054	1.982 s	0.0005 s
A0080	A0082	37.645 s	0.0007 s
A0080	A0111	37.645 s	0.0008 s
A0053	A0246	414.104 s	0.0010 s
A0080	A0246	533.678 s	0.0012 s
A0082	A0246	542.069 s	0.0012 s

Table 7.1: Language union performance results.



a) Timbuk b) libSFTA

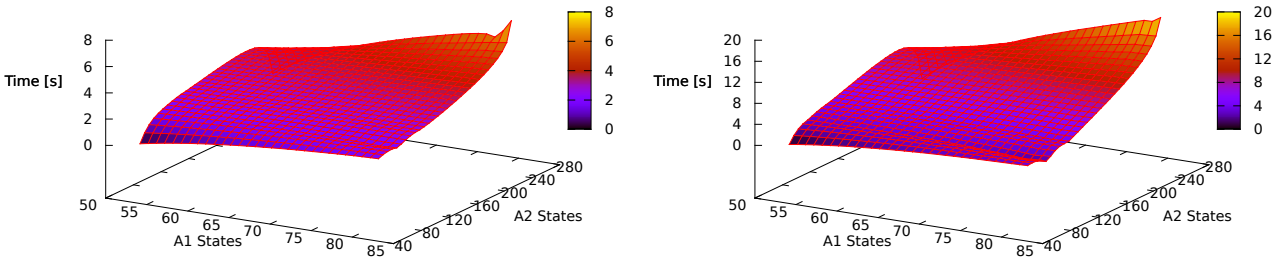
Figure 7.1: Performance comparison of language union for various state set size.

7.2 Language Intersection

The experiments with the language intersection operation on two input finite tree automata were also compared to Timbuk. These experiments were conducted with the same set of test automata and the same testing conditions as the language union operation (see Section 7.1). The results are given in Table 7.2 with the graph in Figure 7.2. The results show that for larger state sets Timbuk computes the finite tree automaton for language intersection slightly faster than libSFTA.

Automata		Timbuk	libSFTA
A0053	A0054	0.076 s	0.057 s
A0053	A0246	0.609 s	0.617 s
A0080	A0082	1.862 s	1.675 s
A0080	A0111	2.483 s	3.765 s
A0080	A0246	6.062 s	18.320 s
A0082	A0246	7.503 s	19.355 s

Table 7.2: Language intersection performance results.



a) Timbuk b) libSFTA

Figure 7.2: Performance comparison of language intersection for various state set size.

7.3 Simulation Reduction

The performance of reduction of a finite tree automaton according to downward simulation relation was measured in the next series of tests. The results were compared to SA [35], an OCaml tool implementing computation of downward simulation over labelled transition systems and finite tree automata.

The execution time for libSFTA comprises computing downward simulation over the input tree automaton, computing symmetric closure of the relation and reducing the automaton according to equivalence given by the simulation and its symmetric closure. As SA cannot perform reduction of the automaton, the execution time of SA consists of the time it takes to load the automaton from a file (which should be negligible according to the authors) and compute the downward simulation. Two different test cases were measured.

1. The first test case shows how the performance depends on the size of the *state set* of the input finite tree automaton for a fixed small alphabet. The results are given in Table 7.3 and Figure 7.3. We can see that the performance of libSFTA is worse when compared to SA. The reason for this is that SA uses a more sophisticated algorithm for computation of simulation. In the future version of the library, we wish to focus on optimising the algorithm we use in order to be able to compete with the solution used in SA even for smaller alphabets.

Automaton	States	Transitions	SA	libSFTA
A0053	53	159	0.04 s	24.6 s
A0054	54	241	0.04 s	29.3 s
A0063	63	571	0.10 s	55.2 s
A0070	70	622	0.07 s	71.5 s
A0080	80	672	0.11 s	274.4 s
A0082	82	713	0.09 s	331.5 s
A0089	89	1006	0.11 s	226.1 s

Table 7.3: Experimental results of simulation reduction for various state set size.

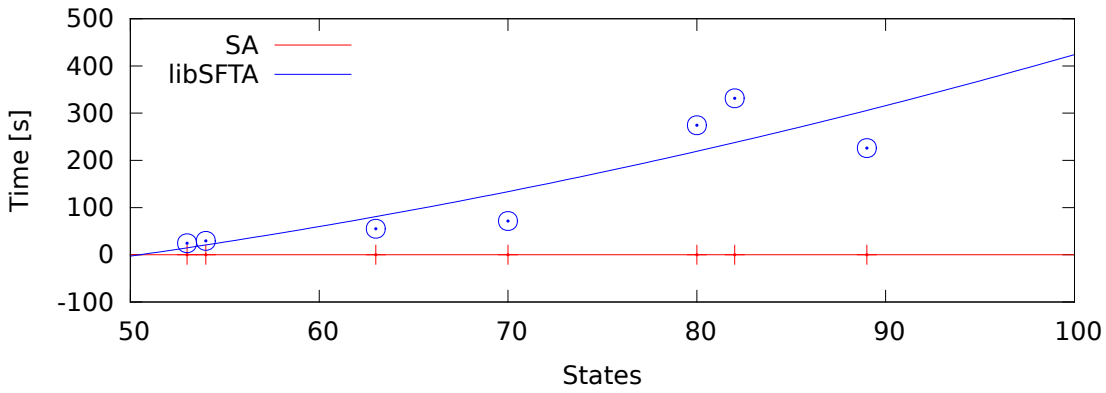


Figure 7.3: Performance comparison of simulation reduction for various state set size.

2. The second test case shows how the performance of libSFTA and SA relates to the size of the alphabet of the input finite tree automaton for a fixed set of states. We created a set of simple tree automata that perform transitions over symbols from alphabet of various size. The number of transitions equals the number of symbols in the alphabet (we use one transition for every symbol). The results of the experiments with the size of the alphabet are in Table 7.4 and Figure 7.4. It is clear that the use of symbolic representation makes the performance of libSFTA far superior to the performance of SA.

Symbols	SA	libSFTA
1337	0.06 s	0.0033 s
3525	0.14 s	0.0051 s
7067	0.26 s	0.0071 s
15136	0.69 s	0.0054 s
31235	2.09 s	0.0031 s
65503	8.86 s	0.0040 s
130023	48.40 s	0.0045 s

Table 7.4: Experimental results of simulation reduction for various alphabet size.

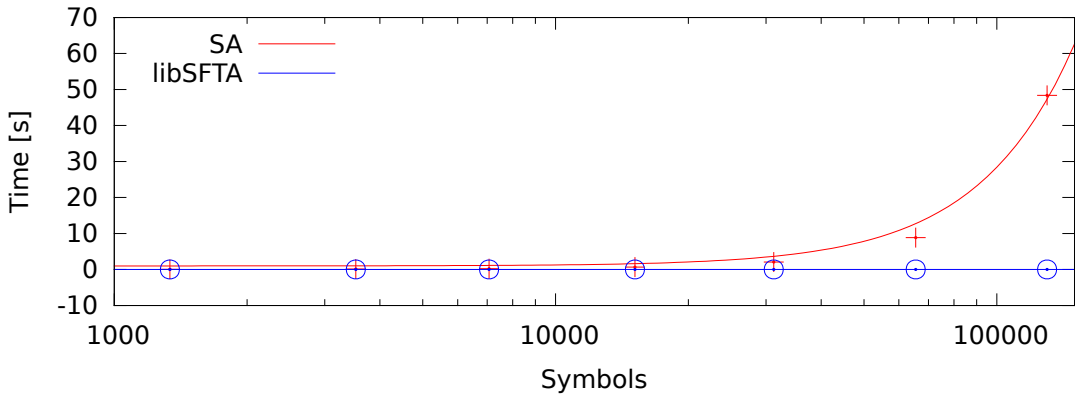


Figure 7.4: Performance comparison of simulation reduction for various alphabet size.

7.4 Discussion

The experiments described in this chapter showed that libSFTA has a good potential to become an interesting tree automata library, especially for applications that need large alphabets and can exploit symbolic representation well. The use of nondeterminism also considerably accelerates the computation of the automaton for language union and can keep automata small and clean.

Chapter 8

Conclusion

The aim of this Master's Thesis was to design and implement an efficient and flexible finite tree automata library for the use in symbolic formal verification, namely to be applicable for tree model checking techniques, such as regular tree model checking and abstract regular tree model checking.

The theoretical background of tree automata has been studied as well as formal verification methods for systems represented by tree automata. Existing packages that implement tree automata have been surveyed and their advantages and disadvantages summarised. An analysis of the aforementioned verification methods have yielded a list of necessary requirements for the library.

A representation of nondeterministic finite tree automata with symbolically represented transition functions has been proposed. The representation is based on MTBDDs provided by an external package (which may be changed though by writing a simple wrapper with given interface for another library). Algorithms that carry out standard as well as some verification-specific operations on tree automata using this representation have also been developed and described in this text.

An object-oriented modular design of the library based on design patterns and policies has been created. A prototype implementation has been programmed, evaluated on testing data, and compared to other tools that provide the same functionality. The results of experiments show that the concepts we employed are viable and that the library can complement currently available tree automata libraries, especially when used for applications with finite tree automata that make use of large alphabets and nondeterminism.

Future work will focus on redesigning the library according to our feedback from the implementation of the prototype and data collected from code profiling. Further, we wish to create a library that supports both explicitly and symbolically represented finite tree automata. We also plan to optimise the algorithms that are used by the library in order to give good performance even for small alphabets and large state sets. Another direction of work then includes implementing a support of further algorithms for simulation reduction (upward and combined simulation based relations) and antichain-based inclusion checking (combination of antichains and simulations), including further research on still more advanced reduction and inclusion checking techniques. A next step should then be to test the library as a part of various verification tools (for instance as a base of abstract regular tree model checking tools or with decision procedures for various logics, such as WS2S or separation logic).

Bibliography

[1] Donald Ervin Knuth. *Lattices of Trees, Part I.* Computer Musings [Videorecording]. Stanford University, 1997. Available at URL: `http://stanford-online.stanford.edu/seminars/knuth/971203-knuth-100.asx` (May 2010).

[2] Alexander Meduna. *Automata and Languages: Theory and Applications.* Springer Verlag, 2005.

[3] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, second edition, 1998.

[4] Bashar Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, May 1997. Available at URL: `http://www.doc.ic.ac.uk/~ban/pubs/ariane5.pdf` (May 2010).

[5] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *12th International Conference on Computer Aided Verification*, July 2000. Available at URL: `http://www.liafa.jussieu.fr/~abou/rmc-cav00.ps.gz` (May 2010).

[6] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *13th International Static Analysis Symposium*, 2006.

[7] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In *14th International Conference on Computer Aided Verification*, 2002. Available at URL: `http://user.it.uu.se/~parosh/publications/papers/trees.ps` (May 2010).

[8] Hubert Comon, Max Dauchet, Remi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications.* 2007. Available at URL: `http://www.grappa.univ-lille3.fr/tata` (May 2010).

[9] Timbuk: Web pages of Timbuk. URL: `http://www.irisa.fr/celtique/genet/timbuk/` (May 2010).

[10] GNU Library General Public Licence, version 2.0: Web pages of GNU Project. URL: `http://www.gnu.org/licenses/old-licenses/lgpl-2.0.html` (May 2010).

[11] MONA: Web pages of MONA. URL: `http://www.brics.dk/mona/` (May 2010).

[12] GNU General Public Licence, version 2: Web pages of GNU Project. URL: `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html` (May 2010).

[13] Julius Richard Büchi. Weak second-order arithmetic and finite automata. Technical report, The University of Michigan, 1959. Available at URL: `http://hdl.handle.net/2027.42/3930` (May 2010).

[14] Ralf Treinen. Predicate logic and tree automata with tests. In *Foundations of Software Science and Computation Structures*, pages 329–343, 2000. Available at URL: `http://www.pps.jussieu.fr/~treinen/publi/fossacs2000.ps.gz` (May 2010).

[15] James W. Thatcher and Jesse B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[16] James Glenn and William Gasarch. Implementing WS1S via finite automata. In *Workshop on Implementing Automata*, 1996. Available at URL: `www.cs.umd.edu/~gasarch/papers/wia96.pdf` (May 2010).

[17] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. Available at URL: `http://www.brics.dk/mona/papers/implementation-secrets/journal.pdf` (May 2010).

[18] Donald Ervin Knuth. *Combinatorial Algorithms*, volume 4 of *The Art of Computer Programming*, chapter Binary Decision Diagrams. A draft of this text is available at URL: `www-cs-faculty.stanford.edu/~knuth/fasc1b.ps.gz` (May 2010).

[19] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *Workshop on Implementing Automata*, 1996. Available at URL: `http://www.brics.dk/mona/papers/algorithms-guided-tree-aut/article2.pdf` (May 2010).

[20] Ahmed Bouajjani, Peter Habermehl, Lukáš Holík, Tayssir Touili, and Tomáš Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *13th International Conference on Implementation and Application of Automata*, July 2008. An extended version is available at URL: `http://www.fit.vutbr.cz/~vojnar/Publications/bhhtv-nartmc-tr-08.pdf` (May 2010).

[21] Lethal: Web pages of Lethal. URL: `http://lethal.sourceforge.net/` (May 2010).

[22] Binary Tree Automata Library: Web pages of Binary Tree Automata Library. URL: `http://www.grappa.univ-lille3.fr/~filiot/tata/` (May 2010).

[23] ELAN — Tree Automata Library: Web pages of ELAN — Tree Automata Library. URL: `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/examples/elan-automata.html` (May 2010).

[24] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking. In *7th International Workshop on Verification of Infinite-State Systems*, August 2006. Available at URL: `http://www.liafa.jussieu.fr/~haberm/Publications/bhrvinfinity05.ps.gz` (May 2010).

[25] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains (on checking language inclusion of nondeterministic finite (tree) automata). In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2010. An extended version is available at URL: `http://www.fit.vutbr.cz/~holik/pub/FIT-TR-2010-001.pdf` (May 2010).

[26] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. Computing simulations over tree automata (efficient techniques for reducing tree automata). In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008. An extended version is available at URL: `http://www.fit.vutbr.cz/~vojnar/Publications/abhkv-simtree-tr-07.pdf` (May 2010).

[27] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. Composed bisimulation for tree automata. In *13th International Conference on Implementation and Application of Automata*, 2008. An extended version is available at URL: `http://www.fit.vutbr.cz/~holik/pub/FIT-TR-2008-004.pdf` (May 2010).

[28] Lukáš Holík and Jiří Šimáček. Optimizing an LTS-simulation algorithm. In *5th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, 2009. Available at URL: `http://www.fit.vutbr.cz/~holik/pub/FIT-TR-2009-03.pdf` (May 2010).

[29] CUDD: Web pages of CU Decision Diagram Package. URL: `http://vlsi.colorado.edu/~fabio/CUDD/` (May 2010).

[30] New and simplified BSD licence: Web pages of Open Source Initiative. URL: `http://www.opensource.org/licenses/bsd-license.php` (May 2010).

[31] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, 1993. Available at URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.2128` (May 2010).

[32] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley, 2001.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[34] Graphviz: Web pages of Graphviz. URL: `http://www.graphviz.org/` (May 2010).

[35] SA Tool: Web pages of SA Tool. URL: `http://www.fit.vutbr.cz/research/groups/verifit/tools/sa/` (May 2010).

Appendix A

Storage Medium

A storage medium (DVD) containing an electronic version of the technical report and source code of the prototype implementation including patched CUDD package is enclosed to this thesis.