BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

IMPROVEMENTS OF THE ASMA TOOL FOR ANALYSIS OF STRING MANIPULATING PROGRAMS VIA SYMBOLIC AUTOMATA

ROZVOJ NÁSTROJE ASMA PRO ANALÝZU PROGRAMŮ S ŘETĚZCI POMOCÍ SYMBOLICKÝCH AUTOMATŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MARTIN KMENTA**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. VOJNAR TOMÁŠ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Bachelor's Thesis Specification

Student: **Kmenta Martin**
Programme: Information Technology
Title: **Improvements of the ASMA Tool for Analysis of String Manipulating Programs via Symbolic Automata**
Category: Formal Verification

Assignment:

1. Get acquainted with symbolic automata and transducers, the approach of abstract regular model checking (ARMC) and with possibilities of its application in verification of string manipulating programs.
2. Familiarise yourself with the ASMA tool prototypically implementing ARMC for verification of string manipulating programs over the Microsoft AutomataDotNet library of symbolic automata.
3. Through experiments as well as studying of the code of ASMA identify weaknesses of its current version, e.g., in the way various special operations on symbolic automata are implemented or in the automata reduction methods used.
4. Propose and implement improvements of the current version of ASMA, with a stress on correctness of the implementation, its efficiency, and maintainability.
5. Validate the new versions experimentally and document the impact of your changes.
6. Summarise the achieved results and discuss possible further enhancements of ASMA.

Recommended literature:

- Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular (Tree) Model Checking. International Journal on Software Tools for Technology Transfer (STTT), 14(2):167--191, Springer-Verlag, 2012.
- Bultan, T., Yu, F., Alkhalaf, M., Aydin, A.: String Analysis for Software Verification and Security. Springer, 2017.
- D'Antoni, L., Veanes, M.: Extended Symbolic Finite Automata and Transducers. Formal Methods in System Design, 47(1): 93-119, Springer, 2015.
- Veanes, M.: The Automata .NET Library, Microsoft, https://github.com/AutomataDotNet.
- Kotoun, M.: Symbolic Automata for Analysing String Manipulating Programs. Master thesis, FIT, Brno University of Technology, 2019.
- Síč, J.: Simulation for Symbolic Automata. Bachelor thesis, FIT, Brno University of Technology, 2017.

Requirements for the first semester:

- The first three items and at least beginning of work on the fourth item.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 11, 2022
Approval date: November 3, 2021

Abstract

In this work we deal with regular model checking which is a technique for analyzing programs whose state space can be infinite due to dealing with, e.g. unbounded queues, parameters, dynamically linked data structures, recursive procedures, or strings. The goal of this work was to implement improvements to the existing prototype tool ASMA implementing regular model checking over the Microsoft Automata library. We analysed the source code of ASMA and reran analyses of all available benchmark programs. We identified some bottlenecks and have tackled several of them. In particular, we integrated a library containing additional reduction algorithms into ASMA, created several new versions of the reverse concatenation operation, which tuned out to be very costly in the benchmarks, improved the command line interface of ASMA, and implemented some other optimizations for ASMA. The computation time was reduced by 90 % when analysing bigger programs.

Abstrakt

V této práci se zabýváme regulárním model checkingem, což je technika pro analýzu programů, jejíchž stavový prostor může být nekonečný v důsledku práce například s neomezenými frontami, parametry, dynamicky propojenými datovými strukturami, rekurzivními procedurami nebo řetězci. Cílem této práce bylo implementovat vylepšení stávajícího prototypu nástroje ASMA implementujícího regulárním model checking nad knihovnou Automata of Microsoftu. Provedli jsme analýzu zdrojového kódu nástroje ASMA a zopakovaly analýzy všech dostupných srovnávacích programů. Identifikovali jsme některá úzká místa a několik z nich jsme vyřešili. Zejména jsme integrovali knihovnu obsahující další redukční algoritmy do nástroje ASMA, vytvořili několik nových verzí operace reverzní konkatenace, která se v benchmarcích ukázala jako velmi nákladná, vylepšili rozhraní příkazového řádku ASMA a implementovali některé další optimalizace. Výpočetní čas se při analýze větších programů snížil o 90 %.

Keywords

regular model checking, RMC, abstract regular model checking, ARMC, finite automata, transducers, symbolic finite automata, AutomataDotNet, ASMA, VeriFIT, reverse concatenation

Klíčová slova

regulární model checking, RMC, abstraktní regulární model checking, ARMC, konečné automaty, převodníky, symbolické konečné automaty, AutomataDotNet, ASMA, VeriFIT, reverzní konkatenace

Reference

KMENTA, Martin. *Improvements of the ASMA Tool for Analysis of String Manipulating Programs via Symbolic Automata.* Brno, 2022. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Vojnar Tomáš, Ph.D.

Rozšířený abstrakt

Jak zajistit vysokou spolehlivost programu? Běžným způsobem, jak ověřit, že program neselže, je jeho testování. Tato metoda ale nemůže prokázat 100 % spolehlivost testovaného softwaru. Způsob, jak vytvořit vysoce spolehlivý software, spočívá v použití metod formální verifikace a analýzy, které jsou schopny spolehlivě odhalit chyby a zajistit tak vysokou spolehlivost programu.

Obtížnou výzvou ve formální analýze je práce s nekonečně stavovými systémy. Regulární model checking (RMC) je technika pro analýzu takových systémů, kde nekonečně stavový prostor může vzniknout např. v důsledku práce s neomezenými frontami, parametry, dynamicky propojenými datovými strukturami, rekurzivními procedurami nebo řetězci. Poslední zmíněná vlastnost je pro tuto práci zvláště zajímavá. Programy manipulující s řetězci si zaslouží zvláštní pozornost např. kvůli nebezpečí Cross-site scripting útoků (XSS). Tyto útoky umožňují útočníkům vložit skripty do webových stránek používaných jinými uživateli.

Přestože je RMC poměrně účinný, neexistuje žádná vhodná implementace RMC, která by obsahovala všechny nejmodernější výsledky z oblasti automatů. Navíc nemáme k dispozici žádnou knihovnu pro práci s konečnými automaty (FA), která by poskytovala implementaci řetězcových operací, používaných v programech pracujících s řetězci nad FA, která je potřebná pro implementaci RMC na takových programech. Máme pouze jeden prototyp takové knihovny, kterým je nástroj ASMA implementovaný Michalem Kotounem v rámci jeho diplomové práce [7].

V současné době existující přístupy, jak analyzovat programy s řetězci (jiné než přístup nástroje ASMA založený na RMC), buď vyžadují spolupráci s uživateli při vytváření nějakých verifikačních podmínek, invariantů apod. nebo jsou nepřesné. Zejména jsou nepřesné buď ve smyslu hlášení falešných chyb, nebo jsou nastaveny tak, aby hlásily méně, což může vést k vynechání skutečných chyb. Můžeme tedy použít přístup, který vyžaduje nezanedbatelnou lidskou pomoc, nebo nepřesnou plně automatickou statickou analýzu, a nebo zvolit něco uprostřed, což může být právě RMC.

Obecným cílem této práce je vylepšit nástroj ASMA. Konkrétněji bylo prvním cílem se seznámit s danou oblastí, pochopit pokročilé algoritmy pracující s automaty a seznámit se s nástrojem ASMA, který využívá knihovnu Microsoft Automata, vyvinutou Margusem Veanesem. Druhým cílem bylo identifikovat slabá místa nástroje ASMA a následně navrhnout a implementovat zlepšení s důrazem na správnost implementace, její efektivitu a udržovatelnost.

V rámci plnění našich cílů jsme postupovali následovně: analyzovali jsme zdrojový kód nástroje ASMA a provedli analýzu všech dostupných příkladů programů, na nichž byla ASMA původně testována. Pomocí profilovacího nástroje jsme zanalyzovali hlavní úzká místa při běhu nástroje ASMA na všech příkladech. Ukázalo se, že hlavními částmi zpomalujícími nástroj ASMA byly reverzní konkatenace vytvářející obrovské množství epsilonových přechodů a redukční algoritmy zabírající příliš mnoho prostředků a času na výpočet.

Naší reakcí na zmíněnou neefektivitu redukčních algoritmů v ASMA byla integrace knihovny AutomatonSimulation od Juraje Síče do nástroje ASMA. Tato knihovna obsahuje metody pro výpočet simulací a pro redukci automatů pomocí těchto simulací. Tyto si-

mulační redukce nám umožňují pracovat s nedeterministickými automaty bez nutnosti je čas od času determinizovat. Tento krok si vyžádal určité změny v architektuře ASMA, které nám umožňují volit mezi různými redukčními algoritmy. Po integraci jsme museli také aktualizovat systém logování a příkazy v AsmaCLI. Výsledkem je, že si uživatel nyní může libovolně zvolit, která redukce má být použita jak během běhu ARMC (pro všechny počítané automaty), tak na daném automatu v AsmaCLI. Přidali jsme také možnost, aby si uživatelé mohli po každém kroku analýzy prohlédnout náhled automatů bez náročné práce.

Naším dalším krokem bylo porovnání běhu analýzy všech příkladů se všemi dostupnými redukcemi. To vedlo k vytvoření vlastního testovacího příkazu v prostředí ASMA, který spustí analýzu pro všechny zadané programy se všemi zadanými typy redukcí. Výsledné porovnání redukčních algoritmů nám bohužel ukázalo, že tyto nové redukční algoritmy jsou pro všechny analyzované příklady příliš pomalé. Mohou však mít potenciál do budoucna. Provedli jsme ale také několik drobných optimalizací, jako je přidání dodatečného ukládání výsledků do mezi-paměti v kritických částech a odstranění zbytečných částí kódu nebo použití redukcí na vhodných místech. Tyto optimalizace pak měly poměrně výrazný vliv na rychlost analýzy programů, a to i při použití staršího přístupu založeného na determinizaci a minimalizaci.

Posledním a nejdůležitějším výsledkem této práce je vytvoření optimalizované operace reverzní konkatenace. Vytvořili jsme čtyři prototypy s postupně rostoucím vývojem. Celkově jsme zkrátili dobu běhu o více než 90 % u analýzy programů, u nichž se vyskytují větší automaty.

Zbytek práce je strukturován následovně: V kapitole 2 uvedeme základní pojmy automatů a převodníků, které jsou nezbytné pro pochopení ARMC. Kapitola 3 obsahuje informace o nástroji ASMA a popis našich kroků při jeho analýze. V kapitole 4 jsme navrhli některá vylepšení a v kapitole 5 jsme popsali pokrok ve vývoji nástroje ASMA. Na konci této práce je v kapitole 6 uveden závěr s možnými budoucími kroky pro další rozvoj ASMA.

Improvements of the ASMA Tool for Analysis of String Manipulating Programs via Symbolic Automata

Declaration

I declare that I have prepared this report independently under the guidance of prof. Tomas Vojnar. More information was provided by Mr. Michal Kotoun and Mr. Juraj Síč. I listed all the literary sources, publications and other sources from which I drew.

.....
Martin Kmenta
May 10, 2022

Acknowledgements

I would like to thank my supervisor prof. Tomáš V. for his consistent support and guidance during the running of this project. Furthermore I would like to thank to Michal K. and Juraj S. for their help.

Contents

Chapter 1

Introduction

How to ensure high reliability of a program? A common way to check that a program will not crash is to test it. However, this method cannot fully prove absence of bugs in tested software. The way to create highly reliable software is to use formal verification and analysis methods that are capable of reliably detecting errors and thus ensure the high reliability of the program.

However, a hard challenge in formal analysis is dealing with infinite-state systems. Regular model checking (RMC) is a technique for analyzing such systems where the infinite-state space can arise due to dealing with, e.g. unbounded queues, parameters, dynamically linked data structures, recursive procedures, or strings. The last-mentioned feature is of particular interest for this work, because it concentrates on programs handling strings. Programs manipulating with strings deserve special attention, e.g., because of the danger of attacks with Cross-site scripting (XSS). These attacks allow attackers to inject scripts into web pages used by other users.

Although RMC is quite powerful, there is no proper implementation of RMC that would contain all state-of-the-art results from the area of automata. Moreover, we do not have any solid finite automata (FA) library that would provide an implementation of string operations, used in string programs, over FA, which is needed for implementing RMC on such programs. We have only one prototype of such a library, which is the ASMA tool implemented by Michal Kotoun for his master thesis [7].

Currently existing approaches how to analyze programs with strings (other than the RMC--based approach of ASMA) are either requiring cooperation with users to create some verification conditions, invariants, etc. or they are inaccurate. In particular, they are inaccurate either in the sense of reporting false errors or they are set to report less which can result in skipping real errors. So, we can use an approach that requires a non-negligible human help, perform inaccurate fully automatic static analysis, or go for something in the middle which can hopefully be RMC.

The general aim of this work is to improve the ASMA tool. More concretely the first goal of this work was to learn about the area, to understand algorithms working with automata, and to get acquainted with the ASMA tool, which uses the Microsoft Automata library, developed by Margus Veanes. The second goal was to identify weaknesses of ASMA and then propose and implement improvements, with a stress on the correctness of the implementation, its efficiency, and maintainability.

As part of the fulfillment of our goals, we proceeded as follows: we analysed the source code of ASMA and ran analyses of all available program examples. Using a profiling tool, we analysed major bottlenecks in the run of ASMA on all examples. It turned out, that the major parts slowing the ASMA tool were the reverse concatenation creating an enormous number of epsilon transitions, and reduction algorithms taking too many resources and time to compute.

Responding to the mentioned inefficiency of the reduction algorithms in ASMA, one of the results of this work is the integration of the AutomatonSimulation library from Juraj Síč into the ASMA tool. This library contains methods for computing simulations for reducing automata using these simulations. These simulations allow us to work with non-deterministic automata. The introduction of the simulation reductions required some changes in the architecture of ASMA, which allow us to choose between these algorithms and the classical approach based on determinization and minimisation as well as using bisimulation-based reductions present already in ASMA. After the integration, we had to upgrade the logging system and AsmaCLI commands. The result is that the user can now freely choose which reduction should be used both during an ARMC run (for all counted automata) as well as on a given automaton in the AsmaCLI. We also added the possibility for users to preview automata after each step of the analysis without hard work.

Our next step was to benchmark all examples with all available reductions. This led to creating a custom testing command within the ASMA environment which can run the analysis for all given programs with all specified reduction types. Unfortunately, the resulting comparison of the reduction algorithms showed us that these new reduction algorithms were too slow for all examples we analyzed. However, they may have potential in the future. We have also performed some minor optimizations such as adding some extra caching of results in critical parts and removal of useless parts of the code or using reductions in proper places. These optimizations had quite a significant impact on the speed of the program analysis even when using the classical approach based on determinization and minimisation.

The last and the most important result of this work is the creation of an optimized reverse concatenation operation. We created four prototypes with gradually improving efficiency. Altogether, we reduced the running time by over 90 % for analysis of programs where bigger automata take their place.

The rest of the thesis is structured as follows: In Chapter 2 we will introduce the basic concepts of automata and transducers which are necessary for understanding the abstract regular model checking. Chapter 3 contains information about the ASMA tool and a description of our steps during its analysis. We propose our improvements of ASMA in Chapter 4 and we described progress in the development of the ASMA tool in Chapter 5. At the end of this thesis there is a conclusion with future steps in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we introduce the basics of both classical FAs as well as SAs and FTs. This chapter is based on [8, 5, 7, 9, 10].

2.1 Alphabets, Strings, and Languages

An alphabet Σ is a finite, nonempty set of elements, which are called symbols. An example of an alphabet containing symbols a, b, c, d is $\Sigma = \{a, b, c, d\}$.

ϵ is an empty string over Σ, $\epsilon \notin \Sigma$, which contains no symbols. For $a \in \Sigma$ and $x, y \in \Sigma^*$, we define $(ax).y = a(x.y)$, and we set $(\epsilon.x = x.\epsilon = x)$ for any $x \in \Sigma^*$. If x is a string over Σ and $s \in \Sigma$, then xs is a string over Σ. The length of a string x denoted as $|x|$ is 0 for $x = \epsilon$ or $x = ax'$ for some $a \in \Sigma \wedge x' \in \Sigma^*$ and $|x| = 1 + |x'|$. The power of a string x is denoted as x^i where i is the power number. If $i = 0$, then $x^0 = \epsilon$, else $x^i = xx^{i-1}$. The reverse string $reversal(x)$ is defined such that $reversal(\epsilon) = \epsilon$, and $reversal(s_1 \ldots s_n) = s_n \ldots s_1$ where $s_1 \ldots s_n \in \Sigma, n \geq 1$.

Let Σ^* denote the language of all strings over Σ and Σ^+ the language of all non-empty strings over Σ. $(\Sigma^+ = \Sigma^* - \{\epsilon\})$ Then every subset $L \subseteq \Sigma^*$ is a language over Σ. A finite language contains a finite number of strings, otherwise it is infinite. Special languages include $L_1 = \oslash$ (the finite empty language) and $L_2 = \{\epsilon\}$ (the finite language containing empty string.) Let L_3, L_4 be languages over Σ. Their concatenation is $L_3.L_4 = \{xy | x \in L_3, y \in L_4\}$, and their difference is $L_3 \setminus L_4 = \{x | x \in L_3, x \notin L_4\}$.

2.2 Regular Expressions (REs)

Regular expressions over Σ and the languages they denote are defined as follows:

- $\oslash$ is an RE denoting $L = \oslash$,
- ϵ is an RE denoting $L = \{\epsilon\}$,
- $a, a \in \Sigma$ is an RE denoting $L = \{a\}$.
- Let e_1 and e_2 be regular expressions denoting L_1 and L_2, then:
  - $(e_1.e_2)$ is an RE denoting $L = L_1L_2$,

- $(e_1 + e_2)$ is an RE denoting $L = L_1 \cup L_2$,
- (e_1^+) is an RE denoting $L = L_1^+$,
- (e_1^*) is an RE denoting $L = L_1^*$.
- There in no other way how an RE can be defined.

An example of the RE can be $(a.b^*)^+ + c$. For example contains, for example, these strings: $a, abbb, abab, c$.

2.3 Finite Automata (FAs)

We use FAs as another way to represent regular languages. An FA is a 5-tuple $M = (Q, \Sigma, R, s, F)$ where:

- Q is a finite set of states,
- Σ is an input alphabet,
- R is a finite set of rules of the form: $pa \to q$, where $p, q \in Q$, $a \in \Sigma$,
- $s \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.
- We can replace the set of rules R by a transition function $\delta : Q \times \Sigma \to 2^Q$.

The below notions are used in following text.

A state is non-terminating if there is no possibility to accept anything starting from it using any sequence of rules $\in R$, and a state is called inaccessible if we cannot get from s into it using any sequence of rules $\in R$.

The language of an FA is defined as $L(M) = \{w \in \Sigma^* \mid \exists q_f \in F : s \xrightarrow{w} q_f\}$, w are strings accepted by M. The forward state language is defined as $L(M, q) = \{w \in \Sigma^* \mid \exists q_f \in F : q \xrightarrow{w} q_f\}$ where $q \in Q$. The backward state language is $\overleftarrow{L}(M, q) = \{w \in \Sigma^* \mid s \xrightarrow{w} q\}$ where $q \in Q$. Two states $p, q \in Q$ are indistinguishable if they have the same forward state languages.

2.3.1 Types of FAs

We call an FA ϵ-free if the following holds: $\forall(pa \to q) \in R$: $p, q \in Q$, $a \in \Sigma$. A deterministic FA (DFA) is then an ϵ-free FA where $\forall q \in Q$, $\forall a \in \Sigma$ there is at most one transition. A DFA is complete if the following holds: $\forall q \in Q, \forall a \in \Sigma$ exists at least one rule $pa \to q \in R$, $q \in Q$ We can get a complete DFA from a DFA by adding one non-terminating state p_{trap} with a „self loop on the whole Σ“ ($\forall a \in \Sigma$: $p_{trap}a \to p_{trap}$). If a complete DFA has no inaccessible state and has at most one non-terminating state, it is called a well-specified FA (WSFA). If a WSFA does not contain any indistinguishable states, we call it a minimal FA. A non-deterministic FA (NFA) is an FA which is not a DFA.

An example of a DFA is shown in Figure 2.3a, and an example of an NFA is shown in Figure 2.1.

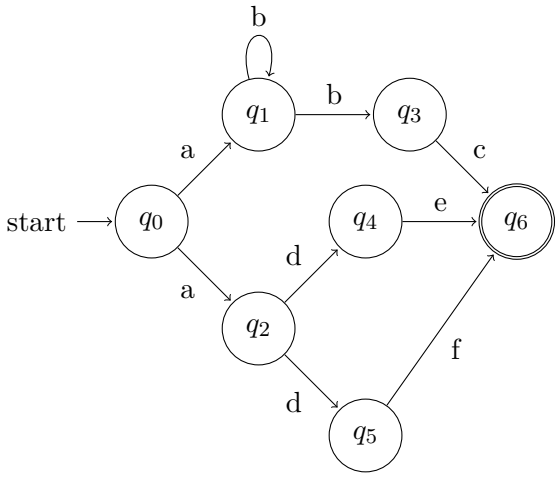


Figure 2.1: An example of an NFA accepting the language given by the RE: $a.((b^+.c) + (d.(e + f)))$

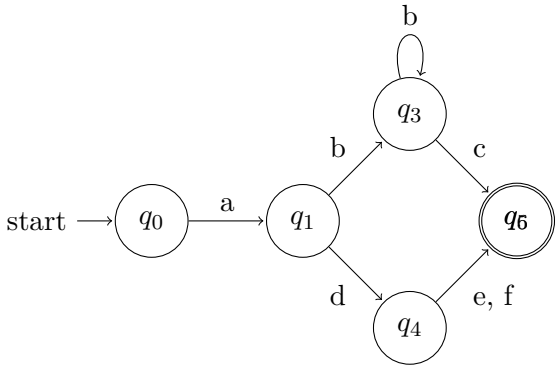


Figure 2.2: An example of a DFA accepting the language of same RE as the NFA in Figure 2.1

2.3.2 Effective Boolean Algebras

This definition of an effective Boolean algebra (EBA) is taken from the article [10].

An effective Boolean algebra $\mathbb{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a *universe* of underlying domain elements. Ψ is a set of unary *predicates* closed under the Boolean connectives $\vee, \wedge : \Psi \times \Psi \to \Psi$ and $\neg : \Psi \to \Psi$; and $\bot, \top \in \Psi$ are the *false* and *true* predicates. Values of the algebra are sets of domain elements, and the *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ satisfies that $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\boldsymbol{Sat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$, and we say that φ is *satisfiable*. We require that $\boldsymbol{Sat}$ as well as $\vee$, $\wedge$, and $\neg$ are *computable* as a part of the definition of an effective Boolean algebra. We write $x \models \varphi$ for $x \in [\![\varphi]\!]$ and we use $\mathbb{A}$ as a subscript of a component when it is not clear from the context, e.g., $[\![\_]\!]_{\mathbb{A}} : \Psi_{\mathbb{A}} \to 2^{\mathfrak{D}_{\mathbb{A}}}$.

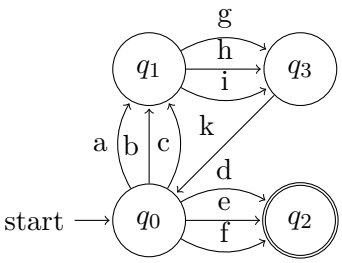
2.3.3 Symbolic Automata (SAs)

One of the drawbacks of FAs is that they do not scale well for large alphabets. Symbolic automata allow us to replace individual characters with sets of characters so we are able to group a large number of transitions into one. These transitions have to have the same target and source state.

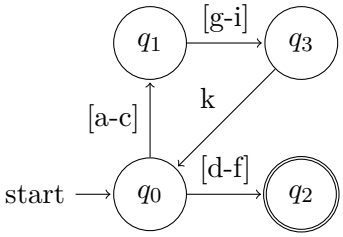
Effective Boolean algebras provide a basis for defining symbolic automata.

An SA is a 5-tuple $M = (Q, A, \delta, s, F)$ where:

- Q is a finite set of states,
- A is an EBA such that:
  - $DomainElements(A) = \Sigma_A$ is an input symbol alphabet,
  - $Predicate(A) = \Psi_A$ is a guard alphabet,
  - $[[...]]_A = \Psi_A \to 2^{\Sigma_A}$,
- δ is a function $\delta : Q \times \Psi_\epsilon \to 2^Q$,
- $s \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.



(a) An example of a DFA accepting the language of the RE:
$([a-c].[g-i].k)^*.[d-f]$



(b) An example of an SA accepting the language of same RE as the DFA in Figure 2.3a

Figure 2.3: An SA example

2.4 Finite State Transducers (FTs)

An FT describes a binary relation between strings of two languages. It is a form of automaton with one input and one output tape.

Let $w \in \Sigma^*$ be a string then the FT T can translate this string into other strings. This translation is the relation which T represents.

An FT is a 6-tuple $T = (Q, \Sigma, \Omega, \delta, s, F)$ where:

- Q is a finite set of states,
- Σ is an input alphabet,

- Ω is an output alphabet,
- δ is a transition relation such that $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times Q$,
- $s \in Q$ is an initial state,
- $F \subseteq Q$ is a set of final states.

An ϵ-move is a transition in δ which does not read nor write a symbol.

An example of an FT is shown in Figure 2.4

A deterministic transducer (DFT) is an ϵ-free FT where $\forall q \in Q$, $\forall a \in \Sigma$, $\forall b \in \Omega \cup \{\epsilon\}$ there is at most one transition. A non-deterministic FT (NFT) is an FT which is not a DFT.

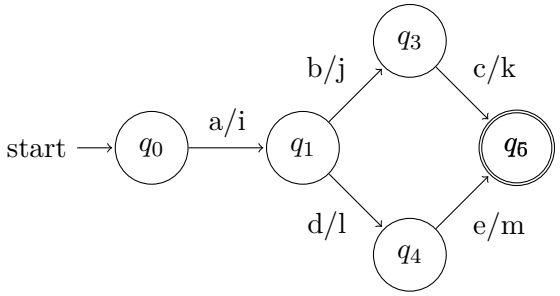


Figure 2.4: A DFT translating the language of the RE $a.((b.c) + (d.e))$ into the language of the RE $i.((j.k) + (l.m))$

2.5 Automata Reverse Concatenation

We are using an automata reverse concatenation in the ASMA tool for the backward run during program analysis. Firstly we are creating a concatenation of the prefix and suffix automata, then we come back with this concatenation (target automaton), which can be modified. The purpose of this automata reverse concatenation operation is to find two automata representing our new prefix and suffix for continuing in the program analysis.

An automata concatenation is an operation that, for given FAs A_1, A_2, produces an FA A such that $L(A) = L(A_1).L(A_2)$. Note that the operation is defined as non-deterministic since we do not require it to return any particular A. We just require the property that $L(A) = L(A_1).L(A_2)$ holds.

An automata reverse concatenation is the reverse operation of automata concatenation. This operation has tree inputs: a prefix automaton (P), a suffix automaton (S), and a target automaton (T) and produces two automata, namely, a reverted prefix (RP) and a reverted suffix (RS). RP and RS are the smallest sets such that, the following holds:

1. Both the reverted prefix and reverted suffix must be subsets of the original prefix and suffix.
2. Some prefixes and suffixes can be missing, but anytime some prefix p and suffix s are such that, their concatenation is in the target, the prefix p and suffix s must be present.

1. $L(RP) \subseteq L(P) \wedge L(RS) \subseteq L(S)$
2. $\forall p \in L(P), \forall s \in L(S) : p.s \in L(T) \Rightarrow p \in L(RP) \wedge s \in L(RS)$

From this definition, we can create conditions for result automata correctness testing. Lets modify second part:

$$\neg\neg(\forall p \in L(P), \forall s \in L(S) : \neg(p.s \in L(T)) \vee (p \in L(RP) \wedge s \in L(RS)))$$

$$\neg(\exists p \in L(P), \exists s \in L(S) : p.s \in L(T) \wedge (p \notin L(RP) \vee s \notin L(RS)))$$

$$\neg(\exists p \in L(P), \exists s \in L(S) : p.s \in L(T) \wedge p \notin L(RP)) \\ \wedge\neg(\exists p \in L(P), \exists s \in L(S) : p.s \in L(T) \wedge s \notin L(RS))$$

$$\neg(\exists p \in (L(P) \setminus L(RP)), \exists s \in L(S) : p.s \in L(T)) \\ \wedge\neg(\exists p \in L(P), \exists s \in (L(S) \setminus L(RS)) : p.s \in L(T))$$

$$\forall p \in (L(P) \setminus L(RP)), \forall s \in L(S) : p.s \notin L(T) \\ \wedge\forall p \in L(P), \forall s \in (L(S) \setminus L(RS)) : p.s \notin L(T)$$

$$(L(P) \setminus L(RP)).L(S) \cap L(T) = \oslash \wedge L(P).(L(S) \setminus L(RS)) \cap L(T) = \oslash$$

So result automata are not correct if one of following conditions holds:

1. The reverted prefix is not subset of the prefix.
2. The reverted suffix is not subset of the suffix.
3. The intersection of the target and the concatenation of the prefix without reverted prefix with the suffix is not empty.
4. The intersection of the target and the concatenation of the prefix with the suffix without reverted suffix is not empty.

1. $\neg L(RP) \subseteq L(P)$
2. $\neg L(RS) \subseteq L(S)$
3. $(L(P) \setminus L(RP)).L(S) \cap L(T) \neq \oslash$
4. $L(P).(L(S) \setminus L(RS)) \cap L(T) \neq \oslash$

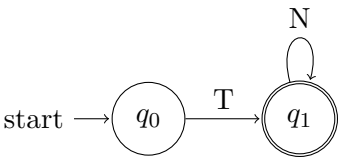
2.6 (Abstract) Regular Model Checking

Model checking (MC) is a technique for verifying whether a finite-state model of a system meets a given specification. The problem is that these days we need to deal with infinity which can be caused, e.g. by some abstract data types or by recursive procedures. One way how to verify infinite-state systems is to use Regular model checking (RMC). Regular model checking is a successful symbolic verification method using FAs to deal with infinity.

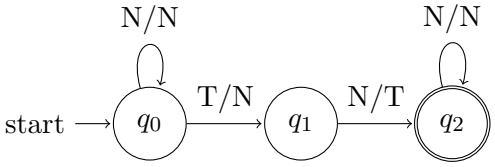
One type of RMC is Abstract regular model checking (ARMC) which can analyze more systems than regular model checking because it uses abstraction for acceleration which reduces the problem of an explosion that can occur while analyzing infinite-state systems. We also use abstraction to ensure termination of reachability computation. For abstraction, there are various techniques. ASMA tool uses so-called predicate abstraction, which is based

on collapsing states. These techniques are described in more detail in the thesis written by Michal Kotoun [7].

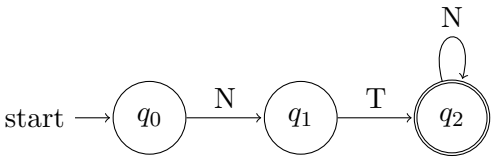
To check the system, we need the set of initial states, something to represent transitions such as transducer or specializes transitions implemented as automata operations, and something representing all bad states. Let's have a simple token passing protocol. This protocol has the alphabet: $\Sigma = \{N, T\}$, the set of initial states (in RE): $T.N^*$, bad states (in RE): $((T + N)^*.T.N^*.T.(T + N)^*) + N^*$. This RE says that we can not have system without any token, and that we can not have more than one token in the system regardless of their position. And we also need the transducer representing token shifting described in Figure 2.5b. Now we can check this protocol by applying the transducer to its set of initial states, abstracting it, and then checking the intersection with bad states. In the next step, we apply the transducer to the new set of states and repeat until the next set of abstracted states is not different from the previous and has no intersection with bad states. In our example we get Figure 2.5c after the first iteration each next iteration is equal to Figure 2.5d.



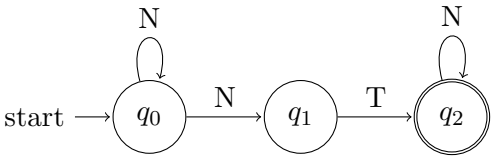
(a) An initialization state of a token-passing protocol



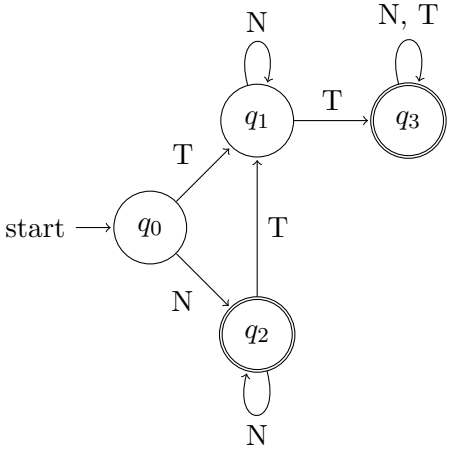
(b) A DFT of a token-passing protocol



(c) The DFT from Figure 2.5b applied on initialization state from Figure 2.5a



(d) The DFT from Figure 2.5b applied on initialization state from Figure 2.5a more times and minimized



(e) Bad states of a token-passing protocol

Figure 2.5: An example of a token-passing protocol inspired by article [4]

Chapter 3

ASMA and Its Analysis

In this chapter, we will cover the basics of the ASMA tool and its usage. This chapter is based on [7].

3.1 ASMA

The ASMA tool is software written in the C# programming language. It implements analysis of programs with strings based on ARMC. ASMA stands for Automata for String Manipulation analysis. This tool was implemented by Michal Kotoun on top of the Automata library [1] written by Margus Veanes for Microsoft.

The source code of the ASMA project is written in C#8 and is compatible with .NET Standard 2.0, .NET 4.7.2, and .NET Core 3.0 or higher. Repositories containing this ASMA project[1] as well as a modified fork of the Automata library[2] are on the private VeriFIT Github repository.

There are three main sub-projects of the ASMA tool: AsmaLib, AsmaSymExec, and AsmaCLI.

AsmaLib extends the functionality of the Automata library with general-use functionality for working with symbolic finite automata and restricted symbolic finite transducers in (A)RMC. The AsmaLib project allows ASMA to have implementations of algorithms for ARMC-based program analysis, e.g. abstraction. More details can be found in the thesis written by Michal Kotoun [7].

In the ASMA symbolic execution project (AsmaSymExe) there are sets of operations for forward and backward runs of program analysis that allow us to analyze programs manipulating with strings using Counter-example guided abstraction refinement which is well described in the thesis [7]. AsmaSymExec also contains a parser for the intermediate code of programs and their representation.

The AsmaCLI project implements the command-line interface (CLI) of the ASMA tool. It contains many commands, e.g., for constructing symbolic finite automata or loading them from a file, constructing an intersection of two SFAs, printing them, or running the whole analysis. In the ASMA CLI, there is also the possibility to run the command „interactive“

[1] Viz https://github.com/VeriFIT/ASMA
[2] Viz https://github.com/VeriFIT/Automata

switching AsmaCLI into the interactive mode which means that the user can type multiple ASMA commands without the need to run the CLI for typing each new command. This mode has the possibility of turning on stepping for the analysis of programs. There is a manual listing of AsmaCLI statements in Appendix A.

3.2 Analysis of ASMA

One of our goals was to identify weaknesses of the ASMA tool through experiments as well as studying its code. We should have focused on the way how various special operations on symbolic automata are implemented and on the used automata reduction methods.

3.2.1 Analysis of the Source Code

At first glance, there were some shortcomings in the AsmaCLI. We wanted to see automata after each step of the analysis and after each command in an interactive mode which meant typing another command to save these automata into files and then open these files in the Visual Studio Code[3] or in the Visual Studio[4]. There was no switch to turn on automatic saving or previewing of automata in the AsmaCLI.

We were also missing some option to work with non-deterministic automata because the automata reduction was always done by determinization and minimization.

At this point, we saw some opportunities to improve the ASMA tool by adding a parameter to the analysis that would allow one to save and preview automata during analysis and by adding further reduction algorithms that would allow the ASMA tool to work with non-deterministic automata. This also mean giving an option to users to choose between those reduction algorithms directly from the AsmaCLI.

3.2.2 Experiments and Profiling

After the analysis of the ASMA source code, we moved into playing with the tool and trying to understand its possibilities. The first thing we did was simply run some analyses on examples mentioned in the master thesis written by Michal Kotoun. The following list of program examples (translated intermediate code from php by Michal Kotoun for comparing the ASMA tool to the Stranger tool[5]) was found and became our testing data set for future bench-marking: `stranger-01-sanit`, `stranger-11-vuln`, `stranger-12-sanit`, `stranger-21-vuln`, `stranger-22-sanit`, `stranger-31-vuln`, `stranger-32-vuln`, `stranger-33-vuln`, `stranger-34-sanit`, `stranger-35-vABS`, `stranger-35-vuln`, `stranger--36-vuln`, `stranger-41-vuln`, `stranger-42-sanit`.

We were able to successfully run an analysis of each of these programs so we started with profiling. The main problems were the reverse concatenation opetarion (described in Section 2.5) and automata reductions which took a significant amount of time and computer resources. An important part of the profiling output is in Table 5.4, where almost 80 % of all resources are used by the reverse concatenation operation.

[3] Viz https://code.visualstudio.com
[4] Viz https://visualstudio.microsoft.com
[5] Viz https://vlab.cs.ucsb.edu/stranger/

All analyses of examples and experiments were run with the same conditions on a PC with Windows 10, AMD Ryzen 5 3600, 48 GB RAM, GTX 1080, and an SSD disk.

After detecting the main bottlenecks, we tried to find their cause. The cause of the main bottleneck was the ASMA's original reverse concatenation operation which was implemented by Michal Kotoun to replace transitions with epsilons instead of creating new epsilon-free automata. (This original implementation is described in Subsection 4.2.1.) Moreover, the automata reduction was fixed to automata determinization and minimization. ASMA was also calling these reductions on not optimal places and had some not optimal heuristics slowing this tool for these reductions.

ASMA tool allows one to save automata in any part or analysis into a file with format .dot or .dgml. These data can be graphically represented in both Visual Studio (.dgml format) or Visual Studio Code with the extension Graphviz Interactive Preview18[6] (.dot format). This extension can do a quick re-rendering of automata stored in a .dot file in addition to Visual Studio. Because of this Visual Studio Code was chosen for live visualization of automata created during ASMA tool analysis.

Using this tool we were able to visualize and confirm that reverse concatenation operation is only replacing transitions with epsilons. Figure 3.1 is a screenshot of a visualization of a simple reverse concatenation operation in Visual Studio Code.

[6] Viz https://github.com/tintinweb/vscode-interactive-graphviz

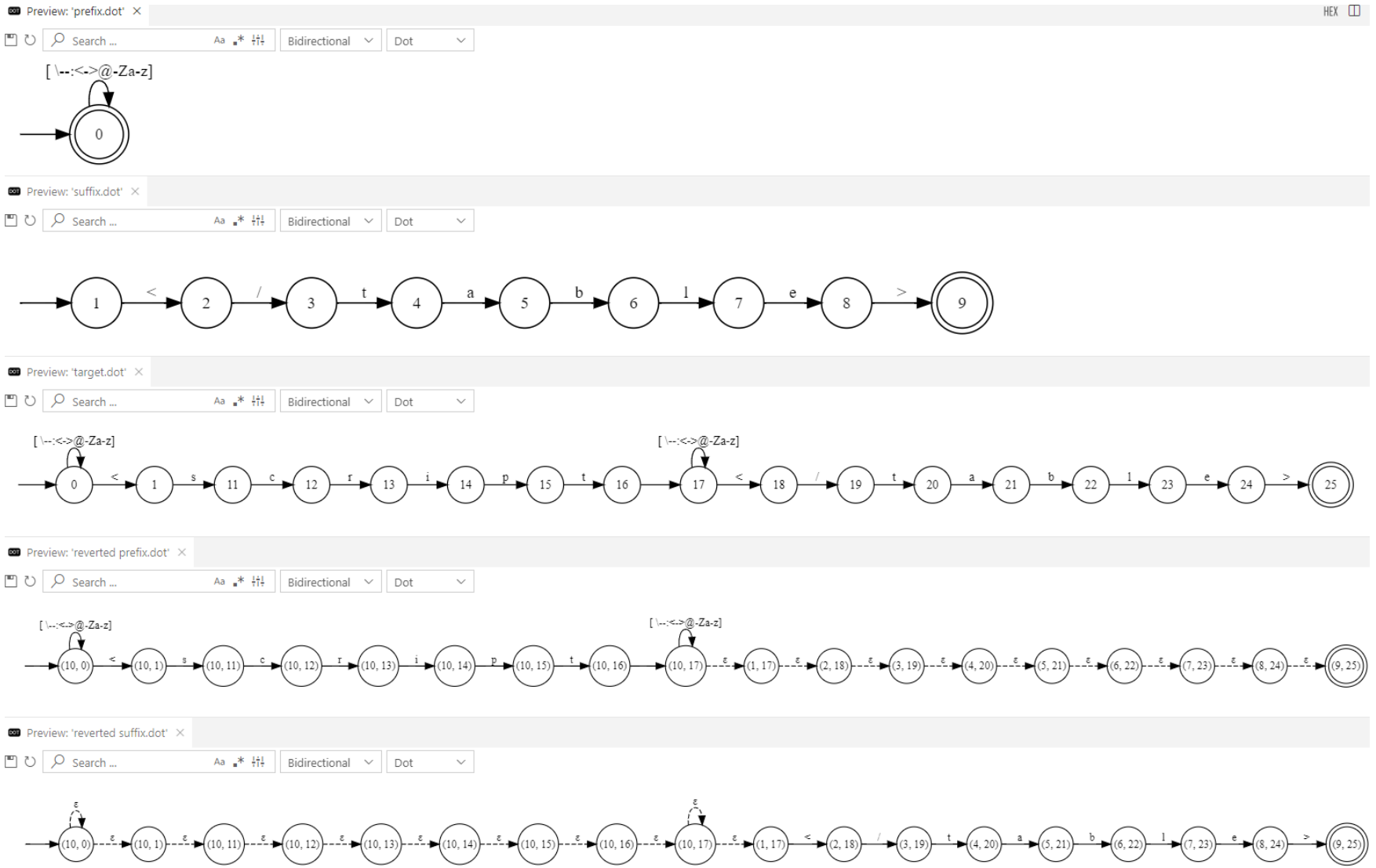


Figure 3.1: An example of automata after the original reverse concatenation operation. In the screenshot there are shown five automata: prefix, suffix, target, reverted prefix and reverted suffix after the first occurred computation of the reverse concatenation operation in an analysis of `stranger-12-sanit` program. We can notice how the original reverse concatenation operation implementation creates ϵ-transitions on the created reverted automata.

Chapter 4

Proposed Improvements

In this chapter, we will move from the analysis of the ASMA tool to all proposed improvements and describe them.

4.1 Reduction Algorithms

First of all, we wanted to add more types of reduction algorithms to allow one to run analysis with non-deterministic automata. We could do both: implement new algorithms or search for some existing ones. Fortunately, we managed to find an existing library with implementations of simulation reductions developed by Juraj Síč for his bachelor thesis [9]. This SymbolicSimulation library is available at a bitbucket repository[1].

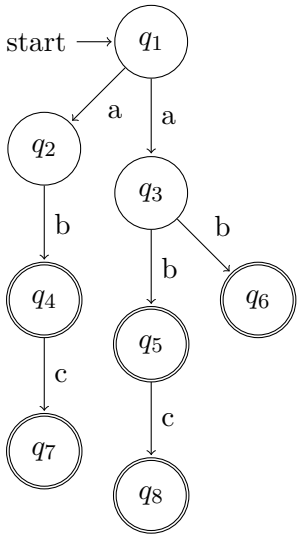
Our first improvement was thus an integration of this library into the ASMA tool. The benefits consist in a possibility for user to freely choose which reduction algorithm should be used both during an ARMC run (for all counted automata) as well as on a given automaton in the AsmaCLI. Each reduction such as bisimulation and simulation are computed in a different way and can produce different results. There is an example of these two reductions in Figure 4.1. The figure illustrates that each type of reduction can indeed reduce automata differently.

Due to our work, ASMA now offers not only the original reductions (`Determinization + Minimization, Bisimulation`) but also (`Local simulation, Local simulation optimized, Simulation no count no opt, Simulation no count, Global simulation`) that we added and that are defined in the thesis written by Juraj Síč [9].

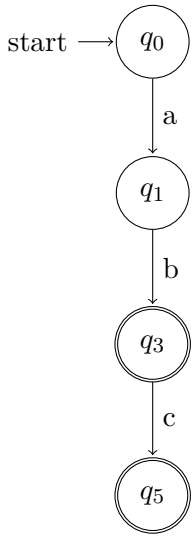
So there are in the ASMA tool three new algorithms for computing simulation preorder on SFAs. First one, algorithm `Global simulation`, which is based on global mintermization. Second one, algorithm `Simulation no count` which do not use mintermization but only the capabilities of symbolic automata and its optimized version. The last one, algorithm `Local simulation` which is a modification of algorithm `Simulation no count` that uses local mintermization and its optimized version. This description was taken from the thesis written by Juraj Síč [9].

There is also a possibility to improve reductions even more in this tool. The next future work on ASMA can be an implementation of combined reduction algorithms which are

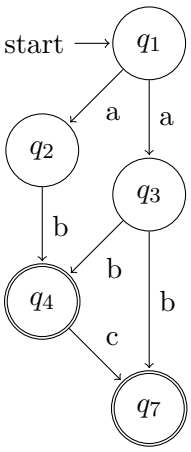
[1] Viz https://bitbucket.org/jsic/symbolicsimulation/src/master/



(a) An automaton for testing reduction algorithms



(b) The automaton from Figure 4.1a reduced by simulation



(c) The automaton from Figure 4.1a reduced by bisimulation

Figure 4.1: An example of automata after bisimulation and simulation reduction of automaton from Figure 4.1a.

described in [2] and [6]. The next huge potential lies in antichain based inclusion checking. There is more info about them in [3] and [6].

4.2 Reverse Concatenation Operation

After new algorithms for automata reduction were added, tested, and evaluated, we wanted to create a new implementation of the reverse concatenation operation which would not create epsilon transitions. The original version is described in Subsection 4.2.1. Because we have not found any implementations of this operation that we could use for our inspiration, we proposed several gradually improving versions ourselves. They are more optimized for ASMA and are described below.

To be sure that the new approaches are working properly, we proposed self-tests for their testing and comparing the results with the original implementation.

This operation as well as self-tests are described in Subsection 2.5.

4.2.1 The Original Implementation

The original implementation written by Michal Kotoun for his thesis [7] is much simpler than our approaches. Algorithm 1 describes this approach. On the first line, there is a creation of a parser for the reverted prefix which is a concatenation of a prefix identity transducer and a suffix epsilon transducer. The second line is the same for the reverted suffix but it is a concatenation of a prefix epsilon transducer and a suffix identity transducer.

Algorithm 1: An algorithm of the reverse concatenation operation original implementation

Input: Prefix NFA (P), Suffix NFA (S). Target NFA (T)
Output: Reverted Prefix NFA (RP), Reverted Suffix NFA (RS)
1 Pparser := P.MakeIdentityTransducer().Concat(S.MakeEpsilonTransducer())
2 Sparser := P.MakeEpsilonTransducer().Concat(S.MakeIdentityTransducer())
3 RP := Pparser.ApplyOn(T)
4 RS := Sparser.ApplyOn(T)
5 **return** *(RP.RemoveEpsilons(), RS.RemoveEpsilons())*

These two parsers are then applied on the target for reverted automata creation (lines 3, 4). On the last line are the reverted automata stripped of epsilon transitions and returned.

An example how the original version of the reverse concatenation operation creates ϵ-transitions can be seen in Figure 3.1

4.2.2 The First New Approach

Our first approach is creating reverted automata without ϵ-transitions and dead states. This approach performs only one simultaneous forward run in the prefix NFA concatenated with suffix NFA and the target NFA which is similar to a depth-first algorithm. This one forward run is shown in Figure 4.2. Each visited state is extended with a structure for storing all data we need to create reverted automata.

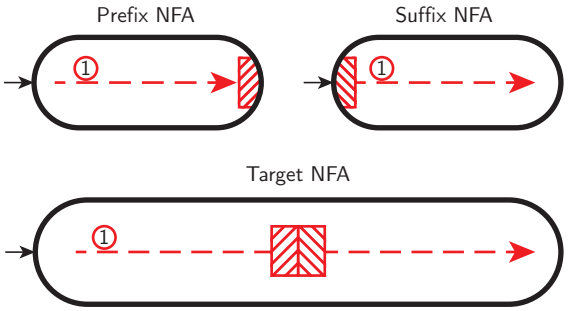


Figure 4.2: The illustration of all runs in automata by our first approach. The symbol 1 in the red circle illustrates the single simultaneous forward run through the automata. In later figures, more lines of this kind will appear.

In this first approach, we firstly define many auxiliary variables, structures, arrays, and dictionaries for storing data about paths in automata, moves, and states such as state id, information if a state or some path leads to the end or if the state is final and data representing paths of loops. To explain what a loop stands for, we present Figure 4.3 with some loops. Each move has a source state, a target state, and label. The source state and the target state are synchronized states in the prefix or suffix and target automata.

We present our top-level pseudo-code of Algorithm 2 describing our approach of the reverse concatenation operation.

In Algorithm 2, we initiate a forward run with the initial state in the prefix and the target NFAs on the first line. On the second line, there is a check if we were here before. If no

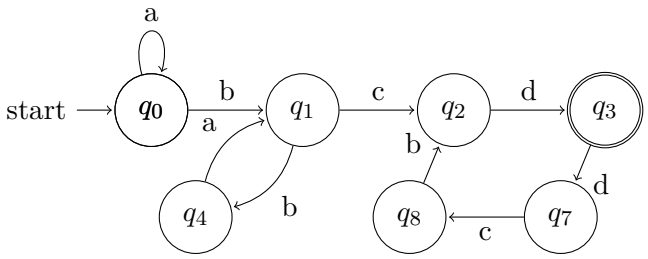


Figure 4.3: An example of loops in an automaton. There are loops outgoing from states q0 (q0 - q0), q1 (q1 - q4 - q1), and q3 (q3 - q7 - q8 - q2 - q3).

Algorithm 2: The first proposed version of the reverse concatenation operation.

Input: Prefix NFA (P), Suffix NFA (S). Target NFA (T)
Output: Reverted Prefix NFA (RP), Reverted Suffix NFA (RS)
`// Perform a depth-first simultaneous forward run through P implicitly concatenated with S and T.`

```
1  foreach state in run do
2      if state is not noted then
3          Store info about sate and move
4      else if loop detected then
5          Store path of the loop
6      if state is final or leading to the end then
7          Save info about current path
8  foreach loop in detected loops do
9      if init. state of loop will be in RP or RS then
10         Save info about the loop path
11 Create initial state for RS
12 Construct RP from stored data
13 Construct RS from stored data
14 return (RP, RS)
```

(line 3), we store info about this state (noting state means storing an info about it into auxiliary variables), the move into this state and we also check if this state is final (final in the suffix and the target.) If yes or if is leading to the end, we copy the missing part of the current path into auxiliary variables for the future reverted automata generation and store info that each state in the current path leads to the end. Line 5 says that, if we were in this state before, we are storing all info about this new loop, which is all states and moves in the whole path of the loop.

On lines 6 and 7, there is a check if the state in the forward run is final. If yes, we store info about it into auxiliary variables (that it is a final state), and once again, we copy the missing part of the current path into auxiliary variables for the future reverted automata generation which is on lines 12, 13 and store info that each state in the current path leads to the end.

After the forward run, we need to process detected loops. So on lines 8, 9, and 10, we check if any state of the loop will be generate in the reverted automata generation, for each loop. If yes, we copy the loop path into auxiliary variables for the future reverted automata generation.

On line 11, there is a creation of the reverted suffix initial state. This implementation of the reverse concatenation operation can create many potentially initial states for the reverted suffix. We have to have only one, so we copy all moves from each potentially initial suffix state into one new state. This new state will be the initial one. We also remove all potentially initial states which are not the target of any transition for any state in the reverted suffix. This process is described in more details in Subsection 11 but without removing dead states.

Lines 11 and 12 generate the reverted prefix and reverted suffix. We have to provide from the auxiliary variables an initial state, array of final states, and moves in the automaton to generate each reverted automaton. This automaton generation is done by the Automata library. These generated automatons are returned on line 14.

We reached quite some speedup after implementing the first approach as shown in Table 4.1 see in particular columns: *R.C. Old* and *R.C. 1*, but the profiler showed us that this operation is still really expensive in the context of our benchmarks. The main cause is storing all data about each state and move which is less effective than making multiple runs through the automata.

4.2.3 The Second Approach

So we decided to continue improving this implementation and the result was a new proposed approach described in Algorithm 3 which is based on multiple forward and backward runs in automata. These forward and backward runs are shown in Figure 4.4. As in the first approach, we firstly define auxiliary variables, arrays, and hash-sets. The hash-set is something like an array in C# language but it is much more efficient.

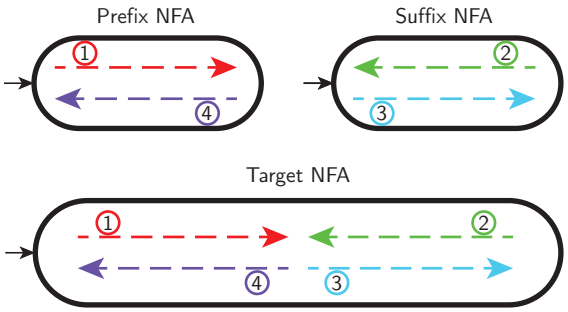


Figure 4.4: An illustration of the second proposed approach to reverse concatenation. The numbers in circles represent the different phases of the approach. Number 1 represents a simultaneous forward run in the prefix and the target NFAs, 2 represents a simultaneous backward run in the suffix and the target NFAs, 3 represents a simultaneous forward run in the suffix and the target NFAs, and 4 represents a simultaneous backward run in the prefix and the target NFAs.

In Algorithm 3, we firstly perform a simultaneous forward run in the prefix and the target NFAs on line 1 and a simultaneous backward run in the suffix and the target NFAs on line 2. In the first run, there is only one initial state, but in the second run, there can be plenty of final states in both automata: in the suffix and the target. So we initiate a backward run from combinations of all final states in the suffix and the target. In those two runs, we

Algorithm 3: The second proposed version of the reverse concatenation operation.

Input: Prefix NFA (P), Suffix NFA (S). Target NFA (T)
Output: Reverted Prefix NFA (RP), Reverted Suffix NFA (RS)
1 Perform simultaneous forward run in P and T
2 Perform simultaneous backward run in S and T
3 Get connection states between P and S
4 Perform simultaneous forward run in S and T
5 Perform simultaneous backward run in P and T
6 Create initial state for RS
7 Construct RP from stored data
8 Construct RS from stored data
9 **return** *(RP, RS)*

only store states we went through, all final states in the first run and all initial states in the second run. On line 3, we create transition states which are intersection of those initial and final states. A state is a transition state if the target state is the same in the suffix initial state as in the prefix final state.

From these transition states, we perform a simultaneous forward run in the suffix and the target on line 4 and a simultaneous backward run in the prefix and the target on line 5. These runs are limited only to go through states we already went through in previous runs. This will eliminate all dead states and ensure that all states we are going through in these two runs will be in reverted automata. So we are saving every move and state during these two runs for the reverted automata generation. The rest is the same as in Algorithm 2.

Unfortunately, the second approach was more than twice slower as the original one according to Table 4.1. The primary reason for this is the second run, performing the backward run from the combinations of target final and suffix final states. If we have to initiate that run from each such a state, it can be really resource-consuming.

4.2.4 The Third Approach

After learning from the mistake in the previous approach, we created another proposed version of the approach to handle the reverse concatenation operation. This version is shown in Algorithm 4 and it is almost the same as the previous Algorithm 3. The main difference is the order of runs. These runs are shown in Figure 4.5.

Algorithm 4: The third proposed version of the reverse concatenation operation.

Input: Prefix NFA (P), Suffix NFA (S). Target NFA (T)
Output: Reverted Prefix NFA (RP), Reverted Suffix NFA (RS)
1 Perform simultaneous forward run in P and T
2 Perform simultaneous forward run in S and T
3 Perform simultaneous backward run in S and T
4 Perform simultaneous backward run in P and T
5 Create initial state for RS
6 Construct RP from stored data
7 Construct RS from stored data
8 **return** *(RP, RS)*

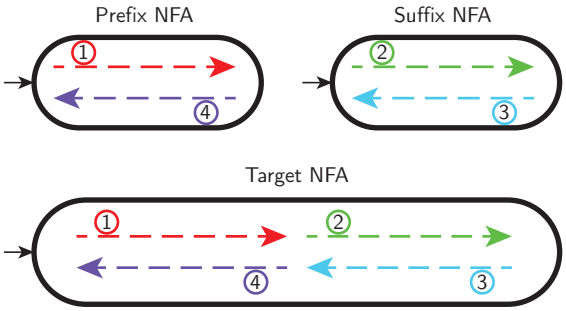


Figure 4.5: An illustration of the third proposed approach to reverse concatenation. The numbers in circles represent the different phases of the approach. Number 1 represents a simultaneous forward run in the prefix and the target NFAs, 2 represents a simultaneous forward run in the suffix and the target NFAs, 3 represents a simultaneous backward run in the suffix and the target NFAs, and 4 represents a simultaneous backward run in the prefix and the target NFAs.

In Algorithm 4, we firstly perform a simultaneous forward run in the prefix and the target on the first line and a simultaneous run in the suffix and the target on the second line. We need to remember only all states we went through as in Algorithm 3. The major change is that we know now which final states in the suffix and the target will be in the reverted suffix automaton. The next step is to perform further two runs as in Algorithm 3 with the same purpose. These runs are a simultaneous backward run in the suffix and the target on line 3 and a simultaneous backward run in the prefix and the target on line 4. The rest is the same as in Algorithm 2 and in Algorithm 3.

The third version is more than twice faster than the original one and much faster than the first approach.

4.2.5 The Fourth Approach

We, however, attempted to improve the last approach even more. Our last version of the reverse concatenation operation is faster than all previously mentioned approaches and about 75 % faster than the original one on some examples. The comparison is in Table 4.1.

The main difference in this new version compared to the others is that we do not aim at creating the reverted automata without dead states as can be seen in Algorithm 5 described in following paragraphs. Dead states are removed later by Automata library during the reverted automata generation. This leads to simple runs in automata shown in Figure 4.6 not showing the effect of the later presented heuristics. The third version was creating something we called „forks“. There is an example of one fork in the Figure 4.7. It is simply an automaton with multiple identical paths creating bigger automaton (in our data sets, created automata are three times or six times bigger.) The cause of this problem are multiple similar independent paths in the target. This problem was solved in our fourth approach. The difference can be seen in Table 4.2 in reverted suffix columns for the third and fourth versions of our approaches.

In Algorithm 5, we first apply some heuristics which can find both reverted automata without the need of performing any simultaneous run in them. (Find a reverted automaton

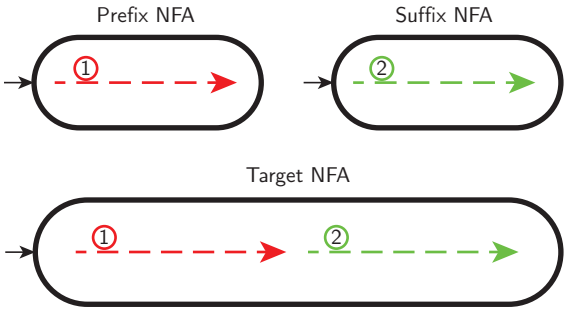


Figure 4.6: An illustration of the fourth proposed approach to reverse concatenation. The numbers in circles represent the different phases of the approach. Number 1 represents a simultaneous forward run in the prefix and the target NFAs, and 2 represents a simultaneous forward run in the suffix and the target NFAs.

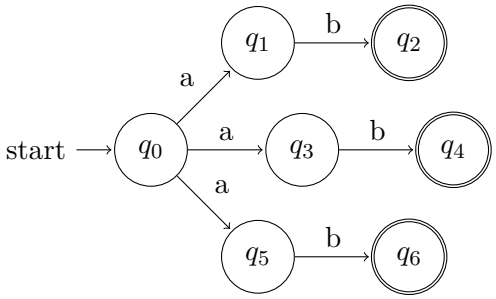


Figure 4.7: An example a fork automaton.

means to construct it.) On the first and second line, we are trying to find reverted automata. If we find both of them, we are returning them on line 4. Both heuristics are focused on automata with only one path in them.

If we did not find the reverted prefix, we have to perform one simultaneous forward run in the prefix and the target on line 6. In this run, we are storing info about each state and move for the reverted prefix generation. If we did not find the suffix, we have to perform one simultaneous forward run in the suffix and the target on line 8 and we are again storing all that data.

At this moment we have all data we need to create the initial state for the reverted suffix on line 9 and construct reverted automata on lines 10 and 12 if needed. During this construction, we simply ask the Automata library to remove dead states. The dead states removal using this library is much more effective than we achieved in our previous approaches.

We will now provide a detailed description of our most complex version of the reverse concatenation operation. The next subsections have similar names as lines in our top-level Algorithm 5 for easier orientation.

Try to Find Reverted Prefix

First of all, we are trying to find both reverted automata without the need of performing any simultaneous run in them. To do that in Algorithm 6, we are first checking if the prefix automaton has only one state which is also the final state, and no moves (as in Figure 4.9a)

Algorithm 5: The fourth proposed version of the reverse concatenation operation.

Input: Prefix NFA (P), Suffix NFA (S). Target NFA (T)
Output: Reverted Prefix NFA (RP), Reverted Suffix NFA (RS)

```
1  Try to solve RP
2  Try to solve RS
3  if prefixFound ∧ suffixFound then
4      return (RP, RS)
5  if !prefixFound then
6      Perform simultaneous forward run in P and T
7  if !suffixFound then
8      Perform simultaneous forward run in S and T
9      Create initial state for RS
10     Construct RS from stored data and remove dead states
11 if !prefixFound then
12     Construct RP from stored data and remove dead states
13 return (RP, RS)
```

Table 4.1: A comparison of the original version and all newly proposed optimisations of the reverse concatenation operation implementation on all stranger examples. Values are the average time of 10 runs of the whole analysis in [ms]. Algorithms: the original version (R.C. Old), the first prototype (R.C. 1), the second prototype (R.C. 2), the third prototype (R.C. 3), the fourth prototype (R.C. 4).

stranger	**R.C. Old**	**R.C. 1**	**R.C. 2**	**R.C. 3**	**R.C. 4**
01-sanit	1	1	1	1	1
11-vuln	28	28	26	26	**18**
12-sanit	67	57	**53**	57	66
21-vuln	178	**77**	169	155	158
22-sanit	544	1,752	1,223	**336**	1,489
31-vuln	8,030	4,685	14,363	4,407	**2,773**
32-vuln	25,298	16,619	87,006	8,027	**7,724**
33-vuln	3,994	4,593	9,878	2,131	**1,687**
34-sanit	470	433	**310**	547	1,047
35-vABS	24,079	15,540	63,018	9,262	**6,477**
35-vuln	42,821	25,638	120,569	13,616	**8,654**
36-vuln	19,913	21,139	22,263	18,662	**18,466**
41-vuln	**37**	**37**	**37**	**37**	39
42-sanit	**28**	32	**28**	**28**	30

Table 4.2: Statistics of automata attributes during a run of ARMC on one selected example `stranger-35-vuln.asmasym` using the different proposed heuristics for the reverse concatenation operation. Each line represents the state of automata immediately after the reverse concatenation operation is completed. Each data cell consists of: | number of states / number of transitions / count of final states |

prefix	suffix	target	rev. prefix	RC3 rev. suffix	RC4 rev. suffix
381 / 870 / 1	18 / 17 / 1	3,603 / 15,031 / 6	8,670 / 30,946 / 6	103 / 102 / 6	**18 / 17 / 1**
208 / 452 / 1	174 / 418 / 1	3,501 / 14,608 / 6	3,943 / 14,562 / 24	4,533 / 15,354 / 6	4,533 / 15,354 / 6
208 / 452 / 1	1 / 0 / 1	341 / 1,005 / 3	483 / 1,334 / 3	1 / 0 / 1	1 / 0 / 1
47 / 46 / 1	162 / 406 / 1	341 / 1,005 / 3	47 / 46 / 1	437 / 1288 / 3	437 / 1,288 / 3
1 / 0 / 1	47 / 46 / 1	47 / 46 / 1	1 / 0 / 1	47 / 46 / 1	47 / 46 / 1
150 / 394 / 1	13 / 12 / 1	295 / 959 / 3	401 / 1,252 / 3	37 / 36 / 3	**13 / 12 / 1**
137 / 381 / 1	14 / 13 / 1	259 / 839 / 3	332 / 1,075 / 3	40 / 39 / 3	**14 / 13 / 1**
122 / 366 / 1	16 / 15 / 1	220 / 709 / 3	261 / 887 / 3	46 / 45 / 3	**16 / 15 / 1**
107 / 351 / 1	16 / 15 / 1	175 / 559 / 3	184 / 675 / 3	46 / 45 / 3	**16 / 15 / 1**
96 / 285 / 56	12 / 11 / 1	130 / 409 / 3	120 / 399 / 80	35 / 35 / 3	**12 / 11 / 1**
41 / 40 / 1	56 / 245 / 56	96 / 239 / 56	41 / 40 / 1	57 / 201 / 57	57 / 201 / 57
26 / 25 / 1	16 / 15 / 1	41 / 40 / 1	26 / 25 / 1	16 / 15 / 1	16 / 15 / 1
22 / 21 / 1	5 / 4 / 1	26 / 25 / 1	22 / 21 / 1	5 / 4 / 1	5 / 4 / 1
159 / 403 / 1	16 / 15 / 1	278 / 824 / 3	337 / 1,027 / 3	46 / 45 / 3	**16 / 15 / 1**
132 / 376 / 1	28 / 27 / 1	233 / 674 / 3	226 / 781 / 3	82 / 81 / 3	**28 / 27 / 1**
122 / 311 / 56	11 / 10 / 1	152 / 404 / 3	142 / 381 / 76	31 / 30 / 3	**11 / 10 / 1**
67 / 66 / 1	56 / 245 / 56	122 / 265 / 56	67 / 66 / 1	57 / 201 / 57	57 / 201 / 57
49 / 48 / 1	19 / 18 / 1	67 / 66 / 1	49 / 48 / 1	19 / 18 / 1	19 / 18 / 1
33 / 32 / 1	17 / 16 / 1	49 / 48 / 1	33 / 32 / 1	17 / 16 / 1	17 / 16 / 1
26 / 25 / 1	8 / 7 / 1	33 / 32 / 1	26 / 25 / 1	8 / 7 / 1	8 / 7 / 1
22 / 21 / 1	5 / 4 / 1	26 / 25 / 1	22 / 21 / 1	5 / 4 / 1	5 / 4 / 1
381 / 870 / 1	18 / 17 / 1	3,603 / 15,031 / 6	8,670 / 30,946 / 6	103 / 102 / 6	**18 / 17 / 1**
208 / 452 / 1	174 / 418 / 1	3,501 / 14,608 / 6	3,943 / 14,562 / 24	4,533 / 15,354 / 6	4,533 / 15,354 / 6
208 / 452 / 1	1 / 0 / 1	341 / 1,005 / 3	483 / 1,334 / 3	1 / 0 / 1	1 / 0 / 1
47 / 46 / 1	162 / 406 / 1	341 / 1,005 / 3	47 / 46 / 1	437 / 1,288 / 3	437 / 1,288 / 3
1 / 0 / 1	47 / 46 / 1	47 / 46 / 1	1 / 0 / 1	47 / 46 / 1	47 / 46 / 1
150 / 394 / 1	13 / 12 / 1	295 / 959 / 3	401 / 1,252 / 3	37 / 36 / 3	**13 / 12 / 1**
137 / 381 / 1	14 / 13 / 1	259 / 839 / 3	332 / 1,075 / 3	40 / 39 / 3	**14 / 13 / 1**
122 / 366 / 1	16 / 15 / 1	220 / 709 / 3	261 / 887 / 3	46 / 45 / 3	**16 / 15 / 1**
107 / 351 / 1	16 / 15 / 1	175 / 559 / 3	184 / 675 / 3	46 / 45 / 3	**16 / 15 / 1**
96 / 285 / 56	12 / 11 / 1	130 / 409 / 3	120 / 399 / 80	35 / 35 / 3	**12 / 11 / 1**
41 / 40 / 1	56 / 245 / 56	96 / 239 / 56	41 / 40 / 1	57 / 201 / 57	57 / 201 / 57
26 / 25 / 1	16 / 15 / 1	41 / 40 / 1	26 / 25 / 1	16 / 15 / 1	16 / 15 / 1
22 / 21 / 1	5 / 4 / 1	26 / 25 / 1	22 / 21 / 1	5 / 4 / 1	5 / 4 / 1
159 / 403 / 1	16 / 15 / 1	278 / 824 / 3	337 / 1,027 / 3	46 / 45 / 3	**16 / 15 / 1**
132 / 376 / 1	28 / 27 / 1	233 / 674 / 3	226 / 781 / 3	82 / 81 / 3	**28 / 27 / 1**
122 / 311 / 56	11 / 10 / 1	152 / 404 / 3	142 / 381 / 76	31 / 30 / 3	**11 / 10 / 1**
67 / 66 / 1	56 / 245 / 56	122 / 265 / 56	67 / 66 / 1	57 / 201 / 57	57 / 201 / 57
49 / 48 / 1	19 / 18 / 1	67 / 66 / 1	49 / 48 / 1	19 / 18 / 1	19 / 18 / 1
33 / 32 / 1	17 / 16 / 1	49 / 48 / 1	33 / 32 / 1	17 / 16 / 1	17 / 16 / 1
26 / 25 / 1	8 / 7 / 1	33 / 32 / 1	26 / 25 / 1	8 / 7 / 1	8 / 7 / 1
22 / 21 / 1	5 / 4 / 1	26 / 25 / 1	22 / 21 / 1	5 / 4 / 1	5 / 4 / 1

(this is done on the first line.) If yes, this state will be also initial for the suffix in the target and the reverted prefix will be a copy of the prefix. We remember it on lines 2, and 3 and mark the reverted prefix as found.

Algorithm 6: Try to find the reverted prefix

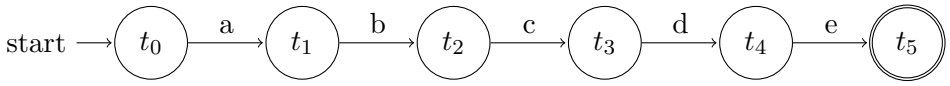
```
   // We are using global variables from Algorithm 5
1  if P.StateCnt == 1 ∧ P.MoveCnt == 0 then
2  |  PstatesFwFinal.Add((T.InitialState, P.InitialState))
3  |  RP = P; prefixFound = true
4  else if P.StateCnt == P.MoveCnt + 1 ∧ P.FinalStatesCnt == 1 then
5  |  var Pstate = P.InitialState; var Tstate = T.InitialState
6  |  for int i = 0; i < P.MoveCnt; ++i do
7  |  |  var Pmove = P.GetMoveFrom(Pstate)
8  |  |  var Tmove = T.GetMoveFrom(Tstate)
9  |  |  if not AreEquivalent(Tmove.Label, Pmove.Label) then
10 |  |  |  failure = true; break
11 |  |  else
12 |  |  |  Pstate = Pmove.Tstate
13 |  |  |  Tstate = Tmove.Tstate
14 |  if not failure then
15 |  |  PstatesFwFinal.Add((Tstate, Pstate))
16 |  |  RP = P; prefixFound = true
```

If not, we check if it is an automaton with only one final state where each state is at most a target of one move and a source of one move (as in Figure 4.9b) on line 4. If yes, we set up auxiliary variables as the initial state in the prefix and the target on line 5. Because we are working with ϵ-free FAs with no dead states in this operation, we can do a for cycle on line 6 to perform a forward simultaneous run in the prefix and the target. Lines 7, and 8 are getting moves from the actual state and on line 9 there is an evaluation of transitions between the prefix and target moves. If there is no intersection of labels of these two moves, we set a failure flag which ends this heuristics with no result. If there is any intersection of these two moves, we move to the next state. If there was no failure in the whole forward run, we do the same on lines 15, and 16 as on lines 2, and 3.

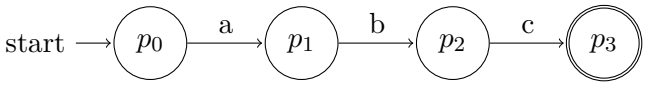
We created an example for easier understanding of this algorithm. If we apply Algorithm 6 on automata in Figure 4.8, we get the state t_3p_3 and store it as a potentially final state in the prefix and the reverted prefix will be created as automaton in Figure 4.8b. (To save space, the format t_3p_3 instead of (t_3, p_3) will be used in text and figures.) The potentially final state in the prefix is a state which is final in the prefix and we have to confirm that there is any continuing way from it in the simultaneous run in the suffix with the target.

Try to Find Reverted Suffix

To try to find the reverted suffix, we have to do almost the same as in previous Algorithm 6 but instead of the forward run, we are performing a simultaneous backward run in the suffix and the target. If the suffix is only one state automaton with no move as in Figure 4.9a, we have to remember all target final states as reverted suffix partially initial states on line 2, and the reverted suffix is constructed as a copy of the suffix and is marked as found in Algorithm 7. (Both the suffix and the target automaton are without dead states.) Partially initial state is a state, which is initial state in the suffix synchronised

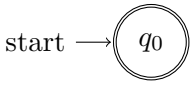


(a) An example of a target automaton

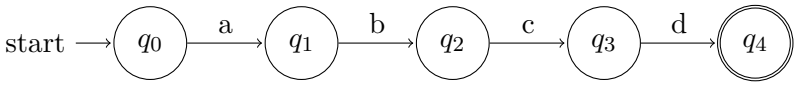


(b) An example of a prefix automaton

Figure 4.8: An example of the prefix and the target automata for applying Algorithm 6 on them.



(a) An example of one-state automaton



(b) An example of a long DFA with only one final state

Figure 4.9: Examples of automata we are trying to detect in our last approach of the reverse concatenation operation.

with a state in the target. There is a possibility of multiple occurs of these states so we call them partially initial states which are later merged into one initial state using Algorithm 11.

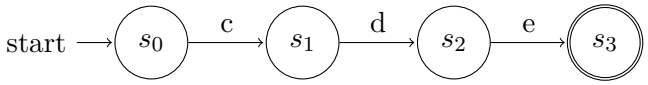
Algorithm 7: Try to find the reverted suffix

```
   // We are using global variables from Algorithm 5
 1 if S.StateCnt == 1 ∧ S.MoveCnt == 0 then
 2     foreach var TfinalState in T.GetFinalStates() do
 3         TstatesSuffixInit.Add(TfinalState)
 4     RS = S; suffixFound = true
 5 else if S.StateCnt == S.MoveCnt + 1 ∧ S.FinalStatesCnt == 1 then
 6     var SstatesToCheck, SstatesToCheckNext
 7     foreach TfinalState in T.GetFinalStates() do
 8         var Sstate = S.FinalState; var Tstate = TfinalState
 9         SstatesToCheck = { (Tstate, Sstate) }
10         for int i = 0; i < SMoveCnt; ++i do
11             while SstatesToCheck.Any() do
12                 (Tstate, Sstate) = SstatesToCheck.Pop()
13                 var Smove = S.GetMoveTo(Sstate)
14                 var Tmoves = T.GetMovesTo(Tstate)
15                 foreach var Tmove in Tmoves do
16                     if Algebra.AreEquivalent(Tmove.Label, Smove.Label) then
17                         SstatesToCheckNext.Add((Tmove.SrcState, Smove.SrcState))
18             SstatesToCheck = SstatesToCheckNext
19             SstatesToCheckNext = { }
20         foreach (var state, _) in SstatesToCheck do
21             TstatesSuffixInit.Add(state)
22     if TstatesSuffixInit.Any() then
23         RS = S; suffixFound = true
```

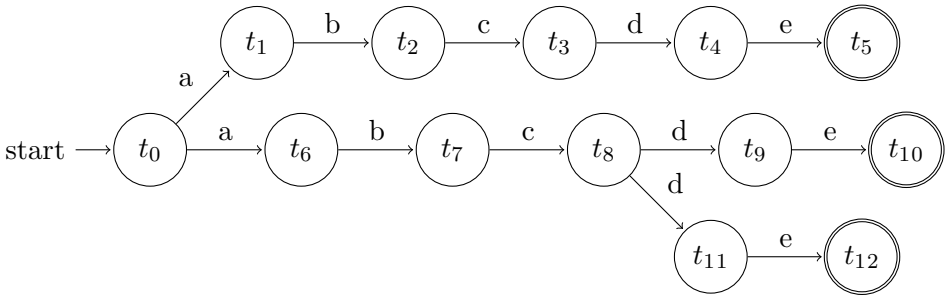
If the suffix is an automaton with only one final state where each state is at most a target of one move and a source of one move as in Figure 4.9b, we are initiating a backward run from all target states, and that one suffix final state on lines 7, 10. There are auxiliary variables for the actual state and a stack for states meant to be check during this run on lines 8 and 9. While there is any state to check in auxiliary stack (lines 11, 12) we are trying to get all moves that have not an empty intersection of labels of the suffix and the target move (line 16). Each such state is pushed into the second auxiliary stack to check in the next iteration (line 17). After the whole while cycle, we move states from the second auxiliary stack into the primary one to check in the following iteration (lines 18, 19).

After the final iteration of each part of the backward run (from each target final state), we mark all states in the auxiliary stack as suffix partially initial states (lines 20, 21). And on the two last lines, we check if our backward run was successful, if yes, the reverted suffix is construct as a copy of the suffix and is marked as found.

If we apply Algorithm 7 on automata in Figure 4.10, we get states t_2s_0 and t_7s_0 as suffix partially initial states, and the reverted suffix will be construct as a copy of the automaton in Figure 4.10a.



(a) An example of the suffix automaton



(b) An example of the target automaton

Figure 4.10: An example of the suffix and target automata for applying Algorithm 7 on them.

Perform a Simultaneous Forward Run in the prefix and the target

If we did not find the reverted prefix, we have to perform a simultaneous forward run in the prefix and the target, so we use the couple consisting of the prefix and target initial states in Algorithm 8 and store it as to check on the first two lines. Then we set the reverted prefix initial state on line 3 and store it into the auxiliary array (line 4). We also have to check if this state is final in the prefix. If yes and we did not find a part of the reverted suffix yet, we will add this state into the array representing all synchronized states in the prefix and the target which are final in the prefix (later called potentially final reverted prefix states, because we to confirm if we can generate them into the reverted prefix automaton by doing a run in the suffix and the target from this state to the suffix and the target final state)

(lines 5, 6, 7). If we found the reverted suffix in Algorithm 7 and it is the final state, we have to check, if it is the target state in in any suffix partially initial state on line 8. If yes, we found a part of the reverted prefix final states.

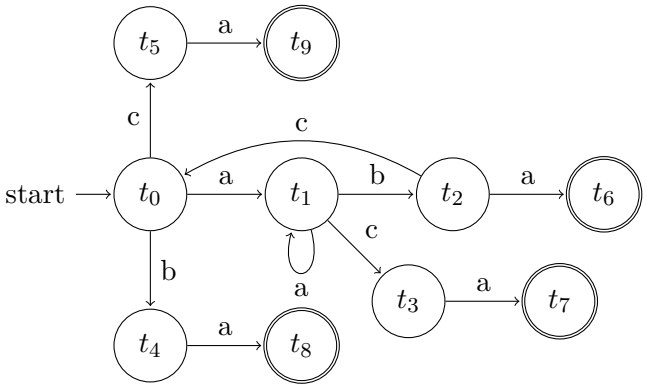
Algorithm 8: Perform a simultaneous forward run in the P and T

```
   // We are using global variables from Algorithm 5
1  var initialPTstate = (P.InitialState, T.InitialState)
2  var PstatesFwToCheck = { initialPTstate }
3  RP.InitState = nextPstateId++
4  PstoredStates[initialPTstate] = RP.InitState
5  if P.IsFinalState(initialPTstate.Pstate) then
6      if !suffixFound then
7          PstatesFwFinal.Add(initialPTstate)
8      else if TstatesSuffixInit.Contains(T.InitialState) then
9          RP.FinalStates.Add(RP.InitState);
10 while PstatesFwToCheck.Any() do
11     var actualState = PstatesFwToCheck.Pop()
12     int actualStateId = PstoredStates[actualState]
13     var psblStates = GetAllPossibleMovesFromState(actualState)
14     foreach psblState in psblStates do
15         var nextState = (psblState.Tstate, psblState.Pstate)
16         if !PstoredStates.TryGetValue(nextState, out var nextStateId) then
17             PstoredStates[nextState] = nextPstateId++
18             PstatesFwToCheck.Add(nextState)
19         if P.IsFinalState(nextState.Pstate) then
20             if !suffixFound then
21                 PstatesFwFinal.Add(nextState)
22             else if TstatesSuffixInit.Contains(nextState.Tstate) then
23                 RP.FinalStates.Add(nextStateId)
24         RP.Moves.Add((actualStateId, nextStateId, psblState.GuardAnd))
```

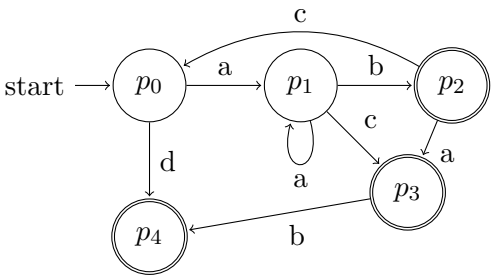
After resolving the initial state, we move on to checking all states to check in this run on line 10. These steps will perform the simultaneous forward run in the prefix and the target. On line 11, we get a state to check, we retrieve its stored information (line 12) and call a function to get all possible states to move from this state on line 13. The last-mentioned function is Algorithm 9

Then in the for loop on line 14, we, for each possible state, set it as a next state (line 15) and try to get its stored information on line 16. If we did not get its information, we were not storing this state before, so we have to store it (line 17) and set it as a next state to check later in next iterations. If the next state is final in the prefix which we are checking on line 19, we have to do the same as on lines 6-9. We also have to store the reverted prefix move from the actual state into the possible state and note the label of this move (line 24).

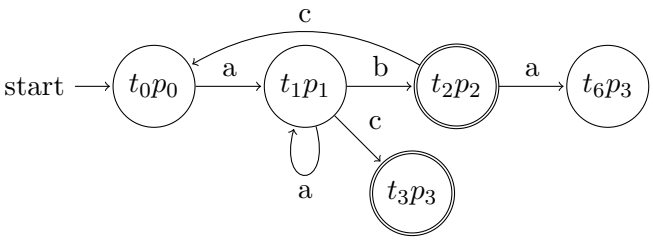
If we apply Algorithm 8 on automata in Figures 4.11a and 4.11b, we receive the automaton in Figure 4.11c as the reverted prefix automaton. We also get two prefix potentially final states which are: t_2p_2 and t_3p_3. Our new reverted prefix has a dead state t_6p_3 which will be removed using the automata library.



(a) An example of a target automaton



(b) An example of a prefix automaton



(c) An example of the reverted prefix automaton after applying Algorithm 8 on automata from Figures 4.11a and 4.11b

Figure 4.11: An example of automata for the simultaneous run in the prefix and the target

Get All Possible Moves From a State

For each simultaneous forward run, we need to know what are our possibilities to move from some state. This `GetAllPossibleMovesFromState(...)` function returns all possible next states. As input, we have to provide the state from which we want to move. It can be either the target and prefix or the target and suffix. The output from this function is a list of all possible moves from a given state in a simultaneous forward run in the target and the (prefix or suffix). Each possible move contains a couple of two moves: move in the target, move in the prefix or suffix, and the intersection of their labels.

So on the second line in Algorithm 9, we create our output array then we get a list of all moves from the given target state. We also get another list of all moves from the given prefix or suffix state. For each combination of these two lists (lines 3, 4) if there is any intersection of their labels (lines 5, 6), we store the move in the target, move in the prefix or suffix, and that intersection of their labels as one item into our output array (line 7) which is returned on the end of this function.

In Figure 4.12 there are two parts of automata. The first part is the prefix or suffix automaton part (Figure 4.12a) and the second one is a part of the target automaton (Figure 4.12b). If we use states q_1 and t_1 from these automata as an input for Algorithm 9, we get two items returned as possible moves from the state t_1q_1.

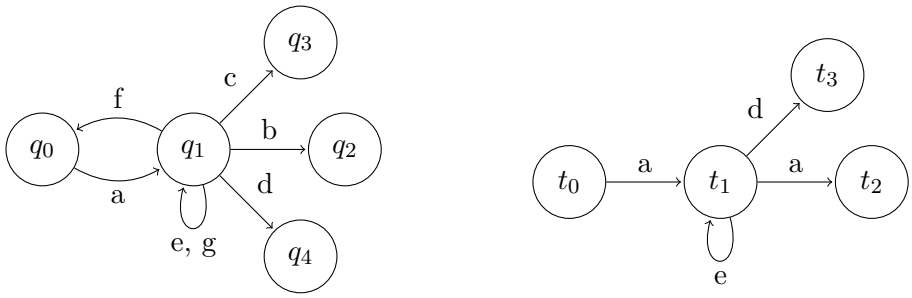
Both items are tuples of two moves and one label: (move in the target; move in the prefix or suffix; intersection of their labels). One of them is $((q_1;\ d;\ q_4);\ (t_1;\ d;\ t_3);\ d)$ and the second one is $((q_1;\ e, g;\ q_1);\ (t_1;\ e;\ t_1);\ e)$.

Algorithm 9: Get all possible moves from a state

Input: Id of a state in the target (Tstate) and id of a state in the prefix (Pstate) or in the suffix (Sstate)

Output: List of all possible moves. Each possible move contains a set of two moves: move in the target, move in the prefix of suffix, and the intersection of their labels.

```
1 Function GetAllPossibleMovesFromState(Tstate, Pstate or Sstate):
2     Stack allPossibleMoves
3     foreach var Tmove in T.GetMovesFrom(Tstate) do
4         foreach var move in (P or S).GetMovesFrom(Pstate or Sstate) do
5             var guardAnd = MakeAnd(move.Label, Tmove.Label)
6             if Algebra.IsSatisfiable(guardAnd) then
                  // if the intersection of two labels is not empty
7                 allPossibleMoves.Push((Tmove, move, guardAnd))
8     return allPossibleMoves
```



(a) A part of the prefix or suffix automaton (b) A part of the target automaton

Figure 4.12: Examples of parts of the target and the (prefix or suffix) automata for Algorithm 9 demonstration.

Perform a Simultaneous Forward Run in the suffix and the target

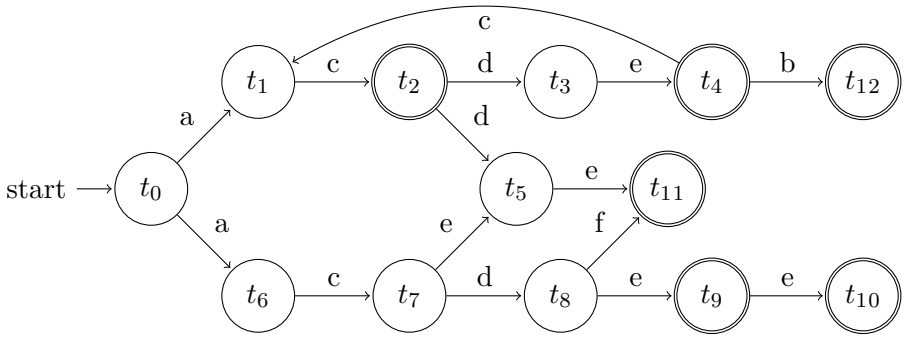
A simultaneous forward run in the suffix and the target is similar to the simultaneous forward run in the prefix and the target but it is somewhat more complicated because we added branches into Algorithm 10. There is an info about a branch added into the info about each state we are storing. This allows us to detect all partially initial states of the reverted suffix automaton.

A branch is a sequence of states and transitions, following each other in the automaton, starting and ending with a state. Branches are generated from the initial states of a simultaneous run in the suffix and the target. Whenever there more transitions from one state in the automaton, the current branch is terminated and we create as many new branches as there are outgoing transitions from the current state. Branch generation is terminated when it is not possible to continue the path, or when we encounter a state that is already in a branch.

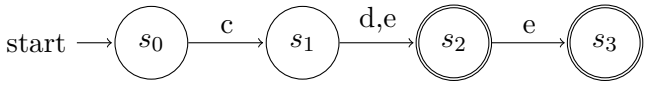
In Algorithm 10, we perform the same for each stored reverted prefix potentially final state. We had to note these states before either in Algorithm 6 by solving the reverted prefix or in Algorithm 8. For each potential prefix final state (line 2), we create a transition into the suffix so we are taking the target state from each potentially prefix final state. This target state is combined with an initial suffix state in order to create a potential reverted suffix

Algorithm 10: Perform a simultaneous forward run in the S and T

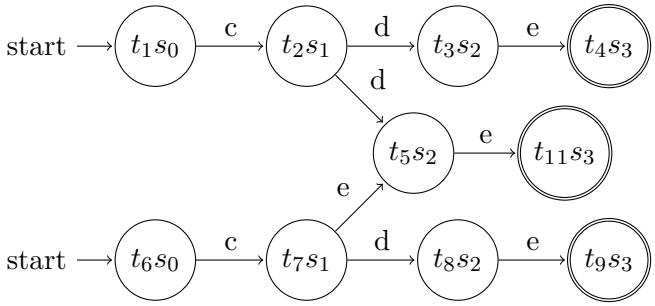
```
   // We are using global variables from Algorithm 5
1  var SstatesFwToCheck, branchesLeadingToEnd
2  foreach (TinitState, PfinalState) in PstatesFwFinal do
3      initialSstate = (TinitState, S.InitialState)
4      SstatesFwToCheck.Add(initialSstate)
5      if SstoredStates.TryGetValue(initialSstate, out var info) then
6          if branchesLeadingToEnd.Contains(info.branch) ∧ !prefixFound then
7              RP.FinalStates.Add(PstoredStates[(TinitState, PfinalState)])
8          RS.InitialStates.Add(info.id)
9          continue
10     info.id = nextSstateId++; info.branch = nextBranchId++
11     RS.InitialStates.Add(info.id); branchesInActualPath = {info.branch}
12     SstoredStates[initialSstate] = info; bool branchLeadingToEnd = false
13     SstatesFwToCheck.Add(initialSstate)
14     if S.IsFinalState(S.InitialState) ∧ T.IsFinalState(TinitState) then
15         RS.FinalStates.Add(info.id)
16         branchLeadingToEnd = true
17     while SstatesFwToCheck.Any() do
18         var actualState = SstatesFwToCheck.Pop()
19         (info.id, info.branch) = SstoredStates[actualState]
20         if info.branch != branchesInActualPath.Peek() then
21             branchesInActualPath.Push(info.branch)
22         var psblStates = GetAllPossibleMovesFromState(actualState)
23         if psblStates.Cnt == 0 then
24             branchesInActualPath.Pop()
25         foreach psblState in psblStates do
26             var nextState = (psblState.State.Tstate, psblState.State.Sstate)
27             if !SstoredStates.TryGetValue(nextState, out var nInfo) then
28                 nInfo.id = nextSstateId++
29                 nInfo.branch = psblStates.Cnt == 1 ? info.branch : nextBranchId++
30                 SstoredStates[nextState] = nInfo
31                 SstatesFwToCheck.Add(nextState)
32             else if branchesLeadingToEnd.Contains(nInfo.branch) then
33                 branchLeadingToEnd = true
34                 foreach branch in branchesInActualPath do
35                     branchesLeadingToEnd.Add(branch)
36             if S.IsFinalState(psblState.Sstate) ∧ T.IsFinalState(psblState.Tstate) then
37                 RS.FinalStates.Add(nInfo.id)
38                 foreach branch in branchesInActualPath do
39                     branchesLeadingToEnd.Add(branch)
40                 branchLeadingToEnd = true
41             RS.Moves.Add(info.id, nInfo.id, psblState.GuardAnd)
42     if branchLeadingToEnd ∧ !prefixFound then
43         RP.FinalStates.Add(PstoredStates[(TinitState, PfinalState)])
```



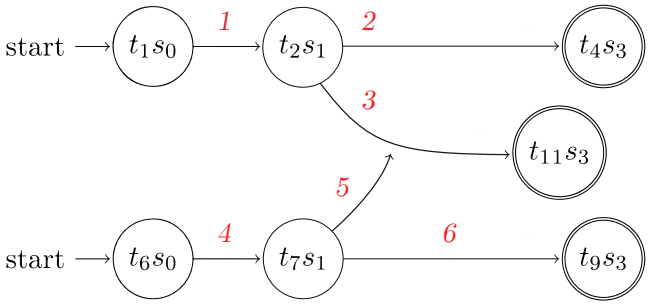
(a) An example of a target automaton



(b) An example of a suffix automaton



(c) An example of the reverted suffix automaton after applying Algorithm 10 on automata from Figures 4.13a and 4.13b



(d) An example of created branches in the reverted suffix automaton from Figure 4.13c.

Figure 4.13: An example of automata for the simultaneous run in the suffix and the target

initial state on line 3. This state is marked as to check on line 4. If we already visited this state (checked on line 5), we mark this state as the initial state for later reverted suffix automaton generation on line 8. If this state also leads to the end (detected by branch which has to be in the auxiliary array containing all branches leading to the end defined on the first line as empty array) and we do not have the reverted prefix found by Algorithm 6, we store the potential prefix final state, used to generate actual partially initial suffix state, for the reverted prefix automaton generation as the final state. We also have to skip to the next prefix potentially final state if we were in this state before (line 9).

If we are in this state for the first time, we create an info for this state containing a new branch info (line 10). We store this state as the reverted suffix initial state and set up its branch into an array containing all branches in the current path (line 11). We store info about this initial state (the state and the branch) and we set an auxiliary variable saying, that any path from this initial state leads to the end, to false on line 12. This initial state is added to the array with states to check on line 13. We also have to check if it is a final state in the target and the suffix. If yes, we store this state as the final reverted suffix state and we are marking that the actual suffix initial state leads to the end (lines 14, 15, 16).

While there are some states in the array with states to check, we pop the last state into the actual state auxiliary variable (line 18). We retrieve its info on line 19. If the the branch from retrieved info is not in the array of branches in the current path, we push it into that array (lines 20, 21). We get all possible states to move from the actual state on line 22 using our function described in Algorithm 9. If there is no such state in returned array, we remove the last branch from the array with branches in the current path (lines 23, 24).

For each possible state (line 25), we mark this state as a next state and check if this next state was already stored before (line 27). If no, we create a info for this state as well as a branch info. Branch is the same as the last branch in the current path if there is only one possible state or a new branch is created (lines 28, 29). If this next state was already stored before, we check if there is the next state branch in the array of branches leading to the end on line 32. If yes then this suffix initial state is marked as leading to the end in line 33 and we copy all branches in the current path into that array with branches leading to the end (lines 34, 35).

If this next state is final in the target as well as final in the suffix, we store this suffix final state on line 37 and do the same as on lines 33–35. Finally, if we found any reverted suffix final state in this iteration, we store the potential prefix final state used to generate the actual partially initial suffix state for the reverted prefix generation as the final state (final two lines).

If we apply Algorithm 10 on automata in Figures 4.13a and 4.13b, we receive the automaton in Figure 4.13c as the reverted suffix automaton. While this automaton generation, we created several branches shown in Figure 4.13d. This automaton has two initial states which have to be later replaced with only one.

Create the Initial State for the Reverted Suffix

While performing the simultaneous forward run in the suffix and the target there is a possibility that we get multiple initial states for the reverted suffix. Our Algorithm 11 can

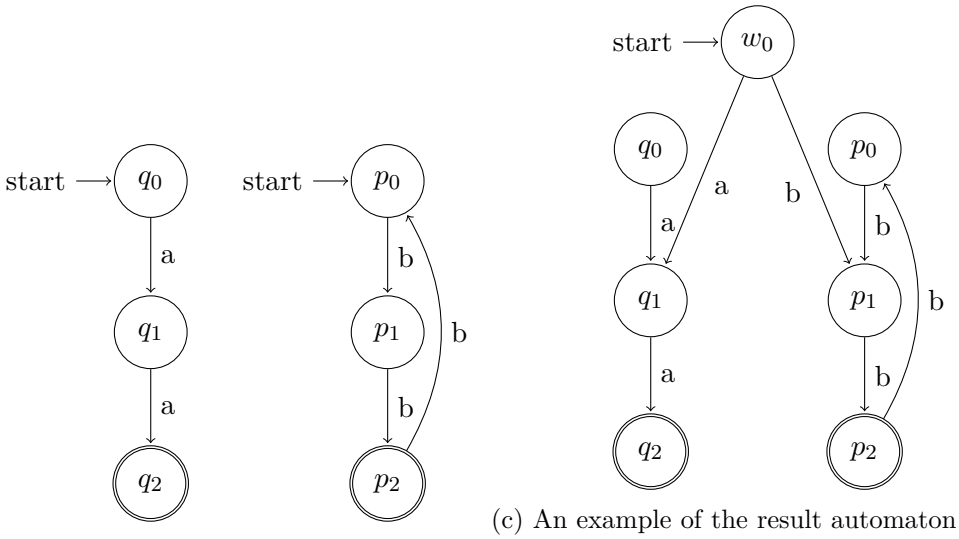
create one initial state from them. We already mentioned this process in Subsection 4.2.2 but this version does not delete dead states because, in our last approach, we are letting the Automata library do this work for us.

Algorithm 11: Create the initial state for the RS

```
   // We are using global variables from Algorithm 5
1  var suffixInitStateIsFinal = false
2  var newInitState = nextPstateId++;
3  foreach state in S.InitialStates do
4      if state is final then
5          suffixInitStateIsFinal = true
6      clone all outgoing moves from state into newInitState
7  if suffixInitStateIsFinal then
8      RP.FinalStates.Add(newInitState)
```

On the first two lines in Algorithm 11, we set an auxiliary variable informing if the initial state should be also a final state to false and set the initial state id. Then for each partially initial state (line 3), we clone all its outgoing moves into our new initial state on line 6 and check if this state is final (line 4). If it is final, then on line 5, the auxiliary variable, informing if the initial state should be also a final state, is set to true. After this cycle, we are checking this auxiliary variable, and based on its value, our new initial noted state as a final reverted suffix state.

For easier understanding of Algorithm 11, if we take two parts of automaton in Figures 4.14a and 4.14b as an input. The result automaton will look like the automaton in Figure 4.14c. We only need to remove the dead state q_0 from it.



(a) The first part of an automaton

(b) The second part of an automaton

(c) An example of the result automaton after applying Algorithm 11 on parts of an automaton from Figures 4.14a and 4.14b

Figure 4.14: An example of automata for Algorithm 11 demonstration

4.3 Optimizations

As mentioned before, we detected some parts of code which we marked as bottlenecks using the Visual Studio 2022 profiler. These parts of code were in the class `AnalysisOperations` and were trying to reduce automata with a special approach, however, more effective was replacing them with a selected reduction. We also analyse other methods and detected, e.g. ineffective epsilon removal in the `AnalysisOperations` class and replaced it with a selected reduction as well.

After the implementation of the third approach of the reverse concatenation operation, we tried to analyze other shortcomings. We managed to propose some improvements in the `Programanalysis` class in the method `InspectWhetherCoveredByOther`. We proposed to ignore all states that were already covered by some other states and to add caching of results into this method described in Subsection 5.2.2.

There were also some places in our third and fourth reverse concatenation operation approach to optimize. We proposed to add one more cache into these approaches and a heuristic that detects if the concatenation of the prefix and suffix is equal to the target. If yes, the prefix is equal to the reverted prefix as well as the suffix to the reverted suffix. Both optimizations are described in Subsection 5.2.1.

Chapter 5

Implementation and Experiments

In this chapter, we provide interesting implementation level details of the ASMA improvements we proposed in Chapter 4.

5.1 Symbolic Simulation Library

As discussed in Section 4.1, we integrated the AutomatonSimulation library written by Juraj Síč into the ASMA tool. We are now going to describe the process of integration of this library.

5.1.1 Integration of the Symbolic Simulation Library

Fortunately, all simulation algorithms created in this library were built on the same Automata library that ASMA uses. We simply took the file `AutomatonSimulation.cs` and placed this file into our version of Automata library. In this file, there is the `AutomatonSimulationExtension` class containing all those simulations we wrote about.

To be able to call them from ASMA, we had to create some kind of interface for simple usage. We added the source code file `AutomatonSimulationExtension.cs` which extends the `AutomatonSimulationExtension` class with an enum containing all reductions names and a method `ReduceSizeBy(automaton, enum_of_method)`.

Unfortunately, there was not any support in the ASMA tool allowing us to simply add any new reduction algorithm. In many code locations, there were hardcoded determinization and minimization. We had to create our own `ITAW` `ITAWWithCustomReduction` interface, wrapper, class, and factory for custom reductions (more information in the thesis [7].) Methods in this class are wrapping automata and can call any reductions on them from the `AutomatonSimulationExtensions` class. The result is that each new reduction algorithm has its new method in this class described in the new interface.

Each important method was calling `Wrap(...)` method and and then a logging method. By adding other important methods (our reductions), we created a lot of duplicate code, so we created the method `WrapAndLog(...)` to improve the logging system and to remove duplicities.

Previous steps allow us only to manually select a reduction type in the source code. We wanted to allow to user to choose which one to use. So we created a new command `ReduceSizeBy` which is able to reduce automata by any given reduction type.

5.1.2 Custom Testing Commands

We needed to have a possibility to test all reduction algorithms on some data sets (examples) and determine which of them is the best one for our benchmarks. So two new testing commands were created. One for testing only one program using a selected reduction algorithm `AnalyseCounters2` and another one `AnalyseCounters3` which can test more programs and returns formatted information about each reduction algorithm such as duration, sizes in different stages, and counts of called operations with their names. To ensure objective results, the second command can also do more runs of each program and the evaluation of the first run can be skipped.

5.1.3 Comparing Reduction Algorithm

After the implementation of commands from Subsection 5.1.2, we were able to compare all reduction algorithms. We ran this command 10 plus one time (the first run at the beginning of each run sequence was skipped for better results) on all available data sets. We had to run some computations only once because of their long computation time but they are not interesting for us (they are marked with * in the table with results.) The obtained results are showed in Table 5.1.

Unfortunately, the results show that all new reduction algorithms are usually much slower than the original combination of determinization and minimization. Only in some cases, the new reductions were faster. The reasons for this somewhat disappointing conclusion deserve some further study, which one is, however, beyond the scope of this work.

5.1.4 Optimization Based on Reductions

We found some useless parts of the code and also some bottlenecks during testing and profiling. This leads to a quite faster implementation by rewriting some codes.

It was mainly in the `AnalysisOperations` class in methods `Assert`, `Replace`, and `RestrictAndCheckVariable`. In these methods there were some parts of the code reducing automata with heuristics. However, a problems was that some of the heuristics built into the code by its original author were slowing the whole analysis according to our benchmark. We found, that more effective was replacing them with a selected reduction.

In the `AnalysisOperations` class in the method `Concat` and in the `PredicateAbstraction` class in the method `Refine`, we added our reduction method which sped up the computation. We present the results in Table 5.2 created using commands from Subsection 5.1.2. This table contains durations of all data sets with and without optimizations.

5.2 Other Improvements

During our analysis of the ASMA tool and later when proposing its improvements, we mentioned some other optimizations which are described in this section.

Table 5.1: A comparision of all the integrated reduction algorithms on all data sets. Values are the average time of 10 runs of the whole analysis in [ms]. Algorithms: `determinize and minimize` (D+M), `reduce size by local simulation optimizied` (LSO), `local simulation` (LS), `simulation no count` (SNC), `simulation no count no opt` (SNCO), `bisimulation` (B). `Reduce size by global simulation` was skipped because it was the slowest and was not able to reduce some automatons in an acceptable time. Names were taken from the thesis [9] and modified.
* Values are data from only one run.

stranger	D+M	LSO	LS	SNC	SNCO	B
01-sanit	2	2	2	2	2	**1**
11-vuln	46	61	32	**26**	31	58
12-sanit	79	47	65	59	**42**	72
21-vuln	153	74	**72**	152	151	171
22-sanit	1,591	414	1,699	**390**	392	2,062
31-vuln	**20,903**	124,596	110,157	64,371	128,858	–
32-vuln	**57,557**	* 347,698	* 330,338	* 173,450	* 342,584	–
33-vuln	**8,193**	61,713	55 864	33,834	61,047	* 152,825
34-sanit	1,094	1,545	**395**	816	1,577	632
35-vABS	**59,692**	* 536,024	* 531,379	* 232,561	* 485,033	–
35-vuln	**80,346**	–	–	–	–	–
36-vuln	**30,505**	* 463,522	* 4,240,726	* 222,220	* 440,287	* 83,948
41-vuln	**46**	394	331	236	390	54
42-sanit	**32**	455	410	590	1,221	5,827

5.2.1 Caching in the Reverse Concatenation Operation

After the implementation of the third approach of the reverse concatenation operation, we tried to analyze it with the Visual Studio 2022 profiler. The result showed an unpleasant fact, namely that the majority of resources are taken by the operation `MakeAnd` in the Automata library. This operation is called recursively and has its own result caching inside it. Unfortunately, usage of this cache starts to be very expensive after a while. For this reason, we have introduced caching in our reverse concatenation operation.

The problem is the `MakeAnd` operation, which, in the current implementation, interacts with the BDD algebra. One possible solution is to try how the analysis of examples would perform if we replace this BDD algebra with another one, for example, if we start working with intervals. We have decided to leave this possibility as future work due to it would require a lot of changes in the code. We have decided to go the way of creating a new method of the same name `MakeAnd`, for easy caching of results of the Automata `MakeAnd` operation, inside our implementation of the reverse concatenation operation. We also tried to remove this cache in the Automata library and let only our new cache. Results were even 3 times slower according to Table 5.3.

We also tried to optimize this reverse concatenation operation (third and fourth version) even more by creating some heuristic that should detect if the concatenation of the prefix and the suffix is equal to the target. If yes, the prefix is equal to the reverted prefix as

Table 5.2: The final comparison of the impact of our whole work on the speed of the ASMA tool. Values are the average time of 10 runs of the whole analysis in [ms]. Explanation: the original implementation of the ASMA tool (Old), after adding all proposed optimization without caching (Opt.), the third reverse concatenation prototype with all proposed optimization without caching (R.C. 3), the third reverse concatenation prototype with all proposed optimization and caching (3+Cch), the fourth reverse concatenation prototype with all proposed optimization without caching (R.C. 4), the fourth reverse concatenation prototype with all proposed optimization and caching (4+Cch).

stranger	**Old**	**Opt.**	**R.C. 3**	**3+Cch**	**R.C. 4**	**4+Cch**
01-sanit	2	**1**	**1**	**1**	**1**	**1**
11-vuln	34	28	26	24	18	**15**
12-sanit	**50**	67	57	51	66	61
21-vuln	139	178	155	50	158	**50**
22-sanit	555	544	336	265	1,489	**247**
31-vuln	14,840	8,030	4,407	3,057	2,773	**2,724**
32-vuln	47,926	25,298	8,027	7,028	7,724	**4,934**
33-vuln	7,683	3,994	2,131	1,312	1,687	**1,254**
34-sanit	1,414	470	547	91	1,047	**146**
35-vABS	63,493	24,079	9,262	5,712	6,477	**4 829**
35-vuln	80,324	42,821	13,616	9,800	8,654	**7,241**
36-vuln	27,160	19,913	18,662	15,613	18,466	**15,170**
41-vuln	**37**	**37**	**37**	**37**	39	39
42-sanit	28	28	28	**27**	30	28

well as the suffix to the reverted suffix. Unfortunately, this seems to be taking a bit more time to compute than the amount of saved time is. Table 5.3 contains the results of the benchmark with this optimization.

5.2.2 Caching in the Program Analysis Class

During profiling in Subsection 5.2.1, we also detected one more possibility of caching. It was in the `ProgramAnalysis` class in the method `InspectWhetherCoveredByOther`. Here we added a new cache right before calling the method which is checking for differences in automata.

Another optimization we purposed in this method was ignoring all states that were already covered by some other states and adding cache into this method before starting the comparison. Unfortunately, this approach was not faster. All these optimizations were benchmarked and the results are in Tables 5.2 and 5.3.

5.2.3 Automatic Display of the DGML

As we mentioned, we wanted to add the possibility for the user to view the result automata after calling an operation from the AsmaCLI during the interactive mode without having to write unnecessary additional commands. So we added the „show“ flag to the „Interactive“ command, which switches the AsmaCLI to the interactive mode. In other words, the user

Table 5.3: Impact of individual optimizations on analysis of all Stranger examples. Values are the average time of 10 runs of the whole analysis in [ms]. Explanation: without any optimization (no o), only with optimized locations of calling reductions (only o), with caching of results in the `ProgramAnalysis` class (1c), with optimized locations of calling reductions and caching of results in the `ProgramAnalysis` class (o+1c), with cache in the reverse concatenation operation (mk), with optimized locations of calling reductions and caching of results in the `ProgramAnalysis` class and cache in the reverse concatenation operation (o1cmk), only the heuristic trying to concatenate the prefix with the suffix and compare it to the target (p+s=t), with optimized locations of calling reductions and caching of results in the `ProgramAnalysis` class and cache in the reverse concatenation operation class but without caching in the Automata library (1cmk-a). We used the third version of the reverse concatenation operation in all program analyses for creating this table.

stranger	no o	only o	1c	o+1c	mk	o1cmk	p+s=t	1cmk-a
01-sanit	1	1	1	1	1	1	1	1
11-vuln	27	24	**24**	**24**	27	**24**	28	32
12-sanit	109	98	**51**	**51**	109	52	63	141
21-vuln	72	82	**50**	51	71	52	164	207
22-sanit	499	532	249	**245**	495	248	1,230	869
31-vuln	5,381	3,952	2,800	2,889	5,258	**2,683**	5,030	9,341
32-vuln	8,158	11,463	7,086	6,835	7,741	**6,746**	9,955	25,902
33-vuln	2,136	1,508	1,220	1,229	2,096	**1,192**	2,605	4,139
34-sanit	1,253	998	166	214	1,248	**172**	1,264	722
35-vABS	6,407	**9,235**	6,220	5,991	6,022	5,937	8,774	23,339
35-vuln	10,619	14,461	9,208	9,234	10,081	**8,768**	11,888	30,014
36-vuln	17,772	16,581	15,655	15,509	17,686	**15,652**	17,701	61,922
41-vuln	38	**36**	38	38	39	38	38	100
42-sanit	29	**27**	29	29	30	29	29	106

can write one command after another without having to call the executable and because of this flag, all result automata will be shown in Visual Studio. To do that, we simply save result automata into a .dgml file (Directed Graph Markup Language) and open it automatically in Visual Studio.

There is a series of commands in Figure 5.1 which produces an output in Visual Studio and there is a screenshot of this output in Figure 5.2.

5.3 Final Experimental Evaluation

To be sure that our improvements were not creating other bottlenecks, we initiated the last profiling. As we expected, reductions are still the most significant operations. However, the reverse concatenation operation has improved considerably. We present our findings in two tables: Table 5.4 containing data from the first profiling on the original version of the ASMA tool and Table 5.5 with the latest profiler data of the actual version. In the results presented in the tables, the results presenting the units seams more interesting. Altogether, we reduced the running by over 90 % for analysis of programs where bigger automata take their place according to Table 5.6.

```
C:\foo\bar\ASMA>AsmaTool.exe -s interactive -show
Stepping is enabled -- some actions might expect you to press any key to continue
Enter command:
$ print-dgml C:\foo\bar\MyAutomaton.json
Enter command:
$ determinize C:\foo\bar\MyAutomaton.json
Enter command:
$ minimize ?
```

Figure 5.1: A series of three commands for the AsmaCLI which is set into the interactive mode with the „show“ option. The tool will show us the result automata in Visual Studio after each operation now. The first command simply loads `MyAutomaton` automaton and stores it into the file with the .dgml format. The second command determinise this automaton. And the last command minimizes the result of the previous operation.

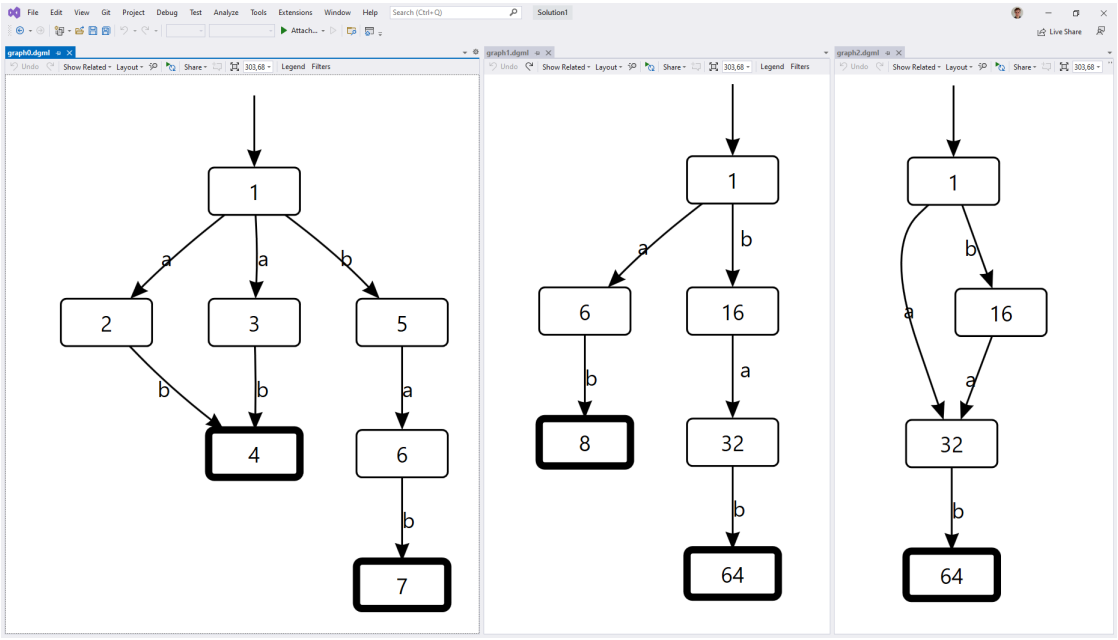


Figure 5.2: The output of the commands from Figure 5.1 in Visual Studio 2022

Table 5.4: A part of the profiling output on the original version. The command used: `analyse-counters C:\foo\bar\stranger-35-vuln.asmasym.txt`. Total CPU [unit, %] gives the samples while executing a function and functions called by this function. The profiler was set to perform 8190 samples per second.

Function Name	**Total CPU [unit, %]**
_ + ProgramAnalysis.Loop()	**636,261** (91,22 %)
___ + CustomCodeProgramAnalysis.ExecuteContextMethod(...)	600,430 (86,08 %)
____ + CustomCode.ConcatNode.ExecuteOn(...)	556,992 (79,85 %)
_____ + Backward.Concat(...)	556,918 (**79,84 %**)
_____ - Asma.TAW<T>.RemoveEpsilons()	358,959 (**51,46 %**)
_____ + BackwardRun.RestrictAndCheckVariable(...)	186,369 (26,72 %)
_____ - Asma.TAW<T>.Determinize()	138,655 (**19,88 %**)
_____ - Asma.TAW<T>.Minimize()	25,994 (3,73 %)
_____ - Asma.TAW<T>.Intersection(...)	21,483 (3,08 %)
[...]	[...]
_____ - Asma.TTW<T, T>.ApplyOn(...)	11,231 (1,61 %)
_____ - Asma.TTW<T, T>.Concat(...)	179 (0,03 %)
[...]	[...]
____ - CustomCode.AssertNode.ExecuteOn(...)	39,433 (5,65 %)
____ - CustomCode.ReplaceNode.ExecuteOn(...)	3,725 (0,53 %)
[...]	[...]
___ - ProgramAnalysis.CheckIsCoveredByOther(...)	35,707 (5,12 %)

Table 5.5: A part of the profiling output on the latest version of the ASMA tool using all improvements proposed. The command used: `analyse-counters3 C:\foo\bar\stranger-35-vuln.asmasym.txt`. Total CPU [unit, %] gives the samples while executing a function and functions called by this function. The profiler was set to perform 8190 samples per second.

Function Name	**Total CPU [unit, %]**
_ + ProgramAnalysis.Loop()	**76,181** (87,33 %)
___ + CustomCodeProgramAnalysis.ExecuteContextMethod(...)	67,285 (77,14 %)
____ + CustomCode.ConcatNode.ExecuteOn(...)	42,314 (48,51 %)
_____ + Backward.Concat(...)	42,213 (**48,39 %**)
_____ - Asma.TAWWithCustomReduction<T>.ReduceSize()	23,992 (**27,50 %**)
_____ - BackwardRun.RestrictAndCheckVariable(...)	13,287 (15,23 %)
_____ - Asma.TAWBase<T>.ReverseConcat4(...)	4,621 (5,30 %)
_____ - Asma.TAWBase<T>.RemoveEpsilons()	287 (0,33 %)
[...]	[...]
_____ + Forward.Concat(...)	95 (0,11 %)
_____ - Asma.TAWBase<T>.Concat(...)	84 (0,10 %)
[...]	[...]
____ + CustomCode.AssertNode.ExecuteOn(...)	19,347 (22,18 %)
_____ - Forward.Assert(...)	19,343 (22,17 %)
[...]	[...]
____ - CustomCode.ReplaceNode.ExecuteOn(...)	5,383 (6,17 %)

Table 5.6: The impact of adding optimizations, caches, and last version of the reverse concatenation operation into the ASMA tool. Values are the average time of 10 runs of the whole analysis in [ms]. Explanation: the original implementation of the ASMA tool (Old), the new implementation of the ASMA tool using all improvements proposed (New), and the speedup calculated as $100 * (1 - New/Old)$ rounded to the nearest whole number.

stranger	**Old**	**New**	**Speedup**
01-sanit	2	1	**50 %**
11-vuln	34	15	**56 %**
12-sanit	50	61	-22 %
21-vuln	139	50	**64 %**
22-sanit	555	247	**55 %**
31-vuln	14,840	2,724	**82 %**
32-vuln	47,926	4,934	**90 %**
33-vuln	7,683	1,254	**84 %**
34-sanit	1,414	146	**90 %**
35-vABS	63,493	4,829	**92 %**
35-vuln	80,324	7,241	**91 %**
36-vuln	27,160	15,170	44 %
41-vuln	37	39	-5 %
42-sanit	28	28	0 %

Chapter 6

Conclusion

The goal of this thesis was to identify weaknesses of the ASMA tool and then propose and implement improvements, with a stress on the correctness of the implementation, its efficiency, and maintainability.

To fulfill our goals, we analyzed the source code of ASMA and ran all benchmarks available with it. Using a profiling tool, we analysed major bottlenecks and identified all major weaknesses of the ASMA tool.

New improvements for the ASMA tool are the integration of the AutomatonSimulation library from Juraj Síč, which required upgrades in the architecture of the ASMA tool allowing ASMA to use algorithms from this library, a possibility for a user to preview automata automatically while using the command-line interface in the interactive mode, four new versions of the reverse concatenation operation and significant optimizations such as caching of results on multiple locations or usage of reductions on optimal places.

To achieve high correctness of our new implementations, we created an extensive set of self-tests for a different proposed implementations of the reverse concatenation operation. Altogether, we reduced the running time by over 90 % for analysis of programs where bigger automata take their place.

Future steps of improving the ASMA tool could be the implementation of combined reduction as suggested in in Section 4.1 or support of antichains mentioned in Section 4.1. The following steps could be advanced refactoring and also finding out if it would be worth to try to convert the calculations in this tool from BDDs to intervals. This is described in more detail in Subsection 5.2.1.

Bibliography

[1] *Veanes, M. The Automata .NET Library.* Microsoft, 2020. Available at: https://github.com/AutomataDotNet/Automata.

[2] Abdulla, P. A., Holík, L., Kaati, L. and Vojnar, T. A uniform (bi-) simulation-based framework for reducing tree automata. *Electronic Notes in Theoretical Computer Science.* Elsevier. 2009, vol. 251.

[3] Abdulla, P. A., Chen, Y.-F., Hol'ik, L., Mayr, R. and Vojnar, T. When Simulation Meets Antichains (On Checking Language Inclusion of NFAs). In: *Proc. of TACAS'10.* Springer, 2010, vol. 6015. LNCS.

[4] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract regular (tree) model checking. *International Journal on Software Tools for Technology Transfer.* 2012, vol. 14, no. 2. DOI: 10.1007/s10009-011-0205-y. ISSN 1433-2779. Available at: http://link.springer.com/10.1007/s10009-011-0205-y.

[5] Češka, M., Vojnar, T. and Smrčka, A. Theoretical Computer Science TIN. *Studijní opora.* 2007. Available at: https://www.fit.vutbr.cz/study/courses/TIN/public/Texty/TIN-studijni-text.pdf.

[6] Holík, L. *Simulations and Antichains for Efficient Handling of Finite Automata.* Brno, 2010. Dissertation thesis. Brno University of Technology, Faculty of Information Technology.

[7] Kotoun, M. *Symbolic Automata for Analysing String Manipulating Programs.* Brno, CZ, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/22529/.

[8] Meduna, A. and Křivka, Z. *Formal Languages and Compilers.* VUT FIT, 2017. Available at: http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/.

[9] Síč, J. *Simulation for Symbolic Automata.* Brno, CZ, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/19745/.

[10] Turoňová, L., Holík, L., Lengál, O., Saarikivi, O., Veanes, M. et al. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages.* ACM New York, NY, USA. 2020, vol. 4, OOPSLA.

[11] Veanes, M. Applications of symbolic finite automata. 2013. Available at: https://link.springer.com/chapter/10.1007/978-3-642-39274-0_3.

Appendix A

Manual to the ASMA Command Line Interface

```
C:\foo\bar\ASMA>AsmaTool.exe --help
You can use variables to speed-up your work with the tool and to avoid repretetive serialization and
deserialization of automatons between commands. A variable name is prefixed with prefix '?'.
You can use variables or filenames (the extension .json is optional) as source/target
automaton/transducer of any command. All command result are stored into implicit variable "?".
Usage: AsmaTool [command] [options]
Options:
  -?|-h|--help                      Show help information
  -s|--enable-interactive-stepping  EnableInteractiveStepping
Commands:
  analyse                           Analyses a program by checking & reporting assertions failures
                                    on all possible execution paths.
  analyse-counters                  Analyses a program by checking & reporting assertions failures
                                    on all possible execution paths. Also prints various counters
                                    for automaton operations.
  analyse-counters2                 Analyses more programs by checking & reporting assertions
                                    failures on all possible execution paths. Also prints various
                                    counters for automaton operations.
  analyse-counters3                 Analyses more programs with all size reductions by checking &
                                    reporting assertions failures on all possible execution paths.
                                    Also prints various counters for automaton operations.
  apply                             Applies the transducer onto the automaton
  batch                             Executes a batch of commands
  complement
  concat
  copy                              Loads automaton or transducer and stores it again (good for use
                                    with variables)
  determinize
  difference
  help                              Prints global help
  interactive                       Starts an interactive shell
  intersect
```

```
  make-epsilon-transducer           Creates a transducer with epsilon updates for the automaton --
                                    the update-expression algebra only of epsilon and identity
  make-identity-transducer          Creates a transducer with identity updates for the automaton --
                                    the update-expression algebra only of epsilon and identity
  make-total
  minimize
  print-dgml                        Prints the automaton as DGML
  print-dot                         Prints the automaton as DOT
  reduce-size-by                    Reduction by selected. Possible selections: 0:
                                    DeterminizeAndMinimize, 1: LocalSimulation, 2:
                                    LocalSimulationOptimizied, 3: SimulationNoCountNoOpt, 4:
                                    SimulationNoCount, 5: GlobalSimulation, 6: Bisimulation
  reduce-size-by-bisimulation       Reduction by bisimulation.
  reduce-size-by-local-simulation   Reduction by LocalSimulation.
  regex-to-char-set-automaton       Creates an automaton from regex with CharSetSolver
  remove-dead
  remove-epsilons
  remove-unreachable
  remove-unreachable-and-dead
  replace-all                       Language-based replacement -- all occurences matching Pattern
                                    are swapped with the Replacement
  show-dgml                         Prints the automaton as DGML and shows it in your default OS
                                    program
  show-dot                          Prints the automaton as DOT, renders into SVG, and shows it in
                                    your default OS program
  string-to-char-set-automaton      Creates an automaton from string literal with CharSetSolver
  test                              Provides various tests for automaton and comparisons of
                                    automata
  transducer-domain                 Creates the automaton representing domain of the transducer
  transducer-image                  Creates the automaton representing image of the transducer
  union
Run 'AsmaTool [command] -?|-h|--help' for more information about a command.
```