BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TRANSDUCERS IN AUTOMATA LIBRARY MATA
PŘEVODNÍKY V AUTOMATOVÉ KNIHOVNĚ MATA

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. DAVID CHOCHOLATÝ

SUPERVISOR
VEDOUCÍ PRÁCE
doc. Mgr. LUKÁŠ HOLÍK, Ph.D.

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

155757

Institut:	Department of Intelligent Systems (DITS)
Student:	**Chocholatý David, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Mathematical Methods
Title:	**Transducers in Automata Library Mata**
Category:	Algorithms and Data Structures
Academic year:	2023/24

Assignment:

1. Familiarize yourself with the finite automata library Mata [1], the work [3], and string solving methods using finite transducers (such methods for solving replaceAll constraints).
2. Design a representation of finite transducers and an appropriate api for the Mata library that can be used to implement transducer based string solving techniques in the string solver Noodler.
3. Implement the proposal in the Mata library and compare the performance of the implementation with available alternatives.
4. Outline a way to extend the string constraint solving algorithm [3] to support relational constraints implemented by finite transducers.

Literature:

1. Mata library. https://github.com/VeriFIT/mata
2. Tomás Fiedor, Lukás Holík, Martin Hruska, Adam Rogalewicz, Juraj Síc, Pavol Vargovcík: Reasoning About Regular Properties: A Comparative Study. CADE 2023: 286-306
3. Frantisek Blahoudek, Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, Juraj Síc: Word Equations in Synergy with Regular Constraints. FM 2023: 403-423

Requirements for the semestral defence:
1, 2

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

Supervisor:	**Holík Lukáš, doc. Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

We implement finite transducers in a new fast and simple automata library Mata. Finite transducers are finite state machines modelling rational relations. Our primary use case for finite transducers is encoding replace operations (replacing a word or a regular pattern with a string literal). A recent automata-based SMT string solver Z3-Noodler uses Mata as a backbone of its decision procedure. Z3-Noodler needs finite transducers to analyse string manipulating programs with replace operations. The analysis of said programs used in web applications prevents software attacks such as cross-site scripting (XSS) or code injection.

The distinctive features of Mata include simplicity (simple to use, modify and extend) and efficiency (fast to run). We design the representation and algorithms for finite transducers to fit the simplicity and efficiency requirements. We inherit and extend the existing data structures and algorithms for finite automata in Mata to represent the finite transducers and their operations. The representation for finite transducers serves as a common data structure and interface for the finite transducers and future representation of automata using multi-terminal binary decision diagrams to handle large alphabets.

We further extend the design with algorithms to construct finite transducers modelling replace operations defined in SMT-LIB.

Finally, we run an experimental evaluation of performance of finite transducers in Mata on a new benchmark with replace operations from runs of Z3-Noodler and from solving problems in pattern matching.

Abstrakt

Implementujeme konečné převodníky do nové rychlé a jednoduché automatové knihovny Mata. Konečné převodníky jsou konečné stavové stoje modelující regulární relace. Naše hlavní použití pro konečné převodníky je kódovaní operací nahrazení (nahrazení slova nebo regulárního vzoru řetězcem). Nový SMT nástroj pro řešení formulí s omezeními nad řetězci Z3-Noodler používá knihovnu Mata jako základ pro jeho rozhodovací proceduru. Z3-Noodler potřebuje konečné převodníky k analýze programů manipulujících s řetězci s operacemi nahrazení. Analýzou zmíněných programů používaných ve webových aplikacích se zabrání útokům jako cross-site scripting (XSS) nebo vložení kódu.

Hlavní odlišující vlastnosti knihovny Mata zahrnují jednoduchost (jednoduchá k užívání, úpravě a rozšíření) a efektivitu (pracuje rychle). Reprezentaci a algoritmy pro konečné převodníky jsme navrhli s ohledem na tyto vlastnosti knihovny.

K reprezentaci konečných převodníků a jejich algoritmů znovupoužijeme a rozšíříme existující datové struktury a algoritmy pro konečné automaty v knihovně Mata. Reprezentace pro konečné převodníky slouží jako společná reprezentace pro konečné převodníky a budoucí reprezentaci automatů využívajících multi-terminálních binárních rozhodovacích diagramů pro práci s velkými abecedami.

Navíc rozšíříme návrh o algoritmy pro konstrukci konečných převodníků modelujících operace nahrazení definovaných v SMT-LIB.

Nakonec experimentálně vyhodnotíme efektivitu konečných převodníků v knihovně Mata na nové sadě příkladů s operacemi nahrazení z běhů nástroje Z3-Noodler a z řešení problémů nalezení vzoru.

Keywords

finite transducers, finite automata, replace operations, string solving, string constraints, SMT, efficiency, simplicity, verification, cross-site scripting, code injection, browser transductions, composition, projection, application, nondeterminism

Klíčová slova

konečné převodníky, konečné automaty, operace nahrazení, řešení řetězcových problémů, řetězcová omezení, SMT, efektivita, jednoduchost, verifikace, cross-site scripting, vložení kódu, prohlížečové transdukce, kompozice, projekce, aplikace, nedeterminismus

Reference

CHOCHOLATÝ, David. *Transducers in Automata Library Mata.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Mgr. Lukáš Holík, Ph.D.

Rozšířený abstrakt

Konečné automaty nachází mnohá použití, ať už v oblasti výzkumu, nebo v průmyslu. Práce s konečnými automaty může ale v některých případech být výpočetně náročná. Z toho důvodu se postupně objevil značný počet automatových knihoven, každá s vlastními podporovanými typy automatů a operacemi nad nimi. Návrh implementace každé z knihoven dává určité výhody a nevýhody. Nedávno byla předvedena nová automatová knihovna Mata. Mata je již nyní používána pro uvažování nad regulárními výrazy a pro řešení omezení nad řetězci. Aby však Mata dobře pracovala i v jiných doménách jako regulární *model checking* nebo rozhodování logik jako WS1S nebo kvantifikovaná Presburgerova aritmetika, podpora pro konečné převodníky je nutná. V této práci přidáváme konečné převodníky do knihovny Mata.

Podobně jako konečné automaty, konečné převodníky jsou konečné stavové stroje. Odlišují se od konečných automatů tím, že pracují s více paměťovými páskami (zatímco konečné automaty pracují pouze s jednou páskou). Každý běh reprezentuje n-tici slov (každá páska reprezentuje jedno slovo). Zatímco konečné automaty modelují regulární jazyky, konečné převodníky modelují relace mezi regulárními jazyky. Konkrétně, konečné převodníky modelují množinu n-tic slov. Konečné převodníky mají rozsáhlé využití v mnoha aplikačních doménách jako zpracování řeči a jazyka (výpočetní lingvistika), překladače programů, parametrické *unit* testování, paralelizace kódu, webová bezpečnost, nebo verifikace programů.

Hlavními charakteristikami knihovny Mata určujícími povahu této práce je efektivita a jednoduchost. Efektivita automatových algoritmů je důležitá pro mnohé příklady použití, kde jsou automatové problémy výpočetně náročné a každé zpomalení práce s automatem způsobuje značné zpomalení celkově. Mnohé automatové knihovny jsou dobře optimalizované na vybranou množinu operací, předpokládají určitý způsob práce s knihovnou, nebo limitují, jak je možné knihovnu využívat. Cílem knihovny Mata je dosáhnout rychlého výpočtu operací v různých aplikacích, poskytovat optimalizované obecné algoritmy i speciální konkrétně zaměřené algoritmy, obzvláště pro její aplikaci v uvažování o omezeních nad řetězci. Jednoduchost knihovny se zaměřuje na dodání pochopitelného a přístupného rozhraní pro automatové operace, jednoduchou integraci knihovny do existujících nástrojů a jednoduchou rozšiřitelnost a upravitelnost implementovaných algoritmů a automatových modelů. Mata využívá datových struktur navržených tak, aby umožnily knihovně rychlé výpočty, ale také jejich jednoduchou modifikaci uživatelem pro vlastní použití a poskytující rozšiřitelné rozhraní pro přidávání nových automatových modelů, nebo speciálních operací.

Naším prvním a hlavním případem použití pro konečné převodníky v knihovně Mata je uvažování o omezeních nad řetězci. V posledních dvou desetiletích se oblast uvažování o řetězcových omezeních dočkala značné pozornosti od výzkumu i z praxe. Výzkum je hlavně motivován použitím nástrojů pro analýzu programů pro webové aplikace. Uvažování o omezeních na řetězci zabraňuje útokům jako cross-site scripting (XSS) nebo code injection objevováním zranitelností v programech. Existuje velké množství SMT nástrojů pro uvažování nad řetězcovými omezeními, každý implementující vlastní techniky a optimalizace.

Použití přístupů k uvažování o omezeních nad řetězci založených na automateh nedávno získalo značnou popularitu. Použití automatů pro uvažování o omezeních nad řetězci je přirozeným krokem, neboť řetězcová omezení obsahují hodně regulárních výrazů a regulárních omezení. Tato práce je značně motivována existujícími technikami založenými na automatech nedávno uvedenými v nástroji pro uvažování o omezeních nad řetězci Z3-Noodler. Z3-Noodler používá automaty jako hlavní podstatu jeho rozhodovací pro-

cedury. Z3-Noodler umí řešit řetězcové rovnice s regulárními omezeními, ale nedokáže jednoduše řešit omezení obsahující operace nahrazení jako `replaceAll` (nahrazení všech výskytů vzoru za řetězec). Jednou z hlavních motivací pro tuto práci je kódování operací nahrazení v nástroji Z3-Noodler.

V této práci navrhneme reprezentaci a algoritmy pro konečné převodníky v knihovně Mata. Upravíme návrh tak, aby splňoval požadavky knihovny na jednoduchost a efektivitu. Modelujeme konečné převodníky pomocí zdědění existujících datových struktur pro konečné automaty v knihovně Mata. Rozšíříme datové struktury pro konečné převodníky jako konečný automat s úrovněmi pro stavy. Tento přístup nám umožňuje zdědit také velké množství existujících operací pro konečné automaty s pouze mírnými úpravami pro funkčnost s konečnými převodníky. Reprezentace konečných převodníků zároveň slouží jako datová struktura i pro automaty s multi-terminálními binárními rozhodovacími stromy pro práci s velkými abecedami. Dále rozšíříme konečné převodníky algoritmy pro konstrukci konečných převodníků modelujících operace nahrazení definované v SMT-LIB. Nakonec experimentálně porovnáme čas běhu konečných převodníků implementovaných v knihovně Mata na nové sadě příkladů získaných z běhů nástroje Z3-Noodler. Zvolený přístup se zdá být vhodný, a přestože pro uvažování o řetězcových operacích je Mata pomalejší, pracuje dostatečně rychle pro naše případy použití. Naopak na příkladech z oblasti nalezení vzorů schopnost knihovny Mata pracovat s nedeterminismem jasně překonává deterministické přístupy.

Transducers in Automata Library Mata

Declaration

Acknowledgements

I would like to thank my supervisor, doc. Mgr. Lukáš Holík, Ph.D., who has allowed me to contribute to his research projects for several years now, provided essential and necessary information about the topic of this thesis, outlined possible solutions and answered every question I have had throughout the whole time.

Contents

Chapter 1

Introduction

Finite automata can be found in numerous fields, both in research and in the industry. In some use cases, handling finite automata can be computationally costly. Therefore, a number of automata libraries have emerged throughout the years, each with their own set of supported automata types and operations. Their respective design and implementation decisions give various advantages and disadvantages to each library. Recently, a new finite automata library called MATA [29] has been introduced. MATA is already used to *reason about regular expressions* and in *string solving*. However, in order for MATA to work well in other applications such as *regular model checking*, string solving with advanced constraints, or *deciding logics* such as *WS1S* or *quantified Presburger arithmetic*, support for *finite transducers* is necessary. In this work, we add *finite transducers* to MATA.

The same as finite automata, finite transducers are finite state machines. They differ from finite automata by working with *multiple memory tapes* (whereas finite automata work with only one memory tape). Each run represents an n-tuple of words (each tape represents a single word). While a finite automaton models a regular language, a finite transducer models a rational relation. That is, an n-tape finite transducer models a set of n-tuples of words. Finite state transducers are of immense practical use in numerous domains such as language and speech processing (computational linguistics) [65], software compilers [7], parameterized unit testing [82], code parallelization [83], web security [52], or verification of software programs [35].

The two distinctive features of MATA, which determine our work in this thesis, are *efficiency* and *simplicity*. The *efficiency* of automata algorithms is crucial since in many use cases, automata problems are computationally hard and every small decrease in performance of the often used low-level operations causes a significant slowdown. Many automata libraries are well-optimized on only a subset of operations, presume a certain specific approach to using them, or make limitations on how one can utilize the library. MATA wants to achieve fast computation on operations in various applications, providing optimized general operations on finite automata as well as several purpose-specific algorithms, especially for its applications in string solving. The *simplicity* of MATA focuses on delivering an approachable interface for automata operations, easy integration of MATA into the existing tools, and further extensibility and modifiability of implemented algorithms and automata models. The underlying data structures in MATA are designed to make it possible for MATA to run sufficiently fast, but also enable *easy modification* by the user for their own use cases, and give an *extensible* interface for adding new automata models, or purpose-specific operations. The design decisions behind MATA, and a comparison with other automata libraries are further discussed in Chapter 3.

Our first and primary use case for finite transducers in Mata, beside others, is string solving. In the last two decades, string solving enjoys a lot of attention from the research and the industry. The work on string solving is primarily motivated by its uses in analysis of programs for web applications. String solving allows us to prevent software attacks such as cross-site scripting (XSS) or code injection by discovering security vulnerabilities in the programs. There exists numerous SMT string solvers, each applying their unique techniques and optimizations for specific operations.

The use of automata-based techniques has recently gained noticeable popularity among the string solving community. Using automata for string solving is a natural approach: we have to deal with a lot of regular expressions and regular constraints in the solved formulae. The work in this thesis is highly motivated by an automata-based technique introduced recently in a novel SMT string solver Z3-Noodler. Z3-Noodler uses automata as the backbone of its decision procedure. Z3-Noodler can solve word equations with regular constraints, but cannot be used to solve word equations and constraints with replace operations such as `replaceAll` (replacing all words or regular patterns with a string literal), as explored in [22] for analysis of string manipulating programs in web applications. One of the main motivations for adding finite transducers to Mata is encoding replace operations in Z3-Noodler. The broader context of string solving and related work relevant to this thesis is given in Chapter 4.

In this work, we design the representation and algorithms for finite transducers in Mata. To maintain the simplicity and efficiency requirements of Mata, we want to keep the data structures and algorithms for finite transducers simple. Since finite transducers have normally n-tape alphabets, a support for special n-tape symbols would have to be introduced to Mata, with special operations and data structures supporting n tapes. We did not want to introduce a new data structure just for finite transducers, or complicate the existing data structures. Therefore, we designed a representation of finite transducers and algorithms for them in such a way that we reuse the existing data structures and algorithms already implemented in Mata without having to do too many modifications. For the representation of finite transducers, we inherit the data structure for finite automata, and only extend the data structure with a mapping of states to n levels, each corresponding to one tape. We represent the n-tape symbols as a simple sequence of single finite automata symbols (a sequence of finite automata transitions with states annotated with a level, one state and transition for each tape). All data structures for finite automata are therefore directly reusable for finite transducers. This also allows us to inherit large portions of finite automata operations for finite transducers with only slight modifications. No special data structures need to be added, and the encoding of a finite transducer as a finite automaton with levels is intuitive.

In order to achieve this, we had to solve technical problems regarding especially epsilon transitions. A single epsilon transition between the subsequent tapes represents a no-operation transition, where the transducer does not read/write anything on the tape. We also work on a parallel project where we need to handle automata with large alphabets using automata with multi-terminal binary decision diagrams (called BDDAs here) as seen in an automata library Mona [44] (see Section 3.2). BDDAs also use n-tape symbols. The representation for n-tape transducer transitions is very close to BDD-like transitions in Mona, integrating MTBDDs into the transition relation. Therefore, the representation for finite transducers simultaneously serves as a common representation for future implementation of BDDAs. BDDAs also use jumps (special transitions performing multiple transitions in a single transition) and don't care symbols (matching any symbol in the alphabet).

Hence, we further design epsilon and don't care jumps for finite transducers. By only using levels, we are able to encode epsilon transitions (a simple change of a state as for the finite automata) between states with the same level. Levels further encode epsilon jumps (no operation is performed on the jumped over tapes), and don't care jumps (any symbol is accepted on the jumped over tapes) between states with different levels. We restrict the jumps to be allowed to jump only up to the next state with the level of the first tape. This allows us to use the existing algorithms such as subset and product construction for the finite transducers without modifications and without having to store the information about lengths of jumps into a pair of states in a worklist. Depending on whether one works with finite transducers or BDDAs, a different interpretation of jumps is possible with a minimal change in the algorithms, all with a single transition representation: a jump in a finite transducer repeats the symbol jumped over on all tapes, whereas in BDDAs, the jumped over symbol is used only for the first tape jumped over, followed by a sequence of don't care transitions. This is to our best knowledge a unique approach for encoding finite automata, transducers and BDDAs with a single automaton data structure and shared algorithms with only minimal modifications and support for epsilon and don't care transitions.

We further extend the design with algorithms to construct finite transducers modelling replace operations defined in SMT-LIB [13].

Finally, we run an experimental evaluation of the performance of finite transducers in Mata on a new benchmark with replace operations from runs of Z3-Noodler and solving problems in pattern matching. The experimental results show that even though Mata is slower than the state-of-the-art, it performs well enough for our use cases. Moreover, the support for nondeterminism in Mata allows for great performance of NFTs on complex regular expressions found in practice.

The described design is feasible. Finite transducers perform sufficiently well and are prepared for Z3-Noodler and string solving in general.

Contribution. The contributions of this work can be summarized as follows:

- Design of data structures and algorithms for finite transducers in automata library Mata.
- Implementation of the proposed data structures and algorithms in Mata.
- A new benchmark of replace operations encoded as finite transducers, derived from SMT-LIB benchmarks from real problems and pattern matching.
- Experimental evaluation of the implemented algorithms on a variety of benchmark problems from practice.

Chapter 2

Preliminaries

In this chapter, we will define several terms and notions used throughout this thesis. We follow the usual definitions of automata theory, as used in works such as [38] or [78] with custom modifications and additions.

Alphabets. We define an *alphabet* $\Sigma = \{a, b, c, \ldots\}$ as a set of *symbols* where symbols are usually denoted by $a, b, c, \ldots$. Σ^* denotes a set of all finite *words* (*literals* or *strings*) over the alphabet Σ. We usually denote words $u, v, w, \ldots$. $u[n : m]$ represents a substring of u containing symbols in the interval of indices (indexed from 0) between n and m (including the symbol on the index n and excluding the symbol on the index m). Furthermore, we use two special symbols: an *epsilon symbol* $\epsilon \notin \Sigma$, representing an empty symbol (or word, $\epsilon \in \Sigma^*$), and a *don't care symbol*, denoted ?, representing any single symbol from Σ. *Alphabet with epsilon symbols* (epsilons) is denoted $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$.

We denote *concatenation* on words w_1, w_2 using $w_1 \cdot w_2$, sometimes for convenience omitting the concatenation operator ($w_1 w_2$), where ϵ is the neutral element of concatenation ($\epsilon \cdot w = w \cdot \epsilon = w$).

2.1 Finite Automata

In this section, we lay foundation to finite automata and their features utilized in this work.

Finite automata are finite state machines used to represent a regular language, comprising a set of words. Operations on finite automata represent various set operations.

We can work with either deterministic, or non-deterministic finite automata, called DFA, or NFA, respectively. An intuitive encoding of problems into automata often leads to NFAs, while automata operations on DFAs are usually faster and simpler.

Definition 1 (**Nondeterministic finite automaton**).
A *nondeterministic finite automaton* (*NFA*) over the alphabet Σ is a 5-tuple

$$\mathcal{A} = (Q, \Sigma, post, I, F)$$

where

- Q is a finite set of *states*,
- Σ is the alphabet of $\mathcal{A}$,

- *post* $: Q \times \Sigma \to 2^Q$ is a *symbol-post function* where $q \xrightarrow{a} q'$ (or (q, a, q')) for $q' \in post(q, a)$ is a *transition*,
- $I \subseteq Q$ is a finite set of *initial states*, and
- $F \subseteq Q$ is a finite set of *final states*.

A set of all transitions of $\mathcal{A}$ forms a transition relation of $\mathcal{A}$, denoted Δ.

$$\Delta(q, a) \equiv post(q, a)$$

.

Furthermore, we define a *state-post function* $post(q) = \{(a, post(q, a)) \mid post(q, a) \neq \emptyset\}$ for a state q. Symbol-post function and State-post function are called *post-image functions*. *post* can be generalized to a set of source states over a given transition symbol, $post(S, a)$ where $S \in Q$ and $a \in \Sigma$ as $post(S, a) = \bigcup_{s \in S} post(s, a)$.

Definition 2 (**NFA with epsilon symbols**).
An *NFA with epsilon symbols* is a 5-tuple $\mathcal{A} = (Q, \Sigma_\epsilon, post, I, F)$ where

- Q, I, F are the same as for normal NFA, and
- symbol-post function *post* $: Q \times \Sigma_\epsilon \to 2^Q$ allows epsilon symbols.

We will often denote NFA with epsilons as just NFA when it is clear whether we allow epsilon transitions or not in regard to the context.

We define a *run* (*path*) of $\mathcal{A}$ over a word $w \in \Sigma^*$ as a sequence of states and symbols $q_0 a_1 q_1 a_2 \dots a_n q_n$ where $\forall 1 \leq i \leq n : q_i \in post(q_{i-1}, a_i) \wedge a_i \in \Sigma_\epsilon \wedge w = a_1 a_2 \dots a_n$. We further distinguish runs as *accepting runs* and *non-accepting runs* (*not accepting runs*). A run is accepting if and only if $q_0 \in I$ and $q_n \in F$. A (potentially infinite) set of words for which there exists an accepting run of $\mathcal{A}$ defines a regular language $\mathcal{L}(\mathcal{A}) \subseteq \Sigma_\epsilon$ of NFA $\mathcal{A}$. $\mathcal{L}(\mathcal{A})^{<i}$ ($\mathcal{L}(\mathcal{A})^{\leq i}$) means a regular language $\mathcal{L}(\mathcal{A})$ containing only words of at most length $i - 1$ (i).

States Q can be divided into *useful states* and *useless* states. A state q is useful if there exists an accepting run over states $q_0, q_1, \dots, q_n$ where $q \in \{q_0, q_1, \dots, q_n\}$. Otherwise, q is useless.

We also distinguish between *reachable* and *unreachable* states. A state q is reachable if there exists a path $q_0 a_1, \dots, a_n q$ in $\mathcal{A}$ such that $q_0 \in I$.

Deterministic finite automaton. We call an NFA $\mathcal{A} = (Q, \Sigma, post, I, F)$ a *deterministic finite automaton* (*DFA*) iff $\forall q \in Q, a \in \Sigma : |post(q, a)| \leq 1$, and $|I| = 1$.

A conversion from NFA to DFA, called *determinization*, is possible (NFAs and DFAs have the same expressive power), but it is a computationally expensive operation.

Algorithms. Often times a computation of a *minimal*, or at least *minimized* automaton can improve performance of operations performed on the automaton, but the operation itself is expensive and choosing the ideal minimization method (Brzozowski's [20], Hopcroft's [53], ...) for typical structures of finite automata appearing in one's problems might be hard.

However, NFAs can succinctly represent large state space. This prevents exponential state-space explosion which is characteristic for working with DFAs, e.g., inclusion testing. The disadvantage of non-determinism is that NFAs require more complex algorithms, such as simulation-based reduction [75, 51, 45].

Definition 3 (**Powerset (subset) construction**).
The algorithm of powerset (subset) construction creates a deterministic finite automaton from its equivalent non-deterministic finite automaton. Powerset construction produces a DFA A', where $Q' = 2^Q$, $F' = \{S \in Q' | S \cap F \neq \emptyset\}$, $I' = I$ and for $S \in Q'$:

$$post'(S, a) = \bigcup_{s \in S} post(s, a).$$

Definition 4 (**Product construction**).
Product construction is an algorithm where, given two NFAs $A_1 = (Q_1, \Sigma, post_1, I_1, F_1)$ and $A_2 = (Q_2, \Sigma, post_2, I_2, F_2)$ over an alphabet Σ, the algorithm yields a product NFA A as a 5-tuple deterministic finite automaton $A = (Q, \Sigma, post, I, F)$ where:

- $Q = Q_1 \times Q_2$,
- $post : Q \times \Sigma \to P(Q)$,
- $I = I_1 \times I_2$, and
- $F = F_1 \times F_2$.

post is constructed as $([q_1, q_2], a) = post_1(q_1, a) \times post_2(q_2, a)$ where $[q_1, q_2]$ denotes a pair of states, often called a *macrostate* or a *product state*. For pairs of states $q_1 \in Q_1$ and $q_2 \in Q_2$ and a common transition symbol a of transitions $q_1' \in post_1(q_1, a)$ and $q_2' \in post_2(q_2, a)$, a single product transition is denoted as $[q_1, q_2] \xrightarrow{a} [q_1', q_2']$, where $[q_1', q_2'] \in post([q_1, q_2], a)$ for the corresponding states $[q_1, q_2]$ and $[q_1', q_2']$ in A.

When applied to computing an *intersection* of NFAs, the language of A is equal to $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

In this work, we will utilize the classic product construction algorithm, as shown in Algorithm 1.

```
Input  : NFA $A_1 = (Q_1, \Sigma, post_1, I_1, F_1)$, NFA $A_2 = (Q_2, \Sigma, post_2, I_2, F_2)$
Output: NFA $A = (A_1 \cap A_2) = (Q, \Sigma, post, I, F)$ with
         $\mathcal{L}(A_1 \cap A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$

1  $Q, post, F \leftarrow \emptyset$
2  $I \leftarrow I_1 \times F_2$
3  $W \leftarrow I$
4  while $W \neq \emptyset$ do
5      pick $[q_1, q_2]$ from $W$
6      add $[q_1, q_2]$ to $Q$
7      if $q_1 \in F_1$ and $q_2 \in F_2$ then
8          add $[q_1, q_2]$ to $F$
9      forall $a \in \Sigma$ do
10         forall $q_1' \in post_1(q_1, a), q_2' \in post_2(q_2, a)$ do
11             if $[q_1', q_2'] \notin Q$ then
12                 add $[q_1', q_2']$ to $W$
13             add $[q_1', q_2']$ to $post([q_1, q_2], a)$
```

Algorithm 1: Product construction algorithm in its classic implementation.

Applications. Many problems can be efficiently encoded into NFAs and solved by performing operations on NFAs, such as membership or emptiness testing, reachability testing and more.

Further improvements to automata theory may be widely applicable to large-scale automata-based technologies, e.g., pattern matching, analysis and verification of complex systems, analysis of genetic information, run-time monitoring, deciding logics (linear integer arithmetic or monadic second-order logic, ...).

2.2 Finite Transducers

In this section, we define *finite state transducers* (further also called *finite transducers* or just *transducers*) and the corresponding operations.

Finite transducers are finite state machines with multiple memory tapes. Each run of a finite transducer represents an n-tuple of words. While a finite automaton models a regular language, finite transducers model *rational relations* (or *regular* relations), subsets of the Cartesian product of regular languages. That is, a finite transducer models a set of n-tuples of words.

Henceforth, each 2-tape finite transducer can be thought of as a translator translating (transducing) between the languages: a 2-tape finite transducer precisely models a translator from an input language to an output language, modelling a so-called *binary rational relation*), a subset of $\Sigma_1^* \times \Sigma_2^*$. In this work, we usually work with binary rational relations, but in general an *n-ary rational relation* can be modelled by an n-tape transducer.

Definition 5 (**Nondeterministic finite transducer**).
An n-tape *nondeterministic finite state transducer* (*NFST*; *nondeterministic finite transducer*, *NFT*) over an alphabet Γ is a 5-tuple $\mathcal{T} = (Q, \Gamma, \mathit{post}, I, F)$ where

- Q is a finite set of *states*,
- $\Gamma = (\Sigma_\epsilon)^n$ is an n-tape alphabet of $\mathcal{T}$,
- $\mathit{post} : Q \times \Gamma \to 2^Q$ is a *symbol-post function* where $q \xrightarrow{\gamma} q'$ (or (q, γ, q')) for $q' \in \mathit{post}(q, \gamma)$ is a *transition* for $\gamma = [\, a^1 \ a^2 \ \dots \ a^n \,] = (a^1, a^2, \dots, a^n) \in (\Sigma_\epsilon)^n$,
- $I \subseteq Q$ is a finite set of *initial states*, and
- $F \subseteq Q$ is a finite set of *final states.*

NFT $\mathcal{T}$ is syntactically an NFA over Γ. $\mathcal{T}$ accepts word $\bar{w} = \gamma_1 \circ \gamma_2 \circ \dots \circ \gamma_m = (a_1^1, a_1^2, \dots, a_1^n) \circ (a_2^1, a_2^2, \dots, a_2^n) \circ \dots \circ (a_m^1, a_m^2, \dots, a_m^n)$ if there exists an accepting run of $\mathcal{T}$ for $\bar{w}$ where $\circ$ is a concatenation operator performing component-wise concatenation over tuples: $(a_1^1, a_1^2, \dots, a_1^n) \circ (a_2^1, a_2^2, \dots, a_2^n) = (a_1^1 a_2^1, a_1^2 a_2^2, \dots, a_1^n a_2^n)$. When the context is clear, we sometimes omit $\circ$, similarly as for NFA.

An alternative definition of NFTs could allow for each tape to have a different alphabet, resulting in $\Gamma = \Sigma_1 \times \Sigma_2 \times \dots \times \Sigma_n$. In this thesis, we consider only NFTs where all tapes have the same alphabet, but generalization to varying alphabets is possible and all algorithms and approaches presented in this work can be easily modified to support this alternative definition.

We use $\mathrm{Id}(\Sigma)$ to symbolically denote a set of identity transition symbols between two states, that is, for states $q, q' \in Q$, transitions

$$(q, \mathrm{Id}(\Sigma), q') = \{(q, \gamma, q') \mid \gamma = (a^1, \dots, a^n) \in \Sigma^n \wedge a^1 = \dots = a^n\}.$$

We represent an NFT which is constructed as an identity NFT for a given NFA, denoted as $\mathcal{T}_{\mathrm{Id}(\mathcal{A})}$. $\mathcal{T}_{\mathrm{Id}(\mathcal{A})}$ is constructed from $\mathcal{A}$ by extending the existing $\mathcal{A}$ transitions by repeating the same transition symbol as is the existing transition symbol in $\mathcal{A}$ for each tape.

A (potentially infinite) set of words for which there exists an accepting run of $\mathcal{T}$ defines a *rational relation* $R(\mathcal{T}) \subseteq (\Sigma_{\epsilon}^{*})^{n}$ of NFT $\mathcal{T}$.

Unless it is clear from the context, we will explicitly differentiate between an *NFA transition* (over a single memory tape), and an *NFT transition* (over multiple memory tapes—all tapes in the NFT).

Deterministic finite transducer. An n-tape *deterministic finite state transducer* (*DFST*; *deterministic finite transducer*, *DFT*) over an alphabet Γ is an NFT $\mathcal{T} = (Q, \Gamma, post, I, F)$ where $|I| = 1$ and $\forall \gamma \in \Gamma, q \in Q : |post(q, \gamma)| = 1$.

2-tape NFTs are called *nondeterministic finite input output transducers* where the first tape is the *input tape* and the second tape is the *output tape*. Unless we explicitly state the number of tapes, we will further consider 2-tape NFTs.

We will sometimes abuse the notation for transitions, especially for transition symbols, to simplify the depiction of design ideas and algorithms. We might use words, or even entire automata as transition symbols, which symbolically represents that the transition (or a single transducer tape) is replaced with a corresponding sequence of transitions containing the transitions in the word or the automaton.

Chapter 3

Automata Libraries

Uses of automata theory in computer science are ubiquitous, with applications in a large group of domains such as reasoning about reactive systems; fast and robust regex matching; reasoning about programs using dynamic memory; and modelling, analysis, or detection of vulnerabilities in software. Furthermore, improvements in the area of SMT string solving may lead to a wide area of possible applications, such as an analysis of security policies.

The success of finite automata is underlined by the vast variety in automata libraries, each providing support for a different set of automata models, and various supported operations on these models using different algorithms.

Numerous libraries aim to serve as a general-purpose automata libraries (e.g., [81, 29, 8]) while others closely specialize on just specific use cases for which the libraries highly optimize (e.g., in [44, 54]).

3.1 Mata

Mata is a new finite automata library introduced in the work [29]. Mata is built with simplicity, extensibility, and performance in mind. Mata currently provides support for DFAs and NFAs, with the set of classic operations on finite automata (testing for membership, emptiness, inclusion, or equivalence; complementation, determinization, intersection, union, minimization, and reduction), parsing finite automata to/from a textual format and parsing NFAs from regular expressions. All the operations are provided in both C++ and Python interfaces, with rich visualization options in the Python interface. Mata works with explicit alphabets, but can handle large alphabets compactly, applying techniques such as *mintermization* during preprocessing, if necessary.

Mata also implements specialized and well-optimized algorithms for simulation-based size reduction [75, 6], and antichain-based inclusion and equivalence testing [34].

Mata not only provides the user with a clear and understandable interface to common automata operations, but also allows for precise low-level access to the underlying data structures to better optimize for user- and use case-specific operations.

The basis of the performance of Mata is its three-level data structure called `Delta`, representing a transition relation of an automaton (see Section 5.1 for an explanation of `Delta`, its advantages and disadvantages, and its application to finite transducers).

The performance of Mata have been evaluated in [40] (as eNfa), and in [29]. Furthermore, Mata is used as an underlying finite automata library for a novel string solver Z3-Noodler in [17, 27, 28], where Mata handles creation and storing of finite automata and performing automata operations on them. In these papers, Mata performs well, and outperforms even the state-of-the-art automata libraries described in Section 3.2.

From these results, we can see that Mata offers an interesting set of features, namely performance and extensibility, and is applicable in numerous areas for both the research and industry use-cases.

To further extend the applicability of Mata in the fields of string solving, (abstract) regular model checking, etc., adding support for other finite automata models is planned. In this work, we add finite transducers. Other models include automata using multi-terminal binary decision diagrams (BDDAs) as in Mona for the handling of large alphabets, finite automata with registers for counting operations, or finite automata with arbitrary registers.

3.2 Other Automata Libraries

There are many automata libraries. Each library uses different data structures for storing the automata instances, and each library represents the transition relation and transition symbols differently. This significantly influences what kind of operations can be used and how they perform. Each library is written in a different programming language, usually providing the interface for that programming language alone. This gives rise to unique advantages and disadvantages for each library.

For example, automata libraries FAdo [8] and Automata.py [39] are automata libraries with a support for NFAs and DFAs. Automata.py stores transitions in a hash map mapping source states to a hash map of symbols with a set of target states. The advantage is that using maps is comfortable for the user, but the disadvantage is that hashing in general is costly.

Libraries such as Spot [36], Mosel [57], and Owl [59] have BDDs on their transitions to represent a set of symbols on a transition succinctly.

Mona [44] is another automata library for DFAs, written in C. Mona is a well-optimized automata library used, beside others, for deciding WSkS logics. It utilizes a symbolic representation of the transition relation with MTBDDs to handle large alphabets with bit vectors. Implicit determinization of automata used in Mona after every operation provides interesting advantages for certain algorithms, but might be unnecessarily expensive for other algorithms (as can be seen in Chapter 7).

BDD automata. An automaton with MTBDDs integrated into the transition relation (we call them BDDAs in this work) is an automata machine where each MTBDD represents one set of all transitions from a given source state, as applied in Mona. However, Mona implements BDDAs restricted to only deterministic automata. The terminals of MTBDDs represent the successor states of the transitions (BDDs alone use only true and false values as terminals). Therefore, MTBDDs even more compactly represent the transition relation.

Another compact symbolic representation of transition relation is used in tree automata library Vata [60]. Vata also utilizes MTBDDs for transition relations from source states. Vata implements fast antichain-based inclusion check [34] and simulation-based size reduction [75, 6] algorithms.

Symbolic representations in comparison with the explicit ones allow interesting optimizations [33], but working with symbolic automata is more complex and applying general automata algorithms for symbolic automata may prove difficult [33, 49].

A different automata models are alternating finite automata (AFA). They provide implicit operations on many automata operations, but similarly to symbolic representations, adaption of complex finite automata algorithms may prove non-trivial. AFA libraries do not seem to outperform the finite automata libraries, as shown in [40].

An implementation of weighted automata appears in Awali [63]. Awali uses vectors to store the transition relations: a vector of transitions and each state maintains a vector of indices of transitions outgoing and a vector of transitions ingoing to the state.

AutomataLib [54], written in Java, and Automata.net [81], written in C#, are popular finite automata libraries.

AutomataLib stores its transition relation as a 2D matrix of states and symbols to target states. It is well optimized for working with DFAs.

On the other hand, Automata.net uses hash maps to store the transition relation where for each state, a dynamic array of transitions is constructed. Automata.net parametrizes automata with effective Boolean algebras. It uses its predicates to annotate each transition of a symbolic NFA. Automata.net is a mature and well-optimized automata library providing a set of novel algorithms (such as optimized minimization [32]).

Brics [68], written in Java, is a finite automata library encoding the transition relation as a set of hash maps for each source state. The transition symbols are represented using symbol ranges.

This all makes it hard for the user to choose one automata library which can be used for all their work. The choice often depends on multiple factors, where not all of them can be fully satisfied. Mata tries to provide an all-around good set of features to allow for easy experimenting with new ideas, implementing prototypes of new algorithms, and when the prototype is feasible, optimizing the implementation to sufficiently replace the state-of-the-art purpose-specific automata libraries. For our use cases (mainly string solving as of now), this approach seems to work well. Moreover, Z3-Noodler already uses Mata for the finite automata. Therefore, Mata is a viable option for our intent to add support for generally usable finite transducers performant enough to be applicable in string solving in Z3-Noodler.

Chapter 4

String Solving

Despite the continuous efforts to mitigate vulnerabilities often found in web applications of today, the lists of common vulnerabilities such as CVE, CWE and OWASP TOP still list vulnerabilities related to cross-site scripting (XSS), and code injection (including SQL-injection) or execution attacks as the top critical software vulnerabilities [72, 73, 74, 30, 31]. And even though vulnerabilities in web applications cost the firms and users considerably more than many other types of software bugs, an efficient technique for fighting these attacks is still missing. Precise detection of security vulnerabilities of web applications is crucial for many players in the industry. The ability to detect security vulnerabilities before the software goes to production and during the production itself minimizes security risks and production costs of many a web technology today.

During the last two to three decades, a great development efforts have been put into improving SMT solvers for specific theories (specific constraint languages). Namely, modelling and deciding constraints over a language of strings, also known as *string constraint solving*, is one of the dominant fields of active research.

In order to prevent the aforementioned attacks, techniques utilizing string solving are often used to discover security vulnerabilities in web applications and web service APIs, as presented by works: [48, 11, 66, 42, 85, 80], and many more. Besides XSS and code injection, string solving is also successfully applied to the analysis of smart contracts [9], and analysis of user accept policies (e.g., as used in Amazon Web Services [61, 10] where there is run a billion SMT queries a day [76] to analyse cloud access policies—placing first in OWASP'21 [74] as the most critical software bug—and correct security configurations).

This greatly motivates the research of tools for string solving and the corresponding techniques, usually as an extension of the existing SMT solvers implementing a solver for the theory of strings. Using an existing SMT solver based on the DPLL(T) approach [71] has an advantage of being able to seamlessly integrate with solvers for other theories, notably the theory of integers (used for lengths of strings), or the theory of arrays (reasoning about array data structures with strings). We have seen a rapid growth of various procedures for SMT string constraint solving, implemented in numerous state-of-the-art SMT solvers.

Some of the most prominent SMT solvers with string solving capabilities include cvc5 [12], Z3 [67], or their derivations such as Trau [2], Z3str3RE [15, 14], Z3str4 [16]. To name a few more independent string solvers, a well known string solvers include OSTRICH [26], Norn [4], and others.

To further motivate the development of string solving, SMT string solvers are annually compared in a competition SMT-COMP [43] on various benchmarks both from the industry and research, solving instances of practical problems.

4.1 Noodler

There are several SMT solvers which make use of finite automata as the model for encoding the solved SMT formulae or parts of the formula, namely Z3STR3RE [15], TRAU [2], NORN [4], and others, e.g., [26]. We can see that the idea of using finite automata is not entirely new, but it is still an area which has not been thoroughly explored. Novel approaches showing great potential to solving SMT string constraints using automata models are regularly devised. Other tools, such as [37], use automata for analysis of web applications, but are limited to input filter regexes.

A new SMT string solver called Z3-NOODLER [17, 27, 28] has been recently introduced, already outperforming the well-established state-of-the-art string solvers. Z3-NOODLER is derived from Z3 [67] where it replaces its theory of string.

Z3-NOODLER brings a novel approach to solving string constraints using a method called *stabilization*, a new procedure for solving word (dis)equations in combination with regular constraints. The stabilization-based procedure seamlessly integrates with the well-used existing algorithms such as Align&Split (first implemented in a string solver NORN [4, 3], and later adopted by a multitude of other state-of-the-art string solvers—OSTRICH [62, 23, 26, 24, 25], Z3STR3RE [15, 14], SLOTH [46], . . .), used to convert regular constraints into length constraints, and Nielsen transformation [70], used for solving quadratic equations.

Thanks to this combination of approaches, Z3-NOODLER is able to uniquely utilize information from both the regular constraints (encoded as finite automata) and word equations to efficiently prune the searched state space, mitigating the effects of possible combinatorial state space explosion.

The stabilization-based procedure makes Z3-NOODLER superior to other state-of-the-art string solvers on many benchmarks, as seen in the latest results from Z3-NOODLER [28], and in previous articles [17, 27].

Z3-NOODLER, as opposed to many other state-of-the-art SMT string solvers, is based on automata techniques, utilizing finite automata to intuitively encode regular constraints. Z3-NOODLER uses MATA as the underlying automata library.

This gives Z3-NOODLER unique opportunities for efficient string solving of SMT formulae, which other SMT solvers cannot utilize. The finite automata represent regular languages of string variables in the formula and the corresponding constraints. Using finite automata allows us to solve the satisfiability of the SMT formula by performing standard automata operations. The performance results of Z3-NOODLER clearly show that, when implemented correctly, automata-based algorithms can be more performant than the classic approaches to string solving, and importantly, are often orthogonal to the techniques used in other SMT solvers which gives a great improvement when adding Z3-NOODLER into a portfolio of string solvers. This is the reason it is so important that all implemented automata models in MATA are as performant as possible.

4.2 Software Vulnerabilities in Web Applications

Many programs utilize strings as ubiquitous data structures, handling both the program data and user-supplied input data. Popular programming languages for web technologies such as JavaScript, PHP, or even Python use strings as the dominant data structure. Strings are used as keys to maps, contain names for procedures to be dynamically executed, or encode other types of information which is to be decoded later on, e.g., for serialization during communication between programs, even over the internet such as communication between data servers and user web applications. Analysis of string manipulating programs is further hindered by the lack of standardization, loose syntax or the dynamic nature of languages used in web applications: mostly scripting languages such as PHP or JavaScript.

At the same time, as shown in [79], classic software verification methods such as symbolic execution or model checking approaches are insufficient when working with strings. Using strings is highly error-prone and can cause security vulnerabilities on multiple levels in the stack: database security, server-site vulnerabilities, application level vulnerabilities such as the aforementioned vulnerabilities of unsafe handling of strings supplied by the users, with potentially malicious intent of stealing information or attacking the infrastructure.

XSS and code injection can be performed when working with strings submitted by an untrusted and unverified user without proper sanitization.

Untrusted inputs from a user should be sanitized (e.g., by escaping the input string) in order to prevent arbitrary malicious code execution. The user with a malicious intent can inject a dangerous code which is afterwards executed on a server of the web application, or locally on user's computers.

To illustrate the vulnerabilities, have a look at the following typical software vulnerabilities of software programs manipulating strings, for example [41, 58] and many more.

Cross-site scripting attack. Have a look at one textbook example of a cross-site scripting attack where text inputs are sanitized incorrectly in Listing 4.1. This code allows for insertion of malicious script executing an arbitrary code.

```
<?php
  $input_from_user = $_POST["update_account_form"];
  $body = replace(
    "/\<script.*?\>.*?\<\/script.*?\>/i",
    "",
    $input_from_user);
  update_user_account($body);
?>
```

Listing 4.1: Example of a cross-site scripting attack. An incorrectly sanitized user input can be stored directly in the database and execute an arbitrary malicious code.

The example accepts input from a user who fills in a form to update user account information on the website. The input is taken through a sanitization function `replace` which replaces a substring matched by the regular expression in the first argument with the string in the second argument in the string supplied by its last argument. The idea is that all occurrences of malicious scripts inside `<script>` and `</script>` tags are replaced with an empty string.

However, this sanitization function may be incorrect. Depending on the replacement semantics, the replacement of `.*` (Kleene star) can be performed either greedily (at which point the function would sanitize correctly) or reluctantly (the function would sanitize incorrectly). Let us see an example of cracker's input:

```
«script></script>script>perform_malicious_action("danger")</script>
```

which will be replaced with the reluctant semantics to

```
<script>perform_malicious_action("danger")</script>
```

matching only the inner pair of `<script>`, `</script>` tags.

But if we apply greedy semantics, the user input will be correctly sanitized:

```
""
```

where XSS is prevented from occuring.

A different example 4.2 adapted from [58] shows how incorrectly handled implicit browser transductions in JavaScript can allow XSS attack by executing arbitrary malicious code on the user's local browser. Implicit browser transductions code and decode HTML codes in the input string (e.g., replacing string containing a single quote with its HTML code "`&#39;`").

Having an input from a user (stored in a variable `goal`), we want to show a button with the goal specified by the user which executes the corresponding action. Ideally, `performAction(goal)` function would refuse to execute the action if the action was unspecified (therefore possibly malicious). If we pass some valid string as a user, for example `Study & Research`, the implicit browser transductions correctly handle the input, encode `&` and the button is displayed. However, if an attacker passes a string that contains an arbitrary malicious code, constructed in such a way that escaping the input will produce an executable piece of code instead of a button, the browser can perform an action specified by the attacker (here simple `alert(danger)`) on user's browser.

```
var text = htmlEscape(goal);
var action = escapeString(text);
element.innerHTML = '<button onclick=
"performAction(\'' + action + '\')">' + text + '</button>';
```

When inserted into the HTML, the button will look like the following examples. Using the correct (expected) input:

```
<button onclick="performAction('Study &amp; Research')">
  Study &amp; Research</button>
```

However, using an unexpected input with malicious intent produces the following:

```
<button onclick="performAction('&#39;);alert(danger);//')">
  &#39;);alert(1);//')</button>
```

Listing 4.2: Example of a classic cross-site scripting attack using incorrectly handled implicit browser transductions where a malicious attacker's input can be run directly in the user's local browser.

We can see that even if the code tries to sanitize the input string collected from the user, the sanitization is not performed correctly, resulting in a malicious code `alert(1)` being executed. For the sanitization to work as expected, the sanitization functions `htmlEscape` (converting HTML special characters to their entity names) and `escapeString` (escaping control characters) need to be performed in the opposite order.

Sadly, such examples do not occur only in textbooks. Real websites use sanitization functions sparingly or do not sanitize user inputs at all. This gives a large attack vector to anyone trying to target the servers of these websites with stored user data, for example. Henceforth, there is a high demand for a toolset to verify that such sanitizations are correct and safe, and implicit browser transductions are handled properly.

Consider recent works [69, 64] which perform static analysis of JavaScript code using abstract automata-based domains for symbolic execution and static analysis, but cannot reason about relations between string variables in the program: modifications of strings such as sanitization functions cannot be properly modelled and reasoned about, and reasoning about additional constraints such as integers, arrays and more in conjunction with strings becomes quickly problematic.

4.3 Replace Operations in String Solving

One of the most needed applications for finite transducers, beside others, is modelling both the input sanitization functions and the implicit browser transductions. Both applications create binary rational relations between two languages, giving a relation between strings from one language and strings in the other language. Such operations allow for replacing a substring specified by a regular pattern or a word with another string. String replace operations and the rewriting rules [55, 56] can be easily encoded as finite transducers, and resolved by applying standard finite transducer operations on the computed transducers. However, as we have seen in the examples above, choosing the correct replacement semantics is important as well. Finite transducers directly implement both the reluctant replacement and greedy replacement. Depending on the use-case, both semantics are useful.

The sanitized inputs are verified by testing for emptiness of intersection with the typical database of XSS attack patterns, represented as regular languages. SMT solvers with support for finite transducers can reason about sanitization functions and implicit browser transductions to verify the requested properties.

By adding support for finite transducers in Mata, we primarily focus on using finite transducers in Z3-Noodler to allow Z3-Noodler to solve replace operations in web applications.

The only unsupported operations in Z3-Noodler are at the time of writing string operations `str.replace_all` and `str.replace_re_all` from the theory of strings in SMT-LIB [21]. Furthermore, the string operations `str.replace` and `str.replace_re` are handled by a modified Z3-Noodler version of theory rewriter (replacing Z3 own theory rewriter) applying rewriting rules to convert both operations to a combination of word equations, disequations, and length and regular constraints. However, the generated corresponding (dis)equations and constraints negatively impact the performance of the decision procedure.

For this reason, adding support for finite transducers in Mata is paramount to the capabilities of Z3-Noodler and its overall expressiveness and performance.

Chapter 5

Finite Transducers in Mata

This chapter describes our proposed solution for representation of NFTs in MATA. We describe how MATA models the transition relation, how we model NFTs using the transition relation, what data structures we use, and how we implement the necessary algorithms for NFTs working with these data structures.

We propose utilizing the existing data structures in MATA (used for NFAs) for implementing finite transducers. The main advantage is that NFAs, NFTs, and BDDAs can all use the same single data structure, sharing a lot of the existing algorithms on the data structures, as further explained in Sections 5.3 and 5.2, which meets the simplicity requirement of MATA.

5.1 Transition Relation in Mata

MATA provides a class `Nfa` which encompasses both NFAs and DFAs and operations on them. The class `Nfa` defines which states in the represented automaton are initial (a set of initial states) and which are final (a set of final states). Furthermore, `Nfa` allows for storing of arbitrary context in the automaton itself.

The most important data structure for the representation of the finite state machines and the efficiency of the algorithms run on the representation is the representation of transition relation, called `Delta` in MATA. `Delta` defines the set of states of the finite automaton, and gives the transition relation of the finite automaton. `Delta` is designed in such a way that the often used operations such as iteration over transitions, or adding and removing transitions are performant, while the less used operations such as reversal of transition relation can be less performant.

States are represented as unsigned integers, numbered from 0. This allows adding new states to the automaton to be as easy as getting the number of states in `Delta` (constant time operation) and adding a number equal to the number of new states we want to add. Such states are immediately allocated in `Delta` and can be used in initial or final states sets.

Transition symbols are internally stored as unsigned integers as well, numbered from 0, where the last several unsigned integer values are reserved for epsilon symbols and special symbols (e.g., *don't care* symbols in the proposed NFTs implementation). MATA provides several alphabet types which map the actual transition symbols to their internal values.

The use of unsigned integers for states and symbols gives implicit ordering over both states and symbols, and querying them by accessing the index corresponding to the internal state/symbol value in an ordered vector. `Delta` is a three-level data structure where each level represents one element from the three-tuple (q, s, q') representing a single transition as:

1. source states,
2. transition symbols,
3. target states.

Each level is internally stored in memory as a sorted vector of unsigned integers, stored in a low-level data structure provided by MATA called `OrdVector`, a wrapper over `std::vector` maintaining a set of elements inserted into `std::vector` ordered.

`OrdVector` therefore has constant time access to stored elements and operations on the largest element (implemented internally as `push_back()`, `pop_back()`), fast linear iteration over the elements, and linear union, intersection, and difference. Since vectors are ordered, lookup for states and symbols is logarithmic using binary search. Insertion and removal of elements are logarithmic, but they must shift elements in the memory in the underlying `std::vector` which slows the operations down.

Due to this, MATA tries to iterate over `OrdVector` as often as possible since internally, `std::vector` stores elements in a continuous array on the heap with good memory locality, and add elements one after the other in ascending order given by the value of the inserted elements at the end of the ordered vector. MATA implements several algorithms for ease of use of the iteration, such as synchronized traversal over multiple ordered vectors. General insertion and removal of transitions are logarithmic, but MATA algorithms are written in such a way that the general insertion and removal are seldom used.

This pairs well with underlying data structures for sets of initial and final states, implemented as sparse sets [19] allowing for constant element lookup, insertion and removal, and fast linear iteration through elements.

The Figure 5.1 visualizes the three-level `Delta` data structure, as presented in [29]. We can see that when we access by source state q the index in the first vector of source states, we get a vector post of type `StatePost` (`StatePost` [q]), representing $post(q)$, which is a vector of symbol posts for each transition symbol a leading from source state q, of type `SymbolPost`. Each symbol post for symbol a represents $post(q, a)$, storing the transition symbol a and a set of target states represented as an `OrdVector`.

Notice that since MATA supports epsilon symbols as maximal unsigned integer values, the symbol posts for epsilons are always at the end of state posts. Epsilon state posts can be therefore accessed in constant time instead of having to search the whole state post to look them up. MATA operations take advantage of this in numerous operations using epsilon symbols, such as in string solving [17].

Example 1. Have a look at the example in Figure 5.2. We perform an intersection of two NFAs (their transition relations at the top) using product construction, as described in Chapter 2, constructing a product transition relation (at the bottom). MATA takes advantage of its `Delta` with ordered states and symbols.

During the construction, new product transitions are often appended at the end of the existing vectors (algorithms try to minimize the number of binary searches in the lower levels in `Delta`). In this example, computation of the new transitions for product state $(1, 3)$ creates 4 new transitions, introducing new product states, e.g., $(0, 1)$ (product state 3), and $(0, 3)$ (product state 4).

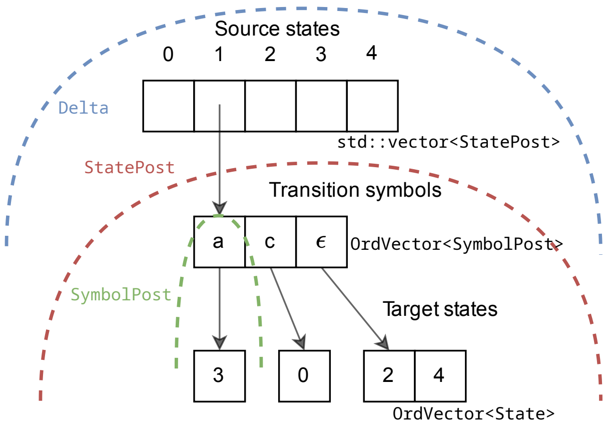


Figure 5.1: The visualization of the three-level data structure representing the transition relation of a finite automaton in MATA, called `Delta`.

We can see that we first insert into a vector of target states for $post(2, c)$ in the product the product state 3, and only then create a new product state 4 and append it after the product state 3.

5.2 Transducers as Level Automata

In this section, we explain how we encode NFTs to reuse data structures for NFAs.

Example 2. In Figure 5.3, we show a 2-tape input/output NFT $\mathcal{T}$ accepting a rational relation with an input language of sequences of abc, and the corresponding output language where each input transition symbol in a sequence is replaced accoringly as: input transition symbol a is replaced with bc, b is erased, and c is replaced with a. That is, for language $\Sigma = \{a, b, c\}$, $\mathcal{T}$ accepting language $(abc)^*$ on the input tape, the following transductions are performed on each sequence abc: $a \to bc$, $b \to \epsilon$, and $c \to a$.

For example, the input word $abcabc$ is accepted on the input tape and produces a word $bcabca$ on the output tape.

A *level automaton* is an NFA extended with a function $\texttt{lev} : Q \to \mathbf{N}$ annotating states with levels.

An n-tape NFT $\mathcal{T}$ accepts a relation $\mathcal{L}(\mathcal{T}) = \{\bar{w} \in (\Sigma^*)^n \mid \bar{w} = \gamma_1 \circ \gamma_2 \circ \ldots \circ \gamma_m = (a_1^1, a_1^2, \ldots, a_1^n) \circ (a_2^1, a_2^2, \ldots, a_2^n) \circ \ldots \circ (a_m^1, a_m^2, \ldots, a_m^n)\}$. We represent the same automaton as a level automaton $\mathcal{A}$ over the alphabet Σ by unfolding $\mathcal{T}$ transitions $(q, (a_1, a_2, \ldots, a_n), q')$ into a sequence of NFA transitions $(q, a_1, q_1), (q_1, a_2, q_2) \ldots, (q_{n-1}, a_n, q')$. $\texttt{lev}(q) = \texttt{lev}(q') = 0$, $\forall i \in \{1, \ldots n-1\} : \texttt{lev}(q_i) = i$. The level automaton corresponding to $\mathcal{T}$ accepts language $\mathcal{L}(\mathcal{A}) = \{\lambda \in \Sigma^* \mid \lambda = a_1^1 \cdot a_1^2 \cdot \ldots \cdot a_1^n \cdot a_2^1 \cdot a_2^2 \cdot \ldots \cdot a_2^n \cdot \ldots a_m^1 \cdot a_m^2 \cdot \ldots \cdot a_m^n\}$. That is, NFT symbol $\gamma = (a_1, a_2, \ldots, a_n) \in \Gamma$ is perceived as a word (a sequence of symbols) $a_1 \cdot a_2 \cdot \ldots \cdot a_n \in \Sigma^*$. The symbol γ is not read by the level automaton as a single symbol, but it is read sequentially, one tape per transition. A similar approach to this simplest representation with unfolding has been used in LASH [84], a tool for reasoning about arithmetic with an integrated automata library.

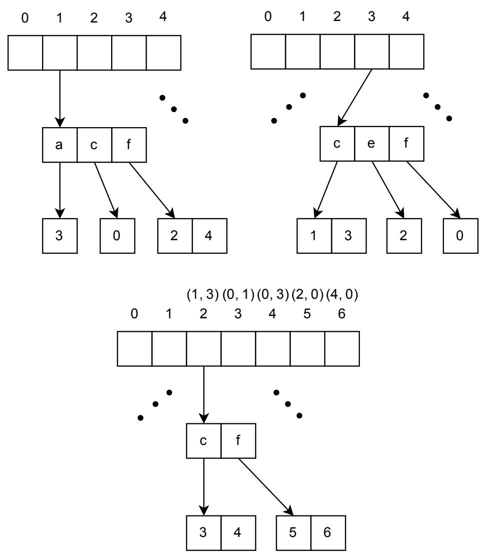


Figure 5.2: The visualization of intersection product construction on NFAs utilizing the advantages of the three-level transition relation `Delta`.

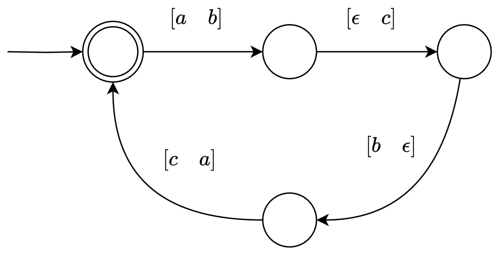


Figure 5.3: Example of a 2-tape input/output NFT.

Example 3. The NFT from Example 2 is represented as a level automaton in the Figure 5.4.

The transitions from states with level 0 represent the input tape, and the transitions from states with level 1 represent the output tape.

Intuitively, we can still see that the same rewriting rules apply. The input word *abcabc* is still accepted on the input tape and produces the same word *bcabca* on the output tape. Notice that we need epsilon symbols to encode the rewriting rules applied here. Efficient handling of epsilon transitions in level automata is a complicated problem. We further discuss this in Section 5.4.

5.3 Class Hierarchy

We propose that `Nfa` is a base class from which NFTs, and future BDDAs and register automata inherit, including all operations on `Nfa` where only a few specific operations need to be modified. The Figure 5.5 shows the proposed class inheritance hierarchy. This hierarchy also allows us to add model-specific operations to each automata model without modifying operations for the other automata. BDDAs in our representation only extend

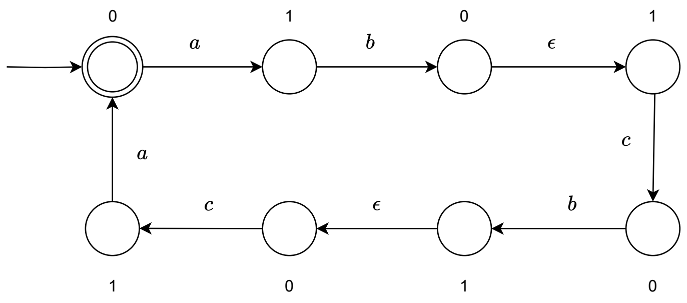


Figure 5.4: 2-tape NFT from Example 2 represented as a level automaton with states annotated by levels.

the functionality of NFTs, and can therefore inherit most of the algorithms directly from NFTs and only modify some algorithms specific to BDDA use cases.

We want to reuse `Delta` for the representation of transition relation in finite transducers.

NFTs and BDDAs can directly utilize `Nfa` with all the underlying data structures. The only modification of `Nfa` to support NFTs and BDDAs is adding a vector of state *levels* (represented by `std::vector`<unsigned>) indexed by the automaton states. The vector maps each automaton state to its level in the finite state machine. n-tape NFT has n levels. Therefore, in a representation of an n-tape NFT, the vector of state levels maps to values from 0 to $n-1$. One transition in NFT corresponds to n transitions in `Nfa`.

Each transition in `Nfa` corresponds to one level of the NFT, i.e., one tape of the NFT. NFT transition always starts at level 0. A transition from level 0 to level 1 represents the operation on the first NFT tape, from level 1 to level 2 the operation on the second NFT tape, and so on. Finally, the transition from state with level $n-1$ back to state with level 0 corresponds to the last tape in the NFT, finishing the sequence of NFA transitions for a single NFT transition.

For this reason, NFTs can have only states q with $\texttt{lev}(q) = 0$ as initial or final states. This is crucial for many algorithms (more in 5.5) which rely on certain invariants in the representation of NFTs, such as that NFTs always synchronize in states with state level 0 and each step in operations must consider the entire state levels sequence from 0 to $n-1$ as a single *abstract* step (a single complete NFT transition).

For the 2-tape (input/output) NFTs, the levels used will be 0 and 1, where transitions from states with levels 0 represent the input transitions and transitions from states with levels 1 represent the output transitions.

Since levels directly represent their corresponding tapes, we use the terms *level* and *tape* interchangeably when the meaning is clear.

5.4 Epsilon and Don't Care Transitions

The approach used in LASH does not support epsilon and don't care transitions. However, finite transducers need support for epsilon transitions: epsilon symbols are often used to, e.g., model a removal of a malicious substring or model sanitization function `HTMLEscape` in web applications.

Notice that BDDAs are similar to the level automata, only slightly modified. BDDAs do not have epsilon symbols, but need don't care symbols (matching any transition symbol) with *jumps*. An interesting and tricky problem is how to add support for epsilons and

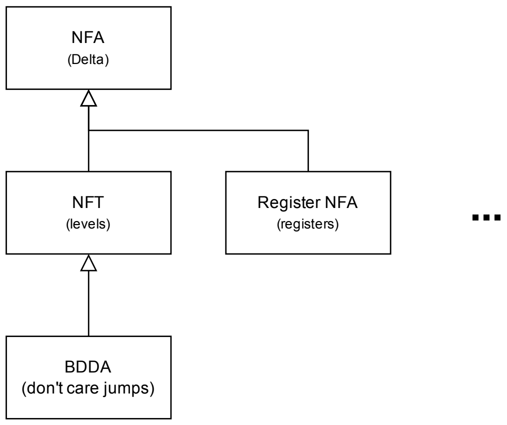


Figure 5.5: A class inheritance hierarchy for NFAs, NFTs, and future BDDAs and register automata. All of these finite state machines utilize the same data structure, with mostly the same algorithms, where only a few algorithms must be modified for each respective type of finite state machines. Each type of finite state machines can further extend the set of operations by their own specific operations. Since BDDAs only extend the functionality of transducers, BDDAs can inherit directly from NFTs, modifying a few algorithms of NFTs together with the base `Nfa` algorithms.

don't care symbols to level automata elegantly to allow reusing the existing finite automata algorithms without large modifications to the algorithms and without complicating the data structures for NFAs used to represent level automata. We pay close attention to the careful design of epsilon transitions and don't care jumps which work well with NFAs and their algorithms.

Mata already supports epsilon symbols ϵ as (several) last transition symbol(s) in the range of possible transition symbols given by the data type for the symbol. Similarly, we reserve a new symbol at the end of the range for the don't care symbol ?.

An epsilon symbol on a transition from state with level k to level $k+1 \mod n$ in `Nfa` representing an NFT means that the tape corresponding to the level k does not read from / write on the tape k any symbol.

We also allow jumps where the transition goes from a state with level k to any other state with level l where $l \neq k+1$. However, we restrict the jumps for n-tape NFT as follows:

- The jump cannot jump over a state with level 0. That is, if we need to jump further, the transition can at most jump to the next state with level 0.
- A jump of length $k > 1$ (the length of one is a normal transition in `Nfa` which we call just a transition) is interpreted the same as if we made k transitions in `Nfa` with the transition symbol on the jump transition. That is, a jump of length 3 (over 3 tapes) with transition symbol ϵ means that the transducer made 3 normal transitions with ϵ as a transition symbol for each of them.
- Jumping from a state q where $\texttt{lev}(q) = 0$ to a next state q' where $\texttt{lev}(q') = 0$ with transition symbol ϵ means that only the internal state of the NFT has changed (the state has changed) without reading or writing anything on any of the tapes.

- Jumping from a state q where $\texttt{lev}(q) = 0$ to a next state q' where $\texttt{lev}(q') = 0$ with transition symbol $a \in \Sigma$ means that we performed an identity NFT transition over a where all tapes read/write symbol a.

NFTs and BDDAs interpret the same instance of an NFT automaton slightly differently. BDDAs change the interpretation of a jump to a single transition symbol (from the jump transition) followed by a sequence of don't care symbols for the remaining tapes (levels), as opposed to the NFT interpretation where all transitions in the jump use the jump symbol. We easily implemented this BDDA interpretation next to the NFT interpretation.

No-operation transitions. An epsilon transition from a state with level k to $k + 1 \mod n$ can be understood as a *no-operation transition* `nop`, i.e., a transition where the tape k does not perform any operation (no reading nor writing). In essence, the impact of `nop` is the same as if reading an empty string ϵ on the tape k. An epsilon jump from a state with level 0 to another state with level 0 is a simple change of states (as per the usual definition of epsilon transitions in NFAs). It has the same effect as a sequence of `nop` transitions on all tapes (or a sequence of ϵ symbols).

Depending on the application, one might prefer to use ϵ symbols or `nop` transitions as needed.

5.5 Algorithms

For general uses of transducers in Mata, we have implemented general automata operations for creating, modifying, and manipulating transducers. Furthermore, we have implemented several transducer-specific operations.

NFA operations. Thanks to our design choices, the general NFA operations such as concatenation, union, etc. can be reused from NFA without many modifications. We only need to remember to correctly set levels of states when renumbering states in the result NFT.

Further, we discuss several more significant modifications to existing operations and algorithms for several NFT-specific operations.

5.5.1 Synchronization

All operations performing some kind of traversal over multiple transition relations (or multiple transitions in one transition relation simultaneously) start the computation from states with level 0. NFTs are always synchronized on states with level 0, that is, after each NFT transition is completed (all tape operations are handled). If the macrostate in a worklist contains only states with levels 0, the NFTs are synchronized and the next macrostate can be computed. When the synchronized NFTs perform one transition, due to the supported jumps, the new macrostate may contain states with different levels.

Since jumps can jump at most to the next state with level 0, we always know which states in the macrostate are "ahead" and which are "behind" by the following method. If we expand a macrostate with states with levels k and l where $k > l$, we know that the state with level k must be ahead of the state with level l. If any of the states has level 0 (and there are other states with non-zero levels), we know that the state 0 must be ahead of all the other states with non-zero levels, and synchronized with all other states with levels

0. That holds since the states with levels 0 can only be the next states with levels 0, and not the states with level 0 behind (as the first step from the synchronized states is to move immediately out of the states with level 0). The same holds for BDDAs using the same constraints as NFTs. Only the state in the macrostate with the level furthest behind is expanded (performs its transitions, if possible). The others are at least one level ahead and therefore cannot be expanded and must wait for the states behind to get to at least their levels first before being expanded themselves.

Example 4. If we perform synchronization over three 4-tape NFTs, starting from macrostate with levels $(0, 0, 0)$, we perform one transition and may end up in a macrostate with levels $(3, 1, 0)$ where the first NFT made a jump of length 3, the second performed normal transition (jump of length 0), and the last NFT performed a jump to the next state with level 0. When expanding the macrostate $(3, 1, 0)$ later on, we know that the state with level 0 is ahead of the other states with levels 3 and 1. And we further know that state with level 3 must be ahead of the state with level 1. Therefore, the state with level 1 must be expanded up until the level reaches 3 or greater (the next level 0). Then we may get $(3, 3, 0)$, both the first and the second state are expanded to states with levels $(0, 0, 0)$ and the macrostate is synchronized again.

The Figure 5.6 shows an example of how may synchronization on a pair of NFTs using jumps look.

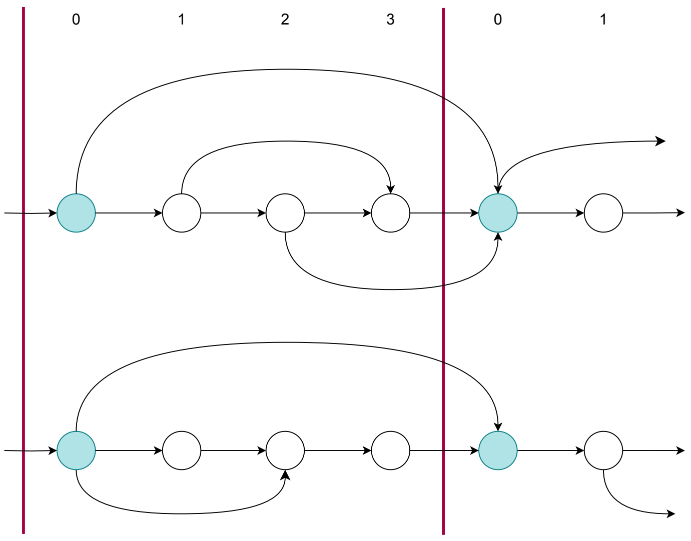


Figure 5.6: An example of 4-tape NFTs with jumps during synchronization. Jumps can start in state with any level and jump up to the next state with level 0. The jumps cannot jump over states with level 0, however. The numbers denote the levels for the corresponding states below. The highlighted states depict the states with level 0, the vertical lines visually set apart NFT transitions (there is one NFT transition, consisting of several tapes, between each pair of vertical lines).

Alternative representation of jumps. If the restriction of allowed jumps being only to the next state with level 0 is too strict, an alternative approach where jumps are allowed to jump over (potentially multiple) states with level 0 can be chosen. However, during operations such as product construction, additional information must be added to the worklist containing macrostates: a list of lengths of each jump performed during the expansion of the last macrostate in order to know how far ahead are the states jumped to in relation to the other states in the macrostate.

We deem this approach inferior to our chosen approach since it distorts the simplicity of a single data structure and algorithms on it for NFAs, NFTs, and BDDAs.

5.5.2 Projection

The first standard operation on transducers is *projection.* Given an n-tape NFT $\mathcal{T}$, we can perform *projection to* a specified set of tapes, or *project out* a specified set of tapes.

The transitions remain the same, the only change is that we remove certain tapes (meaning that for every NFT transition we remove transition symbols on those tapes). In Mata, this means removing all states with certain levels k and their outgoing transitions and redirecting ingoing transitions to these states to the next states with levels $k+1 \mod n$ where the outgoing transitions were previously leading to. This can be iteratively performed on all tapes specified for removal. When the modification is performed, all remaining levels are shifted toward 0 to start at 0 and end with $l-1$ where l is the new number of tapes remaining in the NFT.

Projection to, denoted $\texttt{project\_to}_M(\mathcal{T})$ where $M \subseteq \{0,1,\ldots,n-1\}$ is an operation which keeps only tapes $t \in M$, and removes from $\mathcal{T}$ all other tapes $o \notin M$. In practice, this means that an n-tape NFT will be modified into an $|M|$-tape NFT. If $|M| = 1$, we create a 1-tape NFT (containing only states with level 0) which is semantically equivalent to a corresponding NFA with the same transitions.

Projection out, denoted $\texttt{project\_out}_M(\mathcal{T})$, performs the same operation, only now instead of keeping the specified tapes $o \in M$, we remove all tapes $o \in M$, and keep only the tapes $t \in \{0,1,\ldots,n-1\} \setminus M$. If $|M|+1 = n$, we project out all tapes except for one. This creates a 1-tape NFT, which is again semantically equivalent to a corresponding NFA with the same transitions.

Note that for either variation, removing the first tape in NFT transitions requires setting new initial states (the original targets of transitions from the states with level 0 become the new initial states). Similarly, removing the last tape in NFT transitions requires setting new final states (the original source of ingoing transitions to the final states become the new final states). If jump transitions are involved, one may need to expand the jumps into normal transitions between levels to set the correct initial and final states on the new first and last tape.

Example 5. Given an NFT $\mathcal{T}$ with 5 tapes, with levels $\{0,1,2,3,4\}$.

We can perform the following operations:

- $\texttt{project\_out}_{\{3\}}(\mathcal{T})$ which deletes tape 3 from $\mathcal{T}$, leaving tapes $\{0,1,2,4\}$ which are then renumbered to $\{0,1,2,3\}$.
- On the result, we can perform $\texttt{project\_to}_{\{1,3\}}(\mathcal{T})$ which performs the same as if calling $\texttt{project\_out}_{\{0,2\}}(\mathcal{T})$. Tapes 0 and 2 are removed, leaving tapes $\{1,3\}$, which are then renumbered to $\{0,1\}$.
- Now, performing $\texttt{project\_to}_{\{1\}}(\mathcal{T})$ gives us NFT with only one tape, which represents the NFA for the tape 1. Performing $\texttt{project\_out}_{\{1\}}(\mathcal{T})$ instead would give us NFT with only one tape representing the NFA for the tape 0.

5.5.3 Intersection

For constructing the intersection of two NFTs, we make use of the inheritance of NFTs from NFAs and utilize the classic intersection computation on NFA using the standard product construction algorithm (as described in Chapter 2). The only modification is adding support for proper assignment of levels to new macrostates depending on the levels of the original NFT states in the macrostate, and handling of various combinations of transition symbols during the process of adding transitions leading from the macrostates.

Given $\mathcal{T}_1$ with Q_1 and $\mathcal{T}_2$ with Q_2, construct the intersection $\mathcal{T} = \mathcal{T}_1 \cap \mathcal{T}_2$.

A new level $\texttt{lev}(q')$ for a new macrostate $q' = (q'_1, q'_2)$, given a current macrostate $q = (q_1, q_2) \in Q_1 \times Q_2$ where q'_1 and q'_2 are targets of transitions from q_1 and q_2, is determined as follows:

- If $\texttt{lev}(q'_1) = 0$ or $\texttt{lev}(q'_2) = 0$ (holds for both equal 0, too), $\texttt{lev}(q') = \texttt{max}(\texttt{lev}(q'_1), \texttt{lev}(q'_2))$. If both target states have levels 0, we are finishing an entire NFT transition and enter a synchronized macrostate q', $\texttt{lev}(q') = 0$. Otherwise, only $q'_1 = 0$ ($q'_2 = 0$) holds, and q'_1 (q'_2) must be "further ahead" of q'_2 (q'_1), and must wait for q'_2 (q'_1) to catch up, therefore, $0 < \texttt{lev}(q') < n$.

- Otherwise, $\texttt{lev}(q') = \texttt{min}(\texttt{lev}(q'_1), \texttt{lev}(q'_2))$. That is, if both $0 < \texttt{lev}(q'_1), \texttt{lev}(q'_2) < n$, we must wait for the state further behind to catch up, so we choose the lower level as $\texttt{lev}(q')$.

A new transition symbol $a \in \Sigma$ for a new transition (q, a, q') is chosen by the following rules:

- If $(q_1, a, q'_1) \in post_1$ and $(q_2, a, q'_2) \in post_2$, $(q, a, q') \in post$.

- If $(q_1, \epsilon, q'_1) \in post_1$ and $(q_2, \epsilon, q'_2) \in post_2$, $(q, \epsilon, q') \in post$. Epsilon symbols are handled here as normal symbols $a \in \Sigma$. This ensures that only matching ϵ symbols are synchronized.

- If $(q_1, ?, q'_1) \in post_1$ (or $(q_2, ?, q'_2) \in post_2$), a from (q_2, a, q'_2) ((q_1, a, q'_1)) is used. This holds for $(q_1, ?, q'_1) \in post_1$ and $(q_2, ?, q'_2) \in post_2$, too. This works if one of the symbols is ? and the other is ϵ, too: (q, ϵ, q') is produced in such a case.

The rules are visualized in the Table 5.1.

$\cap$	?	ϵ	a	b
?	?	ϵ	a	b
ϵ	ϵ	ϵ		
a	a		a	
b	b			b

Table 5.1: Synchronization rules for transitions symbols during intersection. The row lists the symbols on transitions in $\mathcal{T}_1$, the column in $\mathcal{T}_2$.

Finally, ? transitions must be handled separately since the classic intersection algorithm for NFAs does not match ? with any other symbol than ?.

5.5.4 Composition

When working with NFTs as transduction machines taking some input, modifying it and outputting some output, we often encounter problems where we want to perform multiple transductions on the input, one after the other. In the same way as we can compose mathematical functions $(f \circ g)(x)$ meaning $f(g(x))$ (perform g on x and perform f on the result of $g(x)$), we can perform multiple NFT transductions sequentially: $\mathcal{T}_2 \circ \mathcal{T}_1$ (perform the transduction of $\mathcal{T}_1$ on the input, and then perform the transduction of $\mathcal{T}_2$ on the result), denoted as `compose`$(\mathcal{T}_1, \mathcal{T}_2)$. Using the Unix *pipe* notation, we can express the same with $\mathcal{T}_1 || \mathcal{T}_2$ (perform the transduction of $\mathcal{T}_1$ on the input, and then pipe the result to $\mathcal{T}_2$ to perform the transduction of $\mathcal{T}_2$ on the result).

For each NFT, we need to specify which tape(s) T to synchronize on with the other NFT. These are the tapes we expect to act as a mediator between the output of $\mathcal{T}_1$ and the input of $\mathcal{T}_2$. To create a new product transition, the symbols on the corresponding tapes in both NFTs must be compatible (the same symbol, or pairs like $(?, a)$ for $a \in \Sigma$, etc.).

The composition comprises the following steps:

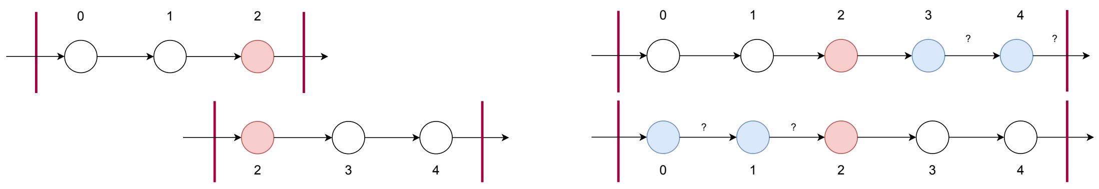
- Extending the NFTs to contain matching tapes,
- Adding special self-loops on states with level 0 to allow for proper handling of ϵ and ? symbols on the transitions,
- Performing NFT intersection as described in Section 5.5.3, and
- Projecting out the synchronization tapes (as described in Section 5.5.2) which were "consumed" by the composition.

Example 6. Throughout this section, we will demonstrate the composition algorithm on the following example. Given NFT $\mathcal{T}_1$ with tapes 0, 1, and 2, and $\mathcal{T}_2$ with tapes 2, 3, and 4, we want to compute $\mathcal{T}_1 || \mathcal{T}_2$, synchronizing both NFTs on tape 2. For clarity, we number the tapes uniquely between both NFTs in this example. Internally, each NFT has tapes with levels from 0 to 2 and for each NFT, we specify which tape(s) the NFTs should synchronize on.

Extending NFTs with additional tapes. The first step of the composition is to extend the input NFTs by inserting additional levels k (additional states with levels k) into each transition with ? symbols on the transitions at levels k. This operation balances out the NFTs so both have the same tapes in the same order. This allows us to perform the classic NFA intersection for each level in both NFTs during NFT intersection. We use ? symbols since for each NFT $\mathcal{T}_i$, the other NFT $\mathcal{T}_j$ does not care what transitions over what symbols $\mathcal{T}_i$ performs during the intersection. Hence, from the point of view $\mathcal{T}_j$, $\mathcal{T}_i$ can perform any transition.

Example 7. Continuing the example, we extend each NFT transition in $\mathcal{T}_1$ with new transitions as follows: from existing states q_2 with level 2 to new states q_3 with levels 3 (using the original transition symbol on transitions outgoing from q_2), from q_3 to new states q_4 with levels 4 using ? symbol, and finally from q_4 to the next existing states q_0 with levels 0 (the original targets of transitions from q_2) using ? symbol (finishing the original NFT transition).

Figure 5.7 shows a depiction of a single transition in $\mathcal{T}_1$ and $\mathcal{T}_2$ before and after extension. We can see that after the extension, both $\mathcal{T}_1$ and $\mathcal{T}_2$ have the same tapes.



(a) Before extension (b) After extention

Figure 5.7: A depiction of a single transition in NFTs $\mathcal{T}_1$ (top) and $\mathcal{T}_2$ (bottom) before extension (Figure 5.7a) and after extension with ? transitions on inserted tapes to balance out the tapes in both NFTs (Figure 5.7b). The red states together with the outgoing transitions represent the tape $\mathcal{T}_1$ and $\mathcal{T}_2$ are synchornized on. The blue states represent the newly inserted tapes with ? symbols on transitions.

Thanks to this general approach, we can compute compositions of any n-tape and m-tape NFTs, synchronizing on any tape(s)[1], with the tapes being in an arbitrary order.

As an optimization, when adding multiple subsequent ? transitions during the extension, we can instead add only a single ? jump from the first state in the sequence leading to the last state in the sequence, without having to create new states for all inserted tapes being jumped over.

Adding self-loops. The next step is to add self-loops to all states with level 0 in both NFTs. These self-loops serve the purpose of nondeterministic "waiting" loops for waiting at the beginning of an NFT transition in one NFT for the other NFT for epsilon transitions on the tapes being synchronized on.

The self-loops is a symbolic name since they are self-loops in the sense of NFT transitions (from state with level 0 back to the same state), but they lead over several tapes (i.e., several states with transitions).

The symbols on the self-loops are added as follows:

- For the transitions on the synchronization tapes, the symbol ϵ is used, being handled as a normal transition symbol the other NFT can synchronize with the ϵ symbol on any non-self-loop transition.
- For the transitions on tapes added by the extension, the symbol ? is used. ? here has the same role as ? symbols added during extension.
- For the transitions on the remaining tapes (the original non-synchronizing tapes), ϵ symbol is used to represent the waiting on all NFT's own transitions.

Example 8. Figure 5.8 shows how a single transition in NFTs looks after adding the "waiting" self-loops.

Intersection. When NFTs are balanced out and self-loops are added, we can perform an NFT intersection, as per Section 5.5.3. The intersection produces a new NFT representing the composition of $\mathcal{T} = \mathcal{T}_1 || \mathcal{T}2$, only currently $\mathcal{T}$ still contains all the tapes from the balanced $\mathcal{T}_1$ and $\mathcal{T}_2$. Other than that, the composition is complete.

[1]When synchronizing on multiple tapes, the tapes must be in the same order in both NFTs.

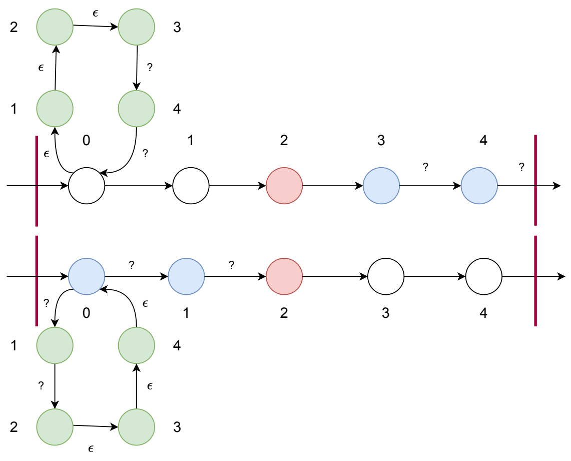


Figure 5.8: A depiction of a single transition in NFTs $\mathcal{T}_1$ (top) and $\mathcal{T}_2$ (bottom) after adding the "waiting" self-loops.

Note that since we added the self-loops, we no longer need to separately handle ϵ transitions. ϵ symbols are handled as normal transitions, producing an ϵ transition in the product iff both NFTs perform an ϵ transition, or one of them performs a ? transition.

Furthermore, we prevent self-loops in both NFTs from synchronizing with each other. That is, we cannot construct a new macrostate $q' = (q_{l1}, q_{l2})$ where both q_{l1} and q_{l2} are self-loop states. This would represent a situation where both NFTs are waiting for the other, but none moves forward. Since these macrostates are not useful, we prevent their creation entirely.

Projecting out the tapes the composition synchronized on. To clean up $\mathcal{T}$, we need to project out the tapes we synchronized on, as these are the tapes that have been "consumed" by the composition (used as the link between the output of $\mathcal{T}_1$ and input of $\mathcal{T}_2$).

Example 9. To finish our example, Figure 5.9 depicts a single product transition after composition is performed, and the synchronization tapes are projected out.

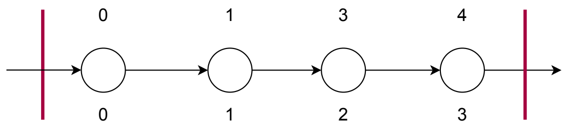


Figure 5.9: A depiction of a single product transition in $\mathcal{T} = \mathcal{T}_1 || \mathcal{T}_2$ synchronized on tape 2. The levels at the top represent the original tapes before renumbering. The levels at the bottom represented the final tapes after renumbering.

Optimization for 2-tape NFTs. For working with only 2-tape NFTs, a specialized algorithm for composition can be utilized. We presume that input/output NFTs always input on the first tape and output on the second tape. The algorithm would still utilize the general idea of NFT intersection, but now no extension of NFTs and no self-loops are necessary. For each macrostate, we would first perform a step in $\mathcal{T}_1$ on level 0, then perform the synchronized step on levels 1 in $\mathcal{T}_1$ and 0 in $\mathcal{T}_2$ (if such a step is possible as per NFT intersection rules), then finish the composition of the current transition by performing a step in $\mathcal{T}_2$ on level 1, constructing a new macrostate from targets of transitions on levels 1 of both NFTs. Handling of jumps is similar to NFT intersection: waiting for the state "left behind" to catch up with the state jumped to. This approach no longer handles ϵ symbols as normal symbols $a \in \Sigma$, but instead uses the classic synchronization on ϵ symbols used in the intersection for NFAs.

When adding support for transducers to Z3-Noodler, it may be advantageous to utilize this specialized variation of composition as string replace operations in Z3-Noodler only ever work with 2 tapes.

5.5.5 Application

When we construct an NFT $\mathcal{T}$, the most intuitive use of $\mathcal{T}$ is to perform the transduction $\mathcal{T}$ encodes (reading the input on the input tapes and outputting the output on the output tapes). This operation is called *application*.

We can *apply* $\mathcal{T}$ on a word w, or on a regular language $\mathcal{L}(\mathcal{A})$ for some $\mathcal{A}$.

Application on a word. We can either apply $\mathcal{T}$ on a word $w \in \Sigma^*$ on a given tape (at level k), or apply $\mathcal{T}$ on an l-tuple of words (w_i) where $l < n$, $i \in \{0, \ldots, n-1\}$ and i specifies on which tape (level) we want to apply the respective words on (for tape i, one word w_i).

The result of the application on a word (or on a tuple of words) is a set of tuples of size $(n - l)$ of words. We read the word(s) and move through the NFT according to the transition symbols in the word(s) on the corresponding tapes, similarly to performing a run on a word in NFA, only now to potentially multiple tapes (multiple words). During the traversal, we construct the result by appending the transition symbols on the remaining tapes to the result constructed so far for the current state. This gives us the corresponding word on the remaining tapes when the words on the given tapes are (w_i).

Focusing on 2-tape NFTs $\mathcal{T}$, the application equals to translation from the input word w to the output word w' (similarly to what a natural language translator does) where $w' = \texttt{apply}(\mathcal{T}, w)$. By convention, we presume the input tape is at level 0 and the output tape is at level 1.

The application on a word is equivalent to $\mathcal{T}_{\mathrm{Id}(\mathcal{A})} || \mathcal{T}$ where $\mathcal{A}$ represents DFA accepting only w. This gives us an NFT encoding the result. For 2-tape $\mathcal{T}$, we get a 1-tape NFT, equal to the corresponding NFA encoding the set of words in the result.

Application on a language. Application of n-tape NFT $\mathcal{T}$ on a regular language $\mathcal{L}(\mathcal{A})$ instead of a single word is performed similarly to the application on a single word, but now the result is a regular language (for 2-tape NFTs) or an NFT $\mathcal{T}'$ with $n - 1$ tapes.

The application on a language $\mathcal{L}(\mathcal{A})$, similarly to the application on a word, is equivalent to $\mathcal{T}_{\mathrm{Id}(\mathcal{A})} || \mathcal{T}$. The resulting NFT encodes the result. For 2-tape $\mathcal{T}$, we get a 1-tape NFT, equal to the corresponding NFA encoding the regular language of the result.

5.6 Implementation

The proposed data structures for finite transducers, and standard operations on finite transducers were implemented in MATA. The code is accessible in a GitHub repository[2] and will be merged into the main MATA GitHub repository[3].

The NFA operations for NFTs have to be modified in places where we need to correctly set and reset the vector of levels for transducer states, or where we handle epsilon, `nop` and don't care symbols in automata algorithms.

Number of levels. Each NFT also stores the number of tapes (levels) used. If the number of tapes is set to 1, it means that the NFT is equivalent to NFA and the NFT operations will behave the same as a corresponding NFA operations (with an equivalent transition relation where all don't care transitions are replaced with transitions between the same states over the whole alphabet). This keeps the behaviour of operations on NFTs and NFAs consistent between automata models.

Single epsilon for epsilon and `nop` transitions. We have decided to unify the symbols for epsilon transitions and `nop` transitions. We represent both transitions with a transition over ϵ, corresponding to the largest unsigned integer value of a symbol in MATA.

To distinguish between both, we check whether the epsilon transition leads from a state with level k to a state with level $k + 1 \mod n$ or not whenever we encounter the symbol in algorithms. This operation has constant complexity, since a set of levels in MATA is a state-indexable vector of levels.

We decided for this approach since all operations in MATA already support epsilon transitions, checking for existence of epsilon transitions and accessing them has constant complexity, and adding handling for yet another special transition symbol further complicates the logic for many operations which do not have to distinguish between epsilon and no-operation transitions.

Specifying tapes to work on. Since we aim to implement NFTs in MATA so they are usable for general purposes, not only limited to string solving, we allow for the majority of operations to specify what is the interpretation of each tape: e.g., which tapes are to be the input tapes, and which are to be the output tapes in application on a word or a language; or which tape(s) to use for synchronization during composition.

Be mindful that using different tapes for some operations may affect the performance of these operations. For example, application on the first tape is more performant than the application on the last tape in an n-tape NFT.

Basis of the implementation. For the implementation of the current interface and algorithms specific for NFTs (composition and projection), we used the existing implementation for a simple prototype BDDA representation using NFAs with levels (supporting simple composition and projection operations), which was implemented previously in MATA. we further modified the interface for the existing operations, expanded the interface with new operations, added string solving-specific operations (construction of typical NFTs used in string solving, NFTs for replace operations, ...). We collaborated with our colleagues

[2]`https://github.com/Adda0/mata/tree/transducers`

[3]`https://github.com/VeriFIT/mata/`

needing BDDAs to define the same interface for NFTs and future proper implementation of BDDAs, inherited from NFTs. As a result of discussions and collaboration, we add support for NFTs (this work, extending the existing implementations for projection and composition) and we plan to add support for BDDAs in the future: using the defined common interface, where BDDAs inherit operations from NFTs where the cores of the NFT- and BDDA-specific operations from Section 5.5 (composition, projection) are modified and extended versions of the existing implementations for simple BDDA implementation. Thanks to this, we are able to add official support for BDDAs into Mata easily. The proper support for BDDAs inheriting from NFTs is planned to be implemented when the current implementation for NFTs matures enough.

Chapter 6

Transducers for String Solving

We discuss in this chapter how to utilize finite transducers in Z3-Noodler to solve SMT string constraints `str.replace`, `str.replace_re`, `str.replace_all` and `str.replace_re_all`.

6.1 Replace Operations

A general regular replacement operation has the following form:

```
replaced_string = replace(input_string, pattern, replacement)
```

where `replace` determines the type of the regular replacement (the semantics of the replacement operation). `input_string` represents the original input string where we want to replace some regular `pattern` with a `replacement` literal. Regular `pattern` can be a string literal (a word) or a regular language describing the substring (or substrings) of `input_string` we are trying to replace.

There are various replacement semantics for replace operations. The popular regular replacement types are *reluctant replacement*, *greedy replacement*, *possessive replacement*, or *declarative replacement*. For example, the reluctant replacement matches a given `pattern` (a regular language) with the shortest substring of the `input_string`, while the greedy replacement matches the longest substring. These types produce the same result when `pattern` is a string literal. Possessive replacement semantics is that of the greedy one, but without backtracking on the substring matched so far (we match on the first longest match). Declarative replacement matches every occurrence of regular `pattern` in the given `input_string`.

A *procedural* regular replacements enforce the matching of the left-most occurrence of `pattern`. Procedural replacements encompass the greedy and reluctant replacements, since both match in `input_string` from the left to the right. They start with the first character in the word and by using backtracking continue through the word until the end of the word.

Since SMT-LIB defines only replacement operations of type reluctant replacement with the left-most matching and other types of the replacement operations are not allowed in SMT formulae in SMT-LIB format (which Z3-Noodler solves), we focus in this work solely on the reluctant replacement type.

We denote reluctant regular replacement as $x_{\pi \to y}$ where x is the `input_string`, π is the regular `pattern` (a literal or a regular language) and y is the `replacement`.

6.2 Encoding Reluctant Replacement as Transducers

SMT-LIB supports two types of reluctant replacements: a *reluctant all replacement* which replaces all occurrences of the shortest left-most matches of `pattern`, and a *reluctant single replacement* which replaces only the first shortest left-most occurrence of `pattern`.

Empty word in `pattern`. If the `pattern` contains an empty word ϵ, a special handling of ϵ matching needs to be performed. In such a case, one encodes the replacement operation as if $\epsilon \notin \pi$, and solves the replacement for two options:

- Tries solving for just ϵ manually, and alternatively
- Solves the reluctant replacement without the epsilon if the ϵ match fails.

Since SMT-LIB works only with reluctant replacements, the reluctant replacement would always pick the shortest replacement, i.e., the ϵ replacement. The reluctant match of ϵ will always succeed, and longer matches will not be needed nor accepted. SMT-LIB declares results to such patterns with ϵ to be equal to `replacement`$\cdot$`input_string` (prepending the replacement to the `input_string`). No replacement NFT needs to be constructed in this case.

Definition 6 (**Reluctant all replacement**).
A *reluctant all replacement* $x^{+}_{\pi \to y}$ is defined as follows:

$$x^{+}_{\pi \to y} = \{u \cdot y \cdot w^{+}_{\pi \to y}\}$$

where $u, v, w, x, y \in \Sigma^*$, π is a regular language (regex), $x = uvw$, $u \notin \Sigma^*\pi\Sigma^*$, $v \in \pi$, and to ensure the shortest left-most matching, the following must hold: $\forall u = u_1u_2, v = v_1v_2, w = w_1w_2$: if $v_2 \neq \epsilon$ then $v_1 \notin \pi$ (the shortest match—if v_1 is not a match, then surely the shortest match is $v = v_1v_2$); if $u_2 \neq \epsilon$ then $u_2v_1 \notin \pi \land u_2vw_1 \notin \pi$ (the left-most match is v since there is no match of π before v).

A reluctant single replacement is defined similarly, only the recursive replacement of w is no longer performed since after the first replacement of v, no more pattern matchings are tried on the substring w after the replaced substring v.

Definition 7 (**Reluctant single replacement**).
A *reluctant single replacement* $x^{1}_{\pi \to y}$ is defined as reluctant all replacement, only with the recursive replacement removed:

$$x^{1}_{\pi \to y} = \{u \cdot y \cdot w\}.$$

Example 10. For $a, b, c \in \Sigma$: $(aabaaba)^{+}_{aa \to c} = \{cbcba\}$; $(aabaaba)^{1}_{aa \to c} = \{cbaaba\}$.

6.3 Transducers for Regex Reluctant Replacement

First, we introduce a method for constructing a finite transducer $\mathcal{T}_{x^{+}_{\pi \to y}}$ for the most general case of reluctant regular replacement, a *regex reluctant all replacement* $x^{+}_{\pi \to y}$, where the pattern π being replaced is specified as a regular language, and we replace all occurrences of π.

We follow the idea of the existing procedure for constructing NFT for regex reluctant all replacement operation taken from [41], but we modify the method to fit SMT-LIB requirements for replacement operations and our implemented representation of NFTs in Mata.

The approach constructs two main NFTs where one non-deterministically finds a possible beginning of a pattern match and inserts a special symbol \$ at the found location in the input word x. The symbol \$ serves as a *begin marker* for the searched pattern.

Since we have a reluctant replacement matching the shortest left-most pattern match, we know that when we insert the correct begin marker into the input word (followed by a match of π), we can simply read the characters following \$ until we read a matching pattern. There is no need for us to explicitly mark the end of the pattern with another special symbol.[1]

However, in order to construct the NFT for the begin marker, we first construct a DFT to find a possible *end marker* \$[2] which denotes the end of a pattern. The end marker DFT is however constructed for the reverse of the searched pattern π, $\mathrm{rev}(\pi)$. This way, we deterministically find the ends of $\mathrm{rev}(\pi)$. This gives us the potential beginning of the reverse of $\mathrm{rev}(\pi)$, that is, the original π in x.

We will explain the construction on a running example. Given $x^{+}_{\pi\to y}$ where $\pi = a^+b^+c$, construct $\mathcal{T}_{x^{+}_{\pi\to y}}$.

6.3.1 End Marker DFT

Here, we construct a DFT $\mathcal{T}_{\mathrm{end}}$ marking the end of a regular pattern $\mathrm{rev}(\pi)$.

First, we construct DFA $\mathcal{A}_{\mathrm{rev}(\pi)}$ for $\mathrm{rev}(\pi)$. Note that here we treat ϵ as a special transition symbol used in the construction of $\mathcal{T}_{x^{+}_{\pi\to y}}$. Hence, $\mathcal{A}_{\mathrm{rev}(\pi)}$ cannot contain ϵ symbols. Also note that we require $\mathcal{A}_{\mathrm{rev}(\pi)}$ to be deterministic.

See Figure 6.1a for DFA $\mathcal{A}_{cb^+a^+}$ for $\mathrm{rev}(\pi) = cb^+a^+$.

$\mathcal{A}_{\mathrm{rev}(\pi)}$ accepts $\mathrm{rev}(\pi)$, but for replace operations, we need the $\mathcal{T}_{x^{+}_{\pi\to y}}$ to read any $x \in \Sigma^*$ and replace only the requested pattern, leaving the rest of x unmodified. Therefore, we *generalize* $\mathcal{A}_{\mathrm{rev}(\pi)}$, producing $\mathcal{A}^{\mathrm{gen}}_{\mathrm{rev}(\pi)}$, which accepts any word $x \in \Sigma^*$. Think of $\mathcal{A}^{\mathrm{gen}}_{\mathrm{rev}(\pi)}$ as the input tape of an NFT which needs to read any word on its input, and only perform a specific operation (replacement here) for a specified pattern in the input word.

Given $\mathcal{A}_{\mathrm{rev}(\pi)} = (Q, \Sigma, \mathit{post}, \{q_0\}, F)$, we construct $\mathcal{A}^{\mathrm{gen}}_{\mathrm{rev}(\pi)} = (Q^{\mathrm{gen}}, \Sigma, \mathit{post}^{\mathrm{gen}}, \{s_0\}, F^{\mathrm{gen}})$ where $Q^{\mathrm{gen}} = \{t \mid t \subseteq Q\}$ following these rules:
We define a bijective mapping $\mathcal{M} : Q^{\mathrm{gen}} \to 2^Q$ such that $\forall s \in Q^{\mathrm{gen}}, a \in \Sigma : s' \in \mathit{post}^{\mathrm{gen}}(s, a)$ iff $\mathcal{M}(s') = \{q_0\} \cup \{q' \mid \exists q \in \mathcal{M}(s) : q' \in \mathit{post}(q, a)\}$ and $\mathcal{M}(s_0) = \{q_0\}$.

Intuitively, we construct a new DFA from $\mathcal{A}_{\mathrm{rev}(\pi)}$ where each state s maps to a set of states in $\mathcal{A}_{\mathrm{rev}(\pi)}$ which can be reached from q_0 with the same substrings of $\mathcal{L}(\mathcal{A}_{\mathrm{rev}(\pi)})$. Since $\mathcal{A}_{\mathrm{rev}(\pi)}$ is deterministic, each substring will end in at most one state s. Thus, $\mathcal{A}^{\mathrm{gen}}_{\mathrm{rev}(\pi)}$ remains deterministic.

However, when we encounter state s where $F \cap \mathcal{M}(s) \neq \emptyset$, we know that we have reached a state in the constructed $\mathcal{A}^{\mathrm{gen}}_{\mathrm{rev}(\pi)}$ where the substring corresponding to s is accepted by $\mathcal{A}_{\mathrm{rev}(\pi)}$. That means that we have reached an end of an accepted word in $\mathcal{A}_{\mathrm{rev}(\pi)}$ which we want to mark (insert an end marker \$ after this substring). Therefore, during the

[1]This would not be the case for, e.g., the greedy replacement, however.

[2]For simplicity, we use the same symbol to denote both the begin marker and the end marker. The reason for this is that these two markers are never used simultaneously and there is a close connection between them. The connection will be further explained later.

construction, we create a new final state s_f, and each time we encounter such a state s, we add a transition $s \xrightarrow{\epsilon} s_f$, and redirect all transitions which would normally go from s to start in s_f. Thus, the only outgoing transition from s is the epsilon transition, and we did not introduce a non-deterministic choice of transitions from s. Then, all states except for states s where $F \cap \mathcal{M}(s) \neq \emptyset$ are made final since we want to accept any arbitrary input. We omit the original final states, since reaching these states is a significant event in reading an arbitrary input, which we need to handle separately.

$\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$ represents an DFA which states how "far" into $\text{rev}(\pi)$ we are when reading an arbitrary input. Each symbol either resets DFA (simulating manual backtracking) to some previous matched state or all the way up to s_0 (no even partial match was found, we start from the beginning of the pattern), or advances to the next state in the DFA following the pattern.

Now, $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$ accepts any string x on its input. During the run of x on $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$, each time an end of a substring x_2 is encountered where $x_2 \in \text{rev}(\pi)$ and $x = x_1x_2x_3$, an epsilon transition is taken. If we were to insert \$ into x each time an epsilon transition is taken, we would modify x in such a way that the word would remain the same, only interspersed with \$ after each end of x_2 (since $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$ is deterministic, no other transition can be taken from the source state of the epsilon transition). These \$ represent the end markers we want to insert with $\mathcal{T}_{\text{end}}$.

See Figure 6.1b showing a generalized modification of DFA $\mathcal{A}_{cb^+a^+}$, $\mathcal{A}^{\text{gen}}_{cb^+a^+}$. For example, a word *cbcbba\$a\$a\$bcb* would be accepted by this automaton, where the imaginary string symbol \$ symbolically represents places where an epsilon transition was taken in the automaton.

Next, we can construct $\mathcal{T}_{\text{end}}$ from $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$. We simply consider the current transitions as the input tape of $\mathcal{T}_{\text{end}}$, and we add transition symbols for the output tape: for all $a \in \Sigma$, the symbol on the output tape is a again, only for the input symbol ϵ, the output symbol is \$.

We can see the final $\mathcal{T}_{\text{end}}$ for $\mathcal{A}^{\text{gen}}_{cb^+a^+}$ in Figure 6.1c.

6.3.2 Begin Marker NFT

We could start constructing the *begin marker* NFT $\mathcal{T}_{\text{begin}}$ from $\mathcal{T}_{\text{end}}$ by reversing all transitions in $\mathcal{T}_{\text{end}}$. However, considering reversing a single transducer transition (consisting of operations on the input and output tapes), represented as a sequence of two NFA transitions in MATA from state with level 0 to 1 and back to some state with level 0, we instead utilize $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$ before its conversion to $\mathcal{T}_{\text{end}}$ (and $\mathcal{T}_{\text{end}}$ is therefore never truly constructed in MATA).

We reverse all transitions in $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$, effectively obtaining the foundation for an NFA for marking beginnings of π. We can now construct $\mathcal{T}_{\text{begin}}$ the same way as we constructed $\mathcal{T}_{\text{end}}$ from $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$, except now \$ as the output symbol for the epsilon symbol on the input tape represents the begin marker instead of the end marker. This is why we use the same marker for symbols for both the begin and end marker, as noted earlier. They mark the same location, only once operating on $\text{rev}(\pi)$, then on π.

However, marking beginnings is a non-deterministic operation. We cannot know in advance whether the currently read symbol in an arbitrary input string marks a beginning of a full match in π. Henceforth, we have to introduce another source of non-determinism, which gives us the smart "look-ahead" ability for marking beginnings for only those inputs that will later on truly match in π. All potentially incorrectly inserted begin markers in the input word end in a non-accepting run for the word with such begin markers. We add a new initial state s_0' and add ϵ transducer transitions $s_0' \xrightarrow{\epsilon} s$ (ϵ as the transition symbol on all tapes) leading to all states s which are final in $\mathcal{A}^{\text{gen}}_{\text{rev}(\pi)}$, that is, all except those which map to a set of states containing the original final state from $\mathcal{A}_{\text{rev}(\pi)}$. The previous initial state of $\mathcal{T}_{\text{begin}}$ is set as the only final state and is removed from the initial states.

The Figure 6.2 shows a $\mathcal{T}_{\text{begin}}$ for rev$(\mathcal{A}^{\text{gen}}_{cb^+a^+})$.

6.3.3 Reluctant Replacement NFT

When we have the begin marker NFT $\mathcal{T}_{\text{begin}}$, we can construct the second NFT performing the actual replacement. These two NFTs are used to finally construct the whole $\mathcal{T}_{x^+_{\pi\to y}}$.

We already can non-deterministically find the correct begin markers, but we now need to perform the actual replacement(s) on the input string annotated with begin markers.

We construct an NFT $\mathcal{T}^{\$}_{x^+_{\pi\to y}}$ which models the actual reluctant replacement. The idea is that we read the input string annotated with begin markers and output the same symbols unmodified on the output tape. Once we encounter the begin marker, we know we have found the first left-most match, which needs to be replaced. Therefore, we switch to a "replacement" mode where we read the match without outputting anything, removing all symbols of the match and all additional begin markers inside the match. After the match is read, we can insert the replacement for the match onto the output tape. When the replacement is inserted, we return to the beginning to potentially perform replacement on another match.

First, we convert the $\mathcal{A}_\pi$ into a new NFA $\mathcal{A}^{\text{short}}_\pi$ which accepts (matches) only the shortest input words. This can be easily performed by removing all outgoing transitions from all final states in $\mathcal{A}_\pi$. Clearly,

$$\mathcal{L}(\mathcal{A}^{\text{short}}_\pi) = \{u \in \mathcal{L}(\mathcal{A}_\pi) \mid \neg\exists u', v \in \Sigma^* : u' \in \mathcal{L}(\mathcal{A}_\pi)^{<|u|} \wedge u = u'v \wedge v \neq \epsilon\}.$$

$\mathcal{A}^{\text{short}}_\pi$ now matches the shortest words in the language of $\mathcal{A}_\pi$, but does not allow skipping over additional \$ encountered during the reading of the match. Such \$ need to be skipped, since \$ inside the shortest left-most match represent beginnings of potential matches further to the right than the left-most match. We need to therefore add a self loop to every state in $\mathcal{A}^{\text{short}}_\pi$ with transition symbol \$.

Now, the final modification is to keep the next begin marker for the next shortest left-most match. $\mathcal{A}^{\text{short}'}_\pi$ can be constructed given $\Sigma^*_{\$} = \Sigma^* \cup \{\$\}$ as $\mathcal{A}^{\text{short}'}_\pi = \mathcal{A}^{\text{short}}_\pi \cap \overline{\Sigma^*_{\$} \cdot \{\$\}}$.

$\mathcal{A}^{\text{short}'}_\pi$ can be converted into $\mathcal{T}^{\text{short}'}_\pi$ where the existing transitions are the input tape transitions, and the output symbols are all ϵ (NFT only reading the input word—the match—and consuming the whole match including \$ inside the match).

Figure 6.3a symbolically shows how to construct $\mathcal{T}^{\$}_{x^+_{\pi\to y}}$ for replacing all shortest left-most occurrences of regular language π with y.

To replace only one occurrence (the first shortest left-most match), a variation of $\mathcal{T}^{\$}_{x^+_{\pi\to y}}$, $\mathcal{T}^{\$}_{x^1_{\pi\to y}}$ can be constructed, as seen in Figure 6.3b. Here we do not return to the beginning after the first replace, but move to a new state with just reads the rest of the word unmodified while removing the additional \$.

6.3.4 Whole Regex Reluctant Replace NFT

We have constructed $\mathcal{T}_{\text{begin}}$ to insert non-deterministically begin markers into the input word, and constructed $\mathcal{T}^{\$}_{x^{+}_{\pi\to y}}$ (or $\mathcal{T}^{\$}_{x^{1}_{\pi\to y}}$) to perform the replacement upon encountering matches prepended with \$. $\mathcal{T}_{x^{+}_{\pi\to y}}$ (and similarly $\mathcal{T}_{x^{1}_{\pi\to y}}$) can be constructed by connecting these two NFTs into a sequence using composition: $\mathcal{T}_{x^{+}_{\pi\to y}} = \mathcal{T}_{\text{begin}}||\mathcal{T}^{\$}_{x^{+}_{\pi\to y}}$ and $\mathcal{T}_{x^{1}_{\pi\to y}} = \mathcal{T}_{\text{begin}}||\mathcal{T}^{\$}_{x^{1}_{\pi\to y}}$.

$$\mathcal{L}(\mathcal{T}_{x^{+}_{\pi\to y}}) = \{(u,v) \in \Sigma^* \times \Sigma^* \mid v = u^{+}_{\pi\to y}\}, \mathcal{L}(\mathcal{T}_{x^{1}_{\pi\to y}}) = \{(u,v) \in \Sigma^* \times \Sigma^* \mid v = u^{1}_{\pi\to y}\}.$$

6.4 Transducers for Literal Reluctant Replacement

Constructing NFTs for regex reluctant replacement is expensive. Therefore, when the `pattern` to be replaced is of a simpler form, namely a string literal, or even a single symbol, a corresponding NFT for reluctant replacement can be constructed faster.

6.4.1 Single Symbol Replacement

We can construct a replacement DFT for a single symbol reluctant replacement by creating a single-state DFT with Id(Σ) self-loop transducer transitions. Then only modify the output transition for the input symbol by replacing it with a transition (or a sequence of transitions) outputting y on the output tape (y can be either a single symbol or a literal) to produce the replacement. The handling of all and single reluctant replacement variations is the same as for regex reluctant replacement: we either return to the original state, or create a new final state with Id(Σ) self-loop transitions.

6.4.2 Replacement of Finite String Literal

To construct an NFT for the replacement of a string literal z with y, we need to construct two NFTs.

First, we want to annotate the end of the input words x similarly to how $\mathcal{T}_{\text{end}}$ annotates the end of the patterns, but now a simple NFT $\mathcal{T}_{\$}$ with an initial state q_0, final state q_f, and transitions $(q_0, \text{Id}(\Sigma), q_0)$, and $(q_0, (\epsilon, \$), q_f)$ will suffice. $(u, u\$) \in \mathcal{L}(\mathcal{T}_{\$})$.

Second, we need to construct an NFT $\mathcal{T}^{\$}_{x^{+}_{z\to y}}$ for replacing all literals, or $\mathcal{T}^{\$}_{x^{1}_{z\to y}}$ for replacing only the first left-most the shortest literal.

$\mathcal{T}^{\$}_{x^{+}_{z\to y}} = (Q, \Gamma, post, \{q_0\}, \{q_f\})$ can be constructed using the following method. We construct an NFT for z where z represents the input tape and a sequence of ϵ symbols represents the corresponding output tape. Each state q uniquely maps to its corresponding substring of y: $\mathcal{M} : Q \to \Sigma^*$ is a bijection where $\mathcal{M}(q_0) = \epsilon$, $\mathcal{M}(q_1) = y[0:1]$, $\mathcal{M}(q_1) = y[0:2]$, $\mathcal{M}(q_2) = y[0:3]$, etc. Each state therefore represents a "buffer" of symbols read so far on the input tape which have not been outputted on the output tape yet. We already have transitions as constructed from z for each state.

Now we add additional transitions as follows:

- If \$ is read on the input tape, we output the contents of the buffer $\mathcal{M}(q)$ where q is the current state (symbols read from the input word but not outputted onto the output tape yet) and move to q_f.

- If $a \in \Sigma$ and $z = \mathcal{M}(q) \cdot a$ (a is the last symbol in y), we perform the replacement: outputting y, and move to the beginning q_0 to continue replacing the next match.
- If $a \in \Sigma$ and $a \neq z[|\mathcal{M}(q)| : |\mathcal{M}(q)| + 1]$ (a is not the next symbol in z—those transitions are already in $\mathcal{T}^{\$}_{x^+_{z\to y}}$), we determine the longest prefix p of $z = pv$ such that $up = \mathcal{M}(q) \cdot a$ from left to right. We output u onto the output tape (as a sequence of transition symbols) and move to state $q' = \mathcal{M}(p)$ which represents the longest prefix of z currently being "buffered" and not processed yet.

When constructing $\mathcal{T}^{\$}_{x^1_{z\to y}}$, instead of moving to q_0 after reading z and outputting the replacement y, we move to a final state q_f, and add transitions to simply read the rest of the input word and output the word unmodified onto the output tape, removing \$ at the end: $(q_f, \mathrm{Id}(\Sigma), q_f)$ and $(q_f, (\$, \epsilon), q_f)$.

To construct $\mathcal{T}_{x^+_{z\to y}}$ (or $\mathcal{T}_{x^1_{z\to y}}$), a composition of $\mathcal{T}_{\$}$ and $\mathcal{T}^{\$}_{x^+_{z\to y}}$ (or $\mathcal{T}^{\$}_{x^1_{z\to y}}$) is performed: $\mathcal{T}_{x^+_{z\to y}} = \mathcal{T}_{\$} || \mathcal{T}^{\$}_{x^+_{z\to y}}$ or $\mathcal{T}_{x^1_{z\to y}} = \mathcal{T}_{\$} || \mathcal{T}^{\$}_{x^1_{z\to y}}$.

Example 11. For literal reluctant all replacement $x^+_{cc\to a}$, we can see that $\mathcal{M}(0) = \epsilon$, $\mathcal{M}(1) = c$, $\mathcal{M}(2) = cc$.

Figure 6.4 shows both variations (replace all and replace single) NFTs for $x^+_{cc\to a}$.

Thanks to our approach, $\mathcal{T}_{x^+_{z\to y}}$ and $\mathcal{T}_{x^1_{z\to y}}$ maintain the same structure after composition as $\mathcal{T}^{\$}_{x^+_{z\to y}}$ and $\mathcal{T}^{\$}_{x^1_{z\to y}}$. The only difference is that marker symbols \$ are replaced with ϵ.

Note that this approach could be generalized to a regex replacement where the regex has finite length, similarly to how [41] models finite regex replacements. However, for our uses in Z3-Noodler, we currently do not solve any industry benchmarks where finite regex replacement occurs. Further, when modelling finite regex replacement for regex π with the maximal length n of a string $u \in \mathcal{L}(\pi)$, one needs to create states for all possible substrings of $\Sigma^{\leq n}$, each with the transitions similar to those in $\mathcal{T}_{x^+_{z\to y}}$. This is computationally costly, and the constructed state space increases dramatically.

6.5 Solving SMT Formulae with Replace Operations

The main decision procedure of Z3-Noodler rewrites SMT formulae into a form where the formula contains regular constraints (where both the variables and the regular constraints have assigned regular languages) and Z3-Noodler checks whether the languages for variables after restricting the languages by the string constraints are non-empty. If during the procedure a language of some variable becomes empty, there is no assignment for the formula which could assign the variable a word satisfying the constraints.

The formulae solved by Z3-Noodler which contain replace operations always follow the following general structure:

$$x \in \ldots \land u = \texttt{replace}(\ldots(\texttt{replace}(\texttt{replace}(x, \pi_1, y_1), \pi_2, y_2)\ldots), \pi_n, y_n) \land u \in \ldots.$$

We have a sequence of nested replace operations. There can be between one and about 200 nested replace operations on the benchmarks from industry solved by Z3-Noodler, usually working with small tens of unique transition symbols[3].

[3]This is thanks to that our decision procedure in Z3-Noodler substitutes transition symbols not explicitly worked with in the formula with just two "dummy" transition symbols.

Example 12. An encoding for operation `toUpper` which replaces all lowercase letters in x with uppercase letters looks like this:

$$x \in \ldots \land u = \texttt{replace}(\ldots(\texttt{replace}(\texttt{replace}(x, \text{`a`}, \text{`A`}), \text{`b`}, \text{`B`})\ldots), \text{`z`}, \text{`Z`}) \land u \in \ldots .$$

.

Another often performed operation is `HTMLEscape` which properly escapes special HTML symbols from the input string x so as not to break the HTML when the user input gets inserted into the HTML webpage:

$$x \in \ldots \land u = \texttt{replace}(\ldots(\texttt{replace}(\texttt{replace}(x, \text{`<`}, \text{``\&lt;``}), \text{`\&`}, \text{``\&amp;``})\ldots), \texttt{from\_code}(39), \text{``\&\#39;``}) \land u \in \ldots$$

where $\texttt{from\_code}(n)$ performs a conversion from integer code n to its corresponding Unicode character.

And finally, there are operations working with regexes π:

$$x \in \ldots \land u = \texttt{replace}(x, \text{``}\texttt{.d}^{+}\texttt{.}\text{``}, \text{``\_\$1``}) \land u \in \ldots$$

where regex "$\texttt{.d}^{+}\texttt{.}$" represents a regex describing one or more occurrences of `d` surrounded by one character from both sides. For example, `adde` matches, but `aed` does not.

All of these examples come with variants for replacing all or only the first occurrence of the regex or literal, and with varying number of nested replacements.

The general approach to extend string solvers with support for replace operations and string transformations in general has already been pioneered in the context of monadic decomposition-based algorithms, as seen in [23, 1, 5, 47]. However, adding this support for Z3-Noodler for its stabilization decision procedure is a challenging task with interesting qualities.

As seen in Example 12, we often reason about regular languages for both the input and output language of a sequence of nested replace operations. Internally in Z3-Noodler, each level of the nested hierarchy will get assigned a new fresh variable x_i, so we obtain from

$$x \in \ldots \land u = \texttt{replace}(\ldots(\texttt{replace}(\texttt{replace}(x, \pi_1, y_1), \pi_2, y_2)\ldots), \pi_n, y_n) \land u \in \ldots .$$

a formula similar to

$$x \in \ldots \land u_1 = \texttt{replace}(x, \pi_1, y_1) \land u_2 = \texttt{replace}(u_1, \pi_2, y_2) \land \ldots \land u = \texttt{replace}(u_{n-1}, \pi_n, y_n) \land u \in \ldots .$$

Since we need to propagate the properties of x over all the replace operations to u, we need a method to propagate the information over the replace operation. Remember that now we consider both u and x to be string variables with an assigned regular language, $\mathcal{L}(u)$ and $\mathcal{L}(x)$. The problem can be therefore split into smaller problems of form $u = x^{+}_{\pi \to y}$ (representing a single replace operation from above) and propagating the information from x to u.

In string solving, we work with NFTs with two tapes: the input tape and the output tape. We define $\texttt{project\_input}(\mathcal{T})$ and $\texttt{project\_output}(\mathcal{T})$ as special cases of projections, performing $\texttt{project\_to}_{\{0\}}(\mathcal{T})$ and $\texttt{project\_to}_{\{1\}}(\mathcal{T})$, respectively. Performing projection on either tape produces an NFA representing the language of the corresponding tape. The projection of NFT $\mathcal{T}$ to its output tape, `project_output`($\mathcal{T}$), gives us the forward image of $\mathcal{T}$, NFA $\mathcal{A}_{\text{out}}$. Similarly, the projection of NFT $\mathcal{T}$ to its input tape, `project_input`($\mathcal{T}$), gives us the backward image of $\mathcal{T}$, NFA $\mathcal{A}_{\text{in}}$.

For the given problem $u = x^{+}_{\pi \to y}$ with $\mathcal{A}_x$ and $\mathcal{A}_u$ being the NFAs representing the regular languages assigned to variables x and u, we can compute the language (the corresponding NFA) of the forward image as `project_output`($\mathcal{T}_{\text{Id}(\mathcal{A}_x)} || \mathcal{T}_{x^{+}_{\pi \to y}}$), i.e., the forward application (propagation) of $\mathcal{T}_{x^{+}_{\pi \to y}}$ on $\mathcal{A}_x$ which gives us the admissible $\mathcal{L}(u)$ for the given $\mathcal{L}(x)$. The backward image can be computed as `project_input`($\mathcal{T}_{x^{+}_{\pi \to y}} || \mathcal{T}_{\text{Id}(\mathcal{A}_u)}$), that is, backward application (propagation) of $\mathcal{T}_{x^{+}_{\pi \to y}}$ on $\mathcal{A}_u$ which gives us the admissible $\mathcal{L}(x)$ for the given $\mathcal{L}(u)$.

Languages of variables as concatenations of regular languages A decision procedure in Z3-Noodler, called *stabilization* [27] performs automata operations on languages for variables which may be represented as a concatenation of multiple regular languages for other variables. For an algorithm called *noodlification* performed as a part of this procedure, we need to concatenate these languages with special epsilon symbols (different from those used in construction of NFTs for replace operations). Therefore, we need to handle these symbols during replace operations as regular symbols, not epsilon symbols, in order to maintain a clear separation of languages of each individual variables in the concatenation.[4].

Flattening of the nested hierarchy of replace operations. To further optimize the computation of u from x through a nested hierarchy of replace operations, one can precompute $\mathcal{T}_{\text{seq}}$ as a sequential composition of all NFTs for the corresponding replace operations.

Then, the hierarchy is flattened into a single $\mathcal{T}_{\text{seq}}$ performing all the replace operations simultaneously: $u = \mathcal{T}_{\text{seq}}(x)$.

Note that in the benchmarks solved by Z3-Noodler, the patterns being replaced are unique in that the replacement of one NFT is never later in the sequence matched to the pattern of another NFT. E.g., operation `toUpper` performs a sequence of transductions, each matching one of the lowercase letters of the alphabet and replacing it with its uppercase version. It never happens that the uppercase letter replacing the lowercase one is later replaced with some other letter. Therefore, an optimization of flattening a sequence of transducers satisfying this property can be utilized: the transductions in the whole sequence are associative. We can construct the intermediate NFTs for composition of any combination of NFTs in the sequence, disregarding the order of the NFTs in the sequence, to more quickly construct the final NFTs for the whole sequence. The reason for why the nested hierarchy is necessary for SMT formulae encoded in SMT-LIB format is that since the SMT-LIB format does not allow specifying a single replace operation with multiple (`pattern`, `replacement`) pairs, performing operations such as `toUpper` can be encoded only as a sequence of independent replacements which is however a suboptimal encoding for SMT solvers, especially those which can utilize the abilities of NFTs for the representation of replace operations.

[4]The epsilon nature of these special symbols is utilized only in noodlification to indicate move between languages of variables in the concatenated language.

Length constraints. Some variables may be length constrained in addition to the string constraints. Length constraints for variables are in Z3-Noodler solved separately using a LIA (linear integer arithmetic) solver provided by Z3. If the variables u and x are length variables, we need to maintain the information about lengths through the replace operation (which can in general change the length of words between u and x). To propagate lengths of words through the replace operations, an option would be to compute an existential Presburger formula ϕ based on a computation of a Parikh image of $x^{+}_{\pi \to y}$, as described in [5].

We compute the standard Parikh image [77] on the NFT for replace operation, with the whole n-tuples of transition symbols being handled as single macro symbols (obtaining an NFA over the language of said n-tuples). ϕ now encodes the relationship between the number of occurrences of n-tuples (macro symbols) in $s \in \mathcal{L}(\mathcal{T}_{x^{+}_{\pi \to y}})$. To get the relationship between the actual letters $a \in \Sigma$ inside the n-tuples α, we must extend ϕ with variables a_i counting the number of occurrences of each a on each tape i (i-th element of α), and compute the relationship between each a_i and $\#\alpha$, representing the number of α symbols in the Parikh image, as: $a_i = \sum_{\alpha \in \Sigma^n_\epsilon : \alpha[i] = a} \#\alpha$ where it must hold that $|x_i| = \sum_{a \in \Sigma} a_i$ is the length of words on each tape i.

The existing decision procedure of Z3-Noodler will have to be modified further as follows:

- Adding saturation of replace operations, similarly to how Z3-Noodler saturates other string functions,
- Inclusion graph [17] used by Z3-Noodler will have to be extended to support replace operations.

6.6 Implementation.

The discussed algorithms for modelling replace operations using NFTs were implemented in Mata. The code is accessible in a GitHub repository[5] and will be merged into the main Mata GitHub repository[6].

Words and identity insertion. We also implemented a few utility functions which are useful when constructing or modifying NFTs[7] such as

- *inserting an identity transition* over all tapes (all tapes read/write the same symbol) over the given alphabet for a specified state.

  This is specially useful for string solving where usually one specially handles a small subset of transitions symbols, but wants to leave the remaining symbols in the input word unmodified.

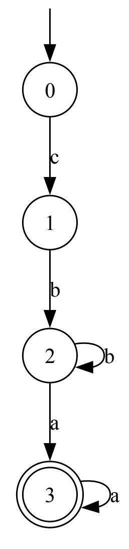
- *inserting a word* for specified tapes or *inserting words*, one for each tape.

  This allows one to quickly create transducers which read a specific word, replace a specific string literal with a given literal, or remove a specified literal from the input word.

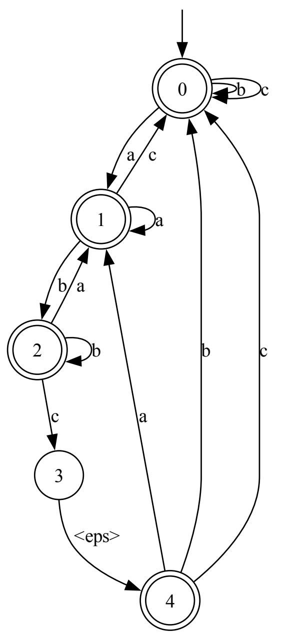
[5]https://github.com/Adda0/mata/tree/transducers

[6]https://github.com/VeriFIT/mata/

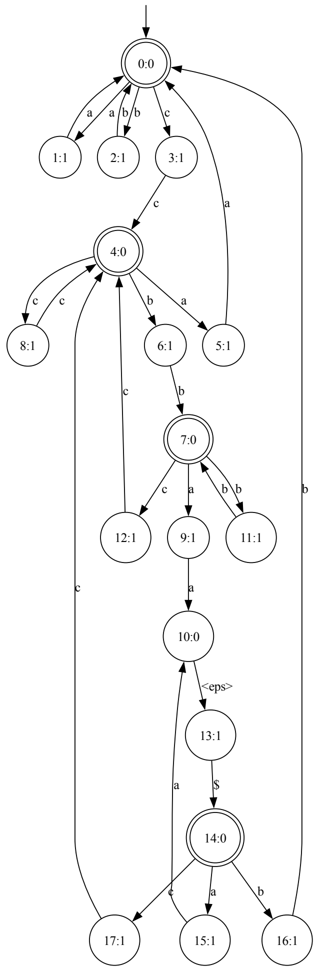
[7]Especially during construction of NFTs for the replace operations and other often used operations in string solving.



(a) DFA $\mathcal{A}_{cb^+a^+}$, serving as a basis for constructing $\mathcal{T}_{\text{end}}$.



(b) DFA $\mathcal{A}^{\text{gen}}_{cb^+a^+}$, a generalized version of $\mathcal{A}_{cb^+a^+}$.



(c) DFT $\mathcal{T}_{\text{end}}$ for $\mathcal{A}^{\text{gen}}_{cb^+a^+}$ inserting the end markers \$ at the correct locations in the read input word.

Figure 6.1: Automata created during $\mathcal{T}_{\text{end}}$ construction.

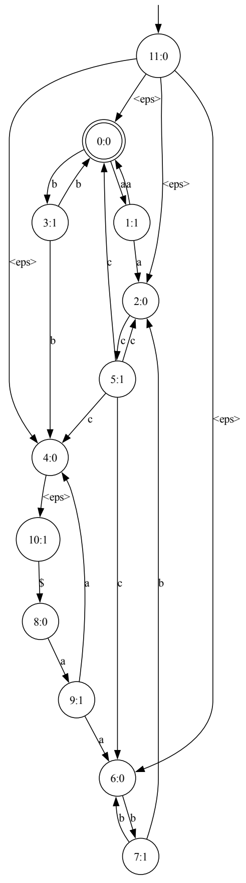
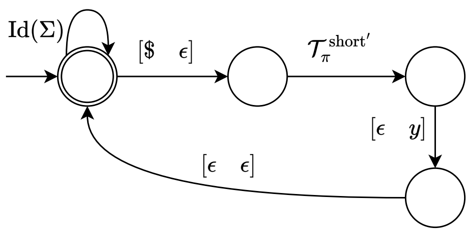
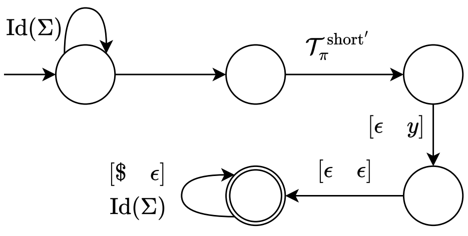


Figure 6.2: NFT $\mathcal{T}_{\text{begin}}$ for $\text{rev}(\mathcal{A}^{\text{gen}}_{cb^+a^+})$ inserting non-deterministically, the begin markers \$ at the beginning of matches in a^+b^+c for the read input word.

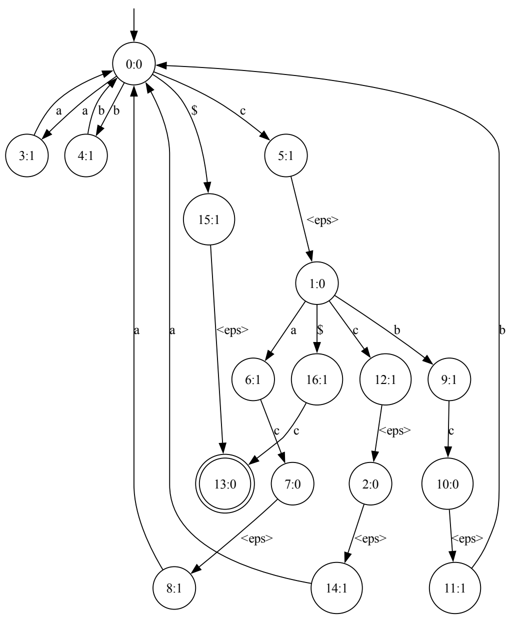


(a) Symbolic representation of $\mathcal{T}^{\$}_{x^+_{\pi\to y}}$ for replacing all shortest left-most occurrences of π with y. All occurrences of π are being replaced.

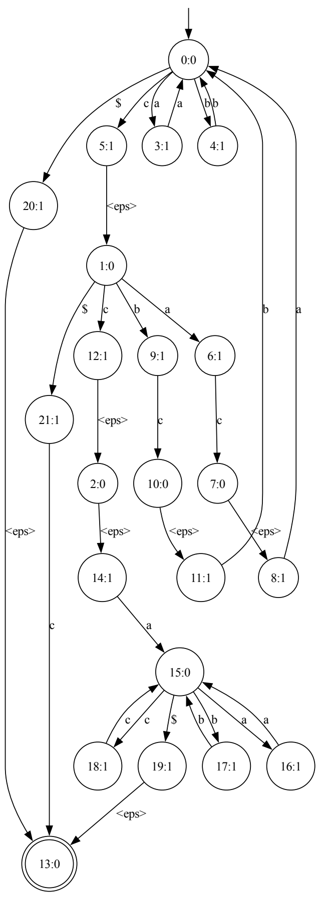


(b) Symbolic representation of $\mathcal{T}^{\$}_{x^1_{\pi\to y}}$ for replacing single shortest left-most occurrence of π with y. the remaining occurrences of π are left unmodified.

Figure 6.3: Symbolic representations of $\mathcal{T}^{\$}_{x^+_{\pi\to y}}$ and $\mathcal{T}^{\$}_{x^1_{\pi\to y}}$.



(a) $\mathcal{T}^{\$}_{x^{+}_{z\to y}}$ for replacement $x^{+}_{cc\to a}$.



(b) $\mathcal{T}^{\$}_{x^{1}_{z\to y}}$ for replacement $x^{1}_{cc\to a}$.

Figure 6.4: $\mathcal{T}^{\$}_{x^{+}_{z\to y}}$ and $\mathcal{T}^{\$}_{x^{1}_{z\to y}}$ for $x^{+}_{cc\to a}$ and $x^{1}_{cc\to a}$ replacements.

Chapter 7

Experimental Evaluation

We implemented NFTs with a simple data structure inherited from NFAs without having to substantially modify the NFA algorithms. For our experiments, we are interested in how MATA performs when running operations on NFTs modelling the replace operations required by Z3-NOODLER. Modelling replace operations in Z3-NOODLER is not the only use case for NFTs in MATA, but it is the first domain where NFTs will be used. We want to see whether MATA is performant enough for our use cases and whether the performance of MATA is comparable with the well-optimized state-of-the-art.

We compare the performance of MATA with a state-of-the-art automata library MONA [44]. Even if MONA does not support finite transducers directly, we have chosen MONA to compare MATA against because MONA is a well-optimized library which, thanks to its ability to encode transition relation into MTBDDs, provides a reasonable model for encoding finite transducers. Each tape in a transducer can be encoded as a single variable in the BDD for a given transition. A sequence of variables then encodes an NFT transition, where the leaves of the MTBDDs are target states and edges represent the transition symbols. Each transition symbol is encoded as a bit vector (a sequence of bits). An NFT transition is then encoded as `dummy_1|bitvec_1|dummy_2|bitvec_2` where `|` denotes the concatenation of bit vectors. `bitvec_i` represents a bit vector for a transition symbol on the tape i. `dummy_i` are additional dummy symbols used for determinization (as MONA implicitly determinizes after every operation).

Performing operations such as composition, projection, or application on NFTs is, thanks to how MATA encodes NFTs as a representation very close to the representation of BDDAs, a natural operation for MONA in the encoding explained above.

Intuitively, MATA performs modified NFA operations on NFTs while MONA performs its BDD-like operations with the proper encoding.

Our goal in these experiments is not to outperform MONA, but to compare how techniques for NFTs using explicit alphabets with support for nondeterminism (MATA) perform against symbolic alphabets with binary representation of transition relations (binary bit vectors in MONA) without support for nondeterminism.

7.1 Benchmarks.

For the first group of benchmarks, we have generated a new set of benchmarks from runs of Z3-NOODLER on benchmarks from SMT-LIB [13] which contain string replace operations `str.replace`, `str.replace_all`, `str.replace_re` and `str.replace_re_all`. The replace

operations appear in SMT-LIB benchmarks
`non-incremental/QF_SLIA/20230331-transducer-plus`
(TransducerPlus, called `transducer plus` in our experiments), and
`non-incremental/QF_SLIA/20230403-webapp`
(Webapp, called `webapp` in our experiments).

We compare the performance of Mata and Mona on benchmarks for the following operations

- projection to input/output tapes,
- application on a literal (forward/backward),
- application on a language (forward/backward),
- composition of two NFTs, and
- composition of NFTs used during replace NFT construction.

We collect NFTs for each unique replace operation in benchmarks from SMT-LIB solved by Z3-Noodler. Furthermore, we extend this list by constructing NFTs for partial compositions of a sequence of replace operations (various subsequences) which appear in the SMT-LIB benchmarks.

Projection benchmark. The projection benchmark consists of unique single 2-tape NFTs for replace operations found in SMT-LIB formulae solved by Z3-Noodler. We perform projection to tape 0 (projecting out the tape 1), and projection to tape 1 (projecting out the tape 0). We remove duplicated NFTs in each benchmark, which in total produces 1749 instances (202 from Webapp, 1547 from TransducerPlus).

Application on a literal benchmark. The application on a literal benchmark consists of pairs $(u, \mathcal{T})$ where u is a string literal and $\mathcal{T}$ is a 2-tape NFT for replace operations found in SMT-LIB formulae solved by Z3-Noodler. u is constructed as the longest word in the `pattern` of the replace operation constrained to the maximal length of word 5 (5 is up to a few exceptions a sufficient length for all replace operations in SMT-LIB). We perform the forward application of $\mathcal{T}$ on u $(\mathrm{Id}(u)||\mathcal{T})$. This simulates the forward image propagation, as described in 6.5, which is an operation to be used in Z3-Noodler. We remove duplicated pairs $(u, \mathcal{T})$ in each SMT-LIB benchmark, which leaves us in total with 2027 instances (211 from Webapp, 1816 from TransducerPlus).

In addition, we create a second benchmark performing the backward application of $\mathcal{T}$ on u $(\mathcal{T}||\mathcal{T}_{\mathrm{Id}(u)})$. This simulates the backward image propagation, as described in 6.5, which is an operation to be used in Z3-Noodler. We remove duplicated pairs $(\mathcal{T}, u)$ in each SMT-LIB benchmark, which leaves us in total with 2030 instances (214 from Webapp and 1816 from TransducerPlus).

Application on a language benchmark. The application on a language benchmark consists of pairs $(\mathcal{T}_{\mathrm{Id}(\Sigma^*)}, \mathcal{T})$ where $\mathcal{T}_{\mathrm{Id}(\Sigma^*)}$ is a regular language Σ^* converted into an identity NFT, and $\mathcal{T}$ is a 2-tape NFT for replace operations found in SMT-LIB formulae solved by Z3-Noodler. We choose $\mathcal{T}_{\mathrm{Id}(\Sigma^*)}$ as the regular language since Σ^* is the most difficult case for our method to compute (for each state in $\mathcal{T}$, we need to compute operations for all symbols in the language). We perform the forward application of $\mathcal{T}$ on $\mathcal{T}_{\mathrm{Id}(\Sigma^*)}$ $(\mathcal{T}_{\mathrm{Id}(\Sigma^*)}||\mathcal{T})$.

This simulates the forward image propagation, as described in 6.5, which is an operation to be used in Z3-Noodler. We remove duplicated pairs $(\mathcal{T}_{\mathrm{Id}(\Sigma^*)}, \mathcal{T})$ in each benchmark, which in total produces 1750 instances (203 from Webapp, 1547 from TransducerPlus).

In addition, we create a second benchmark from forward language application benchmark, performing the backward application of $\mathcal{T}$ on $\mathcal{T}_{\mathrm{Id}(\Sigma^*)}$ $(\mathcal{T}||\mathcal{T}_{\mathrm{Id}(\Sigma^*)})$. This simulates the backward image propagation, as described in 6.5, which is an operation to be used in Z3-Noodler, too. The number of instances is the same as for the forward language application benchmark.

Composition. Last but not least, we compare the performance of Mata and Mona on the composition operation. The composition benchmark contains pairs $(\mathcal{T}_1, \mathcal{T}_2)$ where we perform composition $\mathcal{T}_1||\mathcal{T}_2$. We performed composition of various NFTs appearing in a sequence in SMT-LIB benchmarks, including composition of pre-constructed partial compositions of NFTs for replace operations in these sequences.

We remove duplicated pairs $(\mathcal{T}_1, \mathcal{T}_2)$ in each SMT-LIB benchmark, which leaves us in total with 2879 instances (130 from Webapp, 2749 from TransducerPlus).

Moreover, we collect intermediate NFTs $\mathcal{T}_{\mathrm{begin}}$ and $\mathcal{T}^{\$}_{x^{+}_{\pi\to y}}$ (or $\mathcal{T}^{\$}_{x^{1}_{\pi\to y}}$) from all $\mathcal{T}_{x^{+}_{\pi\to y}}$ (or $\mathcal{T}_{x^{1}_{\pi\to y}}$). We remove duplicated pairs $(\mathcal{T}_{\mathrm{begin}}, \mathcal{T}^{\$}_{x^{+}_{\pi\to y}})$ in each SMT-LIB benchmark, which leaves us in total with 2282 instances (252 from Webapp, 2030 from TransducerPlus).

Pattern matching benchmark. For a second benchmark, we introduce a regular pattern with counters, appearing often in works on pattern matching [50], containing complext `replaceAll` operations. The pattern looks similar to the following regular expression: `#.*u.{i, j}u.{k, l}$` where `#` denotes the beginning of a line, `.` matches any character, `*` denotes Kleene star (zero or more repetitions), `{i, j}` denotes a repetition where the number of repetitions is between i and j, and \$ denotes the end of a line. `u` stands here for an arbitrary specific string, e.g., `<script>`. The pattern reads as follows: Starting with any sequence of characters, we match words that end with two strings `u` separated by i to j characters and followed by another k to l characters until the end of the line.

This is one of many examples of such regular patterns. For the purpose of experimental evaluation, we simplify the pattern as follows: `#.*a.{i}$` where `a` stands for a single symbol and `{ i }` for a fixed number i of repetitions. The pattern reads as: match any word that has symbol `a` i symbols away from the end of the word.

Usually, the integers for repetitions in pattern matching are parametric, giving us a parametric benchmark SymbolFromEnd with varying numbers of repetitions. We have limited our benchmark to numbers i between 1 and 150 We construct $\mathcal{T}_{x^{+}_{\pi\to y}}$ and $\mathcal{T}_{x^{1}_{\pi\to y}}$ for the regular pattern `#.*a.{i}` replacing the pattern with an empty string. In total, we create 300 instances in SymbolFromEnd benchmark.

We compare the performance of Mata and Mona on operations application on a language (applying $\mathcal{T}_{x^{+}_{\pi\to y}}$ and $\mathcal{T}_{x^{1}_{\pi\to y}}$ on a language of the matched pattern), and composition constructing the resulting NFT for replace operation.

Rationale for chosen benchmarks. All of these are the operations which will often be used on NFTs created to replace operations in Z3-Noodler and will make up the main set of operations on NFTs once support for transducers is introduced in Z3-Noodler as well. Furthermore, these are the operations usually performed in any general use of transducers, since they comprise all the transducer-specific operations one needs when working with transducers. The benchmarks TRANSDUCERPLUS and WEBAPP from runs of Z3-NOODLER are practical, but very simple benchmarks with small regular patterns, which is however the first planned application of NFTs in MATA. The benchmark SYMBOLFROMEND created from pattern matching benchmarks is practical, and produces a complex problem instances. The benchmark represents what happens when NFTs are applied to complex `replaceAll` problems and problems from other domains than the simplest string solving examples in SMT-LIB.

Note that since self-loops cannot be easily added to an existing NFT in MONA, for all operations using composition, we extend the NFTs from benchmarks with additional tapes to synchronize the tapes in both NFTs and add self-loops manually before measuring the runtimes of operations.

The generated benchmarks are accessible in a GitHub repository[1].

7.2 Results

The experiments were run on GNU/Linux system Ubuntu 22.04.4 LTS (with the Linux kernel GNU/Linux 5.15.0-106-generic x86_64, CPU AMD Ryzen 5 5600G @ 3.892GHz) with timeout set to 120 seconds. Due to small runtimes on the majority of instances (a few milliseconds and less), each benchmark instance was run 3 times and the resulting runtime is computed as the mean of the 3 runs, which mitigates unexpected runtime deviations caused by waiting for the assignment of computation unit.

The experimental pipeline and all the raw results of run experiments are accessible in a GitHub repository[2].

The scatter plots show the comparison between MATA and MONA. The axis are linear with runtimes in milliseconds. The horizontal and vertical dotted lines represent the timeouts. For graphs with limited range on the axis to more clearly see the comparison on the majority of the results, all results beyond the displayed area are depicted as timeouts on the dotted lines. When constructing the tables, after computing the number of timeouts for each tool, the instances with timeouts are dropped from the results. Therefore, the tables show runtimes for instances where both tools finished. All tables in this chapter show the tool used; number of timeouts; and runtimes: minimal, maximal, mean, quantile 0.25, median (quantile 0.50), quantile 0.75, and standard deviation. The times shown in the tables are in milliseconds.

7.2.1 Projection

We show the performance of projection on a 2-tape NFTs on the input tape (projecting out the output tape), and the output tape (projecting out the input tape) separately, since in 2-tape NFTs, both cases project the first or the last tape, which requires a more complex modification of the NFT than projection of a tape in the middle between another tapes. This results in different runtimes for each projection.

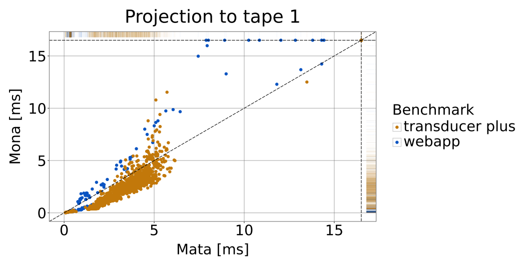
Table 7.1 shows comparison of performance of MATA and MONA on the projection operation on both benchmarks.

The figures in Figure 7.1 show the same comparison between MATA and MONA with scatter plots: Figure 7.1a shows the projection to tape 1 (projecting out the tape 0) and Figure 7.1b the projection to tape 0 (projecting out the tape 1).

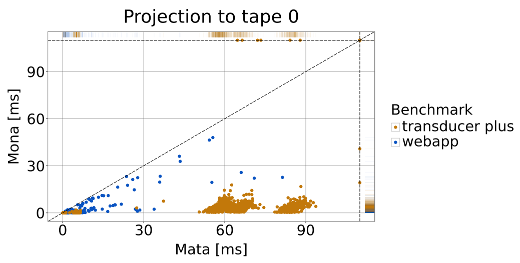
[1] `https://github.com/Adda0/smt-lib-replace-transducers-benchmarks`
[2] `https://github.com/Adda0/Transducers-in-Automata-Library-Mata_experiments`

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata-0	0	0.07	321.07	3.22	1.39	2.70	3.92	11.40
mona-0	0	0.02	2846.18	6.47	0.40	1.58	2.83	98.03
mata-1	0	0.10	2894.50	53.88	7.43	58.67	67.35	100.94
mona-1	0	0.03	3189.42	7.84	0.47	2.23	4.08	109.72

Table 7.1: Projection on both benchmarks. $-i$ after the tool name specifies which tape was projected out.



(a) Projection to tape 1 (projecting out the tape 0).



(b) Projection to tape 0 (projecting out the tape 1).

Figure 7.1: Scatter plots comparing performance of Mata and Mona on operation projection.

We can see that all projections to tape 1 finish in under 350 ms in Mata. By comparing means, we can see that the Mata is about 2 times slower than Mona, but is still fast, finishing 75% of executions in under 4 ms, compared to Mona's runtime under 3 ms.

On the other hand, projecting to tape 0, i.e., projecting out the tape 1 is more expensive operation for Mata in the implemented data structure. However, even than Mata finishes 0.75% of executions in under 70 ms. Comparing to Mona, encoding of NFTs in Mona ensures that the projecting out the tape 1 is similarly expensive to projecting out the tape 0. If the performance of Mata on projection to tape 0 proves insufficient when support for NFTs is added in Z3-Noodler, one can omit projecting synchronization levels after every composition (and keep the synchronization levels inside the NFT), or apply the suggested specialization for composition for 2-tape NFTs which does not need to perform projection at all.

7.2.2 Application

For each unique NFT $\mathcal{T}$, we applied $\mathcal{T}$ on a word and on a language using NFT application.

Table 7.2 shows comparison of performance of Mata and Mona on the application operation on both benchmarks applying $\mathcal{T}$ on a language Σ^*. The NFT is forward applied on the language $(\mathcal{T}_{\text{Id}(\Sigma^*)}||\mathcal{T})$. This simulates the forward image propagation, as described in 6.5.

Table 7.3 shows comparison of performance of Mata and Mona on the application operation on both benchmarks applying $\mathcal{T}$ on a language Σ^*. The NFT is backward applied on the language $(\mathcal{T}||\mathcal{T}_{\text{Id}(\Sigma^*)})$. This simulates the backward image propagation, as described in 6.5.

Table 7.4 shows comparison of performance of Mata and Mona on the application operation on both benchmarks applying $\mathcal{T}$ on a literal.

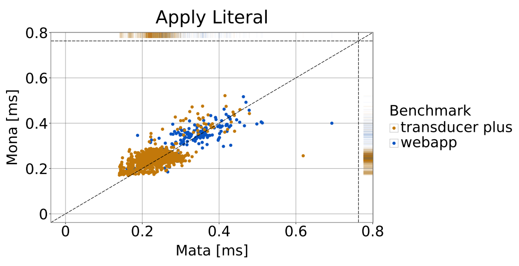
method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	1	0.07	4.77	2.65	0.91	3.26	3.63	1.41
mona	1	0.05	1.97	0.89	0.39	1.05	1.15	0.47

Table 7.2: Language application on both benchmarks where $\mathcal{T}$ is forward applied on the language $(\mathcal{T}_{\mathrm{Id}(\Sigma^*)}||\mathcal{T})$, simulating the forward image propagation.

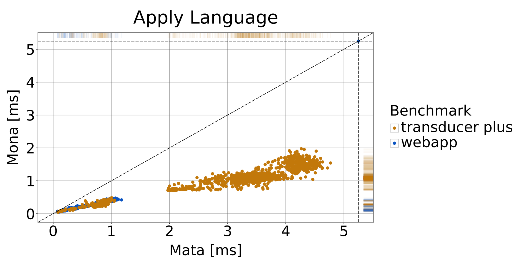
method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	5	0.20	98138.12	436.31	5.95	31.09	50.79	4598.41
mona	5	0.05	8186.49	45.42	1.18	17.19	46.31	298.65

Table 7.3: Language application on both benchmarks where $\mathcal{T}$ is backward applied on the language $(\mathcal{T}||\mathrm{Id}(\mathcal{L}(\pi)))$, simulating backward image propagation.

Table 7.5 shows comparison of performance of Mata and Mona on the application operation on both benchmarks backward applying $\mathcal{T}$ on a literal.



(a) Application on a literal.



(b) Application on a language.

Figure 7.2: Scatter plots for comparing the performance of Mata and Mona on forward application operation on both benchmarks.

From the results in Table 7.4 and Table 7.2 and scatter plots in Figures 7.2 and 7.3, we can see that applying $\mathcal{T}$ on a fairly constrained language (in our case directly a literal) is a quick operation where Mata performs comparably with Mona. However, applying $\mathcal{T}$ on a large language such as Σ^* (used in our experiments), the application gets more expensive for Mata, but even then Mata runs in under 4 ms on the forward application benchmark.

This application on Σ^* is actually the worst case scenario for Mata, as with Σ^*, Mata has to apply all transition symbols $a \in \Sigma$ for each state $s \in Q$ of $\mathcal{T}$. Nevertheless, the computation for both the forward and backward application operations for both the literal and the language finishes in 75% cases in at most 50 ms, which should be fast enough for Z3-Noodler, especially considering the results of the applications and computations can be cached and reused during a run of Z3-Noodler. Notice that for application on a literal, Mata is about 2 to 3 times slower, however, when the language gets larger (Σ^* in our case), the performance on the 3rd quartile is comparable to Mona on the backward application on the language, and only 2 times slower on the forward application on the language. We can see that the backward application is more difficult for both libraries. When backward applying $\mathcal{T}$ on a literal, Mata times out on a few instances. This calls for a close inspection of the particular instances. If we find a reason for this slowdown, subsequent optimization of the data structures and algorithms in Mata according to the results should be possible. Considering our future uses for application operation in Z3-Noodler, we will sometimes

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	0	0.14	0.69	0.24	0.22	0.24	0.26	0.05
mona	0	0.17	0.52	0.25	0.24	0.25	0.26	0.05

Table 7.4: Literal application on both benchmarks.

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	16	0.08	51476.79	554.98	3.65	19.41	39.40	4282.76
mona	1	0.05	11863.97	79.63	1.44	11.07	32.10	707.64

Table 7.5: Backward literal application on both benchmarks.

apply on the whole language Σ^*, but more often we will apply on a more constrained subset $L \subset \Sigma^*$ which should therefore perform better on Z3-Noodler runs. If the performance is then still insufficient, we can implement the aforementioned optimization for composition for 2-tape NFTs and use in Z3-Noodler this version instead of the general application implementation for multi-tape NFTs.

7.2.3 Composition

Table 7.6 shows comparison of performance of Mata and Mona on the composition operation on both benchmarks.

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	15	0.07	54411.24	434.64	3.20	4.16	28.52	3977.53
mona	0	0.05	10476.19	51.04	1.26	1.93	20.97	542.09

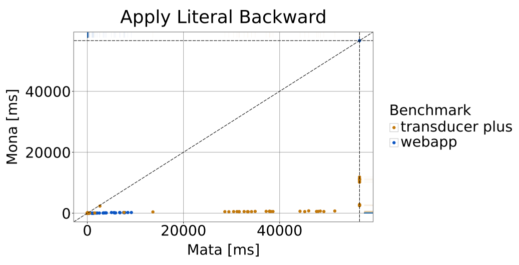
Table 7.6: Composition on both benchmarks.

Table 7.7 shows comparison of performance of Mata and Mona on the composition operation constructing NFTs for replace operations on both benchmarks.

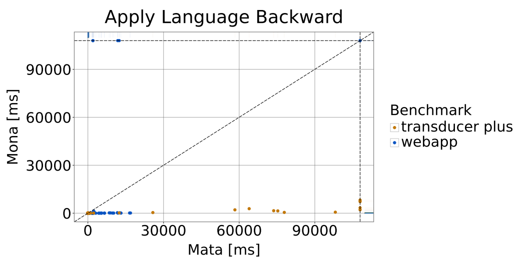
We can see that performance of Mata is comparable to the performance of Mona on numerous instances for composition constructing replace NFTs, but there is also a similarly large group of instances in the TransducerPlus benchmark where Mata runs slower. Mata also has 15 timeouts on the largest compositions created as partial compositions of a long sequence of compositions. In general, the runtime of Mata is sufficient. We can see that 75% of instances common in string solving are solved in less than 28 ms in Mata compared to 21 ms in Mona. Only the partial compositions containing considerably larger NFTs show noticeable slowdown for Mata. Composition creating the replace NFTs runs much faster, and Mata is comparable to Mona on many instances, even if slower on other instances. The runtime for 75% of cases is under 9 ms, about two times higher then for Mona. As mentioned earlier, if the performance of NFTs in Mata proves to be a bottleneck in string solving, a specialized algorithm for composition of 2-tape NFTs can be used instead of the general algorithm for multi-tape NFTs.

7.2.4 Pattern matching benchmark.

The Table 7.8 shows the comparison of Mata and Mona on pattern matching benchmark SymbolFromEnd on application on the language operation.



(a) Application on a literal.



(b) Application on a language

Figure 7.3: Scatter plots for comparing the performance of Mata and Mona on backward application operations on both benchmarks.

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	0	0.15	25.52	6.35	2.48	7.89	8.89	3.49
mona	0	0.18	25.44	2.59	1.01	2.73	3.52	1.81

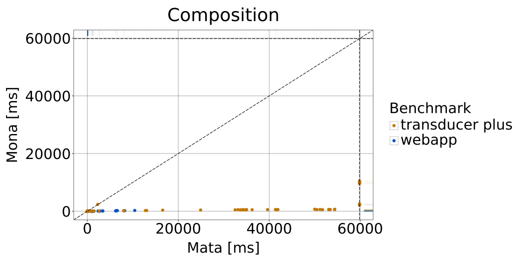
Table 7.7: Composition constructing NFTs for replace operations on both benchmarks.

The Table 7.9 shows the comparison of Mata and Mona on pattern matching benchmark SymbolFromEnd on the construction of the replace NFT operation.

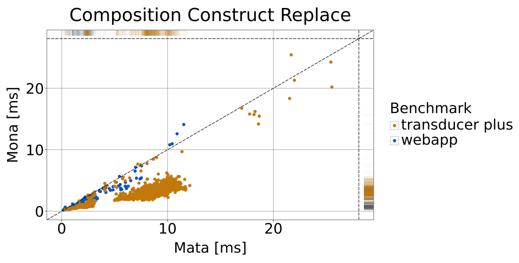
Mata computed all pattern matching benchmarks quickly, running at most 8 s on the largest instance with 150 repetitions on the application on the language operation. Mona managed to compute only the first smallest 14 instances (for both the replace single and replace all variants, in total finishing 28 on instances). In the remaining 272 instances, Mona fails to determinize the pattern and runs out of memory. The similar results can be seen in construction of replace NFTs. Even if the intermediate NFTs are small compared to the final NFTs for replace operations, Mona manages to compute only the first smallest 35 instances (for both the replace single and replace all variants, in total finishing on 70 instances). In the remaining 232 instances, Mona fails to determinize the pattern and runs out of memory. The tables show that on the smallest instances, Mona runs faster, but quickly starts timing out when the lack of support for nondeterminism starts being an issue for Mona. This benchmark clearly shows that once you start using NFTs on problems from practice more complex than the simple replace operations in string solving benchmarks from SMT-LIB, the ability to use explicit symbols on transitions and handling nondeterminism becomes crucial to all tools applicable to these problem domains.

7.3 Summary

For the string solving benchmarks, the experiments show that Mata has higher base runtime in comparison with Mona, but in general, 75% of examples are finished in about the same time in Mata as in Mona, finishing in under 50 ms. Mata runs comparably with Mona on application on a literal and projection to tape 1. Otherwise, Mata runs slower than Mona, but even then its runtime is reasonable. There are a few instances where Mata fails to compute the composition and times out whereas Mona succeeds in computing the composition. Considering that Mona is a well-optimized library for applications on binary representations of transition symbols, and the experiments are run on the extreme cases (application on Σ^*) of the operations performed in string solving, which are the most un-



(a) Composition operation.



(b) Composition constructing replace operations.

Figure 7.4: Scatter plots for comparing the performance of Mata and Mona on composition operations on both benchmarks.

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	0	0.97	8070.18	2397.81	461.98	1743.10	4035.48	2233.84
mona	272	0.29	13095.06	1614.42	2.39	39.25	970.33	3485.11

Table 7.8: Application on language on SymbolFromEnd.

favourable cases for Mata, we consider a success that NFTs in Mata are able to get close to the performance of Mona, and even outperform Mona in some cases. For most cases, the performance of Mata should be improved by implementing the optimized composition specialized for 2-tape NFTs which omits having to insert self-loops and additional levels into the composed NFTs.

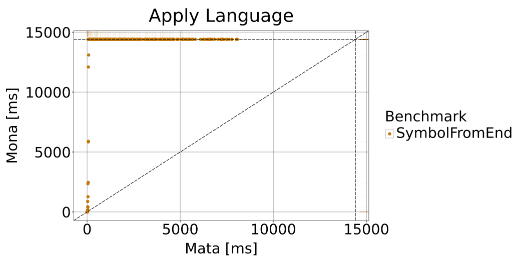
The experiments shown are not the only use cases for NFTs in Mata, but they represent a selection of operations that will be performed in Z3-Noodler. Furthermore, the instances of replace operations appearing in SMT-LIB benchmarks are relatively small and easy to compute. When applying NFTs to examples from practice from other domains, e.g., pattern matching [50], the patterns being matched get progressively larger and introduce more nondeterminism. With the larger patterns, the advantage of Mata being able to handle nondeterminism naturally proves to be crucial for efficient creation and handling the corresponding NFTs, quickly outperforming tools without proper support for nondeterminism. That is because the complexity of determinization of similar patterns usually increases exponentially to the numbers in counters.

Due to how close are the representations of NFTs in Mata and Mona, both libraries perform similar operations, only Mona on deterministic versions of these NFTs. Either Mona implements more performant data structures than `Delta` for NFAs in Mata, or Mona uses some algorithmic heuristics unknown to us at the moment, which should be however applicable to Mata if we find out about them. We believe that the fast performance of Mona on small replace operations from string solving shows that there is still space for optimizations of the performance of data structures and low-level operations used in Mata. We noticed similar problems with noticeable slowdowns on various NFA benchmarks in our earlier work on Mata in article [29]. Mata should be able to get even closer to the performance of Mona after thorough optimization of the data structures in Mata. Since this is the initial implementation of NFTs, we plan to further thoroughly analyse and optimize the operations used for both NFAs and NFTs.

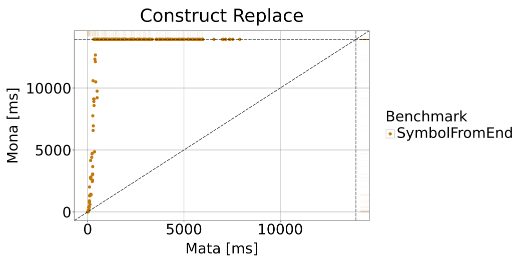
We conclude that NFTs in Mata are a competitive implementation of NFTs, are prepared and ready to be used in Z3-Noodler, and the experimental evaluation shows that

method	TOs	min	max	mean	q(0.25)	median	q(0.75)	std. dev
mata	0	1.79	7900.85	2211.33	446.27	1623.44	3680.46	1983.68
mona	232	1.04	12664.70	2987.85	79.46	1304.86	4614.69	3791.32

Table 7.9: Composition for replace construction on SymbolFromEnd.



(a) Application on the language.



(b) Composition constructing replace operations.

Figure 7.5: Scatter plots for comparing the performance of Mata and Mona on SymbolFromEnd.

the performance of Mata should be sufficient for the uses in string solving. Moreover, we conclude that NFTs in Mata are capable of handling complex tasks with nondeterminism or large regular expressions, such as working with counting regular expressions, and are generally applicable to a large group of domains beside string solving.

Chapter 8

Conclusion

We have added support for a new model for finite state machines, finite transducers, to Mata, explained the principles behind NFTs and specifically NFTs in the context of Mata and Z3-Noodler for string solving. We have created a new data structure derived from the existing data structures for NFAs in Mata as NFAs with levels, which fit well in the overall structure and design decision of Mata. The introduced algorithms on NFTs are closely related to the corresponding BDD-like algorithms on BDDAs and maintain the two main goals of Mata: simplicity and efficiency, while keeping Mata easily extensible and understandable for a wide variety of users and applicable to both the areas of automata research and the industry. Thanks to these design decision, further extending Mata with support for handling BDDAs will be effortless. We further focussed on utilizing NFTs in string solving by adding algorithms to model string replace operations defined in SMT-LIB which will be used in Z3-Noodler for solving SMT formulae with string replace operations, previously unsolvable by Z3-Noodler.

We compared the performance of NFTs in Mata with a natural encoding of transducers in the automata library Mona. The experiments show that Mata performs slower than Mona in string solving, but still runs reasonably fast. When applied to complex regular expressions such as pattern matching, Mata clearly outperforms approaches without proper support for nondeterminism. The performance of Mata on 2-tape NFTs can be further improved by applying the specialized algorithm for composition for 2-tape NFTs.

Overall, the design and implementation of NFTs in Mata is feasible, functional, performs sufficiently enough, and is widely usable, with ready-for-use support for solving SMT formulae with string replace operations in Z3-Noodler.

Future work. We continue optimizing and developing Mata and the implementation of NFTs. In the future, we want to apply NFTs to solving string SMT formulae in Z3-Noodler. In addition, depending on the performance of NFTs in Z3-Noodler, we intend to add support for symbolic representations for large alphabets using bit vectors representing sequences of bits, as used in tool Lash [18] and transition representation specific to NFTs to encode NFT operations such as identity more easily. If the performance of composition of NFTs in Z3-Noodler proves to be a bottleneck, we can implement the optimized algorithm specialized for 2-tape NFTs. Since the representation of NFTs in Mata is close to BDDAs, we intend to study various optimizations used for BDDAs to further improve the performance of NFTs in Mata, leading to adding support for BDDAs in Mata.

Bibliography

[1] Abdulla, P. A.; Atig, M. F.; Chen, Y.; Diep, B. P.; Holík, L. et al. Flatten and conquer: A framework for efficient analysis of string constraints. In: Cohen, A. and Vechev, M. T., ed. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 2017, p. 602–617. Available at: https://doi.org/10.1145/3062341.3062384.

[2] Abdulla, P. A.; Atig, M. F.; Chen, Y.; Diep, B. P.; Holík, L. et al. Trau: SMT solver for string constraints. In: Bjørner, N. S. and Gurfinkel, A., ed. *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. IEEE, 2018, p. 1–5. Available at: https://doi.org/10.23919/FMCAD.2018.8602997.

[3] Abdulla, P. A.; Atig, M. F.; Chen, Y.; Holík, L.; Rezine, A. et al. String constraints for verification. In: Biere, A. and Bloem, R., ed. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Springer, 2014, vol. 8559, p. 150–166. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-319-08867-9_10.

[4] Abdulla, P. A.; Atig, M. F.; Chen, Y.; Holík, L.; Rezine, A. et al. Norn: An SMT solver for string constraints. In: Kroening, D. and Pasareanu, C. S., ed. *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 2015, vol. 9206, p. 462–469. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-319-21690-4_29.

[5] Abdulla, P. A.; Atig, M. F.; Diep, B. P.; Holík, L. and Janků, P. Chain-Free String Constraints. In: Chen, Y.; Cheng, C. and Esparza, J., ed. *Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings*. Springer, 2019, vol. 11781, p. 277–293. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-030-31784-3_16.

[6] Abdulla, P. A.; Bouajjani, A.; Holík, L.; Kaati, L. and Vojnar, T. Computing Simulations over Tree Automata. In: Ramakrishnan, C. R. and Rehof, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Springer, 2008, vol. 4963, p. 93–108. Lecture

Notes in Computer Science. Available at:
https://doi.org/10.1007/978-3-540-78800-3_8.

[7] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Pearson Education, Inc, 2006. ISBN 0-201-10088-6.

[8] Almeida, A.; Almeida, M.; Alves, J.; Moreira, N. and Reis, R. FAdo and GUItar: Tools for Automata Manipulation and Visualization. In: Maneth, S., ed. *Implementation and Application of Automata.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 65–74. ISBN 978-3-642-02979-0.

[9] Alt, L.; Blicha, M.; Hyvärinen, A. E. J. and Sharygina, N. SolCMC: Solidity compiler's model checker. In: Shoham, S. and Vizel, Y., ed. *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I.* Springer, 2022, vol. 13371, p. 325–338. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-031-13185-1_16.

[10] Backes, J.; Bolignano, P.; Cook, B.; Dodge, C.; Gacek, A. et al. Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N. S. and Gurfinkel, A., ed. *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018.* IEEE, 2018, p. 1–9. Available at: https://doi.org/10.23919/FMCAD.2018.8602994.

[11] Balzarotti, D.; Cova, M.; Felmetsger, V.; Jovanovic, N.; Kirda, E. et al. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In:. June 2008, vol. 0, p. 387–401. ISBN 978-0-7695-3168-7.

[12] Barbosa, H.; Barrett, C. W.; Brain, M.; Kremer, G.; Lachnitt, H. et al. Cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D. and Rosu, G., ed. *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I.* Springer, 2022, vol. 13243, p. 415–442. Lecture Notes in Computer Science. Available at:
https://doi.org/10.1007/978-3-030-99524-9_24.

[13] Barrett, C.; Fontaine, P. and Tinelli, C. *The Satisfiability Modulo Theories Library (SMT-LIB)* www.SMT-LIB.org. 2016.

[14] Berzish, M.; Day, J. D.; Ganesh, V.; Kulczynski, M.; Manea, F. et al. Towards More Efficient Methods for Solving Regular-expression Heavy String Constraints. *Theor. Comput. Sci.*, 2023, vol. 943, p. 50–72. Available at:
https://doi.org/10.1016/j.tcs.2022.12.009.

[15] Berzish, M.; Kulczynski, M.; Mora, F.; Manea, F.; Day, J. D. et al. An SMT solver for regular expressions and linear arithmetic over string length. In: Silva, A. and Leino, K. R. M., ed. *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II.* Springer, 2021, vol. 12760, p. 289–312. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-030-81688-9_14.

[16] Berzish, Murphy. *Z3str4: A Solver for Theories over Strings.* Dissertation. Available at: http://hdl.handle.net/10012/17102.

[17] Blahoudek, F.; Chen, Y.; Chocholatý, D.; Havlena, V.; Holík, L. et al. Word Equations in Synergy with Regular Constraints. In: Chechik, M.; Katoen, J. and Leucker, M., ed. *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings.* Springer, 2023, vol. 14000, p. 403–423. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-031-27481-7_23.

[18] Boigelot, B. and Latour, L. Counting the solutions of Presburger equations without enumerating them. *Theoretical Computer Science*, 2004, vol. 313, no. 1, p. 17–29. ISSN 0304-3975. Available at: https://www.sciencedirect.com/science/article/pii/S0304397503005322. Implementation and Application of Automata.

[19] Briggs, P. and Torczon, L. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery, mar 1993, vol. 2, 1–4, p. 59–69. ISSN 1057-4514. Available at: https://doi.org/10.1145/176454.176484.

[20] Brzozowski, J. A. Canonical regular expressions and minimal state graphs for definite events. In: *Proc. of Symposium on Mathematical Theory of Automata.* 1962.

[21] Cesare Tinelli, P. F. *SMT-LIB Theory of Strings* online. 2020. Available at: https://smt-lib.org/theories-UnicodeStrings.shtml.

[22] Chen, T.; Chen, Y.; Hague, M.; Lin, A. W. and Wu, Z. What is decidable about string constraints with the ReplaceAll function. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery, dec 2017, vol. 2, POPL. Available at: https://doi.org/10.1145/3158091.

[23] Chen, T.; Chen, Y.; Hague, M.; Lin, A. W. and Wu, Z. What is decidable about string constraints with the ReplaceAll function. *Proc. ACM Program. Lang.*, 2018, vol. 2, POPL, p. 3:1–3:29. Available at: https://doi.org/10.1145/3158091.

[24] Chen, T.; Flores-Lamas, A.; Hague, M.; Han, Z.; Hu, D. et al. Solving string constraints with Regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.*, 2022, vol. 6, POPL, p. 1–31. Available at: https://doi.org/10.1145/3498707.

[25] Chen, T.; Hague, M.; He, J.; Hu, D.; Lin, A. W. et al. A decision procedure for path feasibility of string manipulating programs with integer data type. In: Hung, D. V. and Sokolsky, O., ed. *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings.* Springer, 2020, vol. 12302, p. 325–342. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-030-59152-6_18.

[26] Chen, T.; Hague, M.; Lin, A. W.; Rümmer, P. and Wu, Z. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.*, 2019, vol. 3, POPL, p. 49:1–49:30. Available at: https://doi.org/10.1145/3290362.

[27] Chen, Y.; Chocholatý, D.; Havlena, V.; Holík, L.; Lengál, O. et al. Solving String Constraints with Lengths by Stabilization. *Proc. ACM Program. Lang.*, 2023, vol. 7, OOPSLA2, p. 2112–2141. Available at: https://doi.org/10.1145/3622872.

[28] Chen, Y.-F.; Chocholatý, D.; Havlena, V.; Holík, L.; Lengál, O. et al. Z3-Noodler: An Automata-based String Solver. In: Finkbeiner, B. and Kovács, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Cham: Springer Nature Switzerland, 2024, p. 24–33. ISBN 978-3-031-57246-3.

[29] Chocholatý, D.; Fiedor, T.; Havlena, V.; Holík, L.; Hruška, M. et al. Mata: A Fast and Simple Finite Automata Library. In: Finkbeiner, B. and Kovács, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Cham: Springer Nature Switzerland, 2024, p. 130–151. ISBN 978-3-031-57249-4.

[30] CWE. *2022 CWE Top 25 Most Dangerous Software Weaknesses.* 2023. Available at: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html#analysis.

[31] CWE. *2023 CWE Top 25 Most Dangerous Software Weaknesses.* 2023. Available at: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[32] D'Antoni, L. and Veanes, M. Minimization of Symbolic Automata. In: *Proc. of POPL'14.* ACM, 2014. ISBN 9781450325448.

[33] D'Antoni, L. and Veanes, M. Minimization of Symbolic Tree Automata. In: *Proc. of LICS'16.* ACM, 2016.

[34] Doyen, L. and Raskin, J. Antichain Algorithms for Finite Automata. In: *Proc. of TACAS'10.* Springer, 2010. LNCS.

[35] D'Souza, D. and Shankar, P., ed. *Modern Applications of Automata Theory.* World Scientific, 2012. IISc Research Monographs Series. ISBN 978-981-4271-04-2. Available at: https://doi.org/10.1142/7237.

[36] Duret Lutz, A.; Renault, E.; Colange, M.; Renkin, F.; Gbaguidi Aisse, A. et al. From Spot 2.0 to Spot 2.10: What's New? In: Shoham, S. and Vizel, Y., ed. *Computer Aided Verification.* Cham: Springer International Publishing, 2022, p. 174–187. ISBN 978-3-031-13188-2.

[37] Eriksson, B.; Stjerna, A.; Masellis, R. D.; Rümmer, P. and Sabelfeld, A. Black Ostrich: Web Application Scanning with String Solvers. In: *CSS'23.* ACM, 2023, p. 549–563. Available at: https://doi.org/10.1145/3576915.3616582.

[38] Esparza, J. *Automata Theory: An Algorithmic Approach* online. 2017. Available at: https://www7.in.tum.de/~esparza/automatanotes.html. [cit. 2020-10-31].

[39] Evans, C. *Automata.* 2023. Available at: https://github.com/caleb531/automata.

[40] Fiedor, T.; Holík, L.; Hruska, M.; Rogalewicz, A.; Síc, J. et al. Reasoning About Regular Properties: A Comparative Study. In: Pientka, B. and Tinelli, C., ed. *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings.* Springer, 2023, vol. 14132, p. 286–306. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-031-38499-8_17.

[41] Fu, X. and Li, C. Modeling Regular Replacement for String Constraint Solving. In: Muñoz, C. A., ed. *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings.* 2010, NASA/CP-2010-216215, p. 67–76. NASA Conference Proceedings.

[42] Fu, X.; Powell, M.; Bantegui, M. and Li, C.-C. Simple linear string constraints. *Formal Aspects of Computing*, november 2013, vol. 25.

[43] group, S.-C. *SMT-COMP International Satisfiability Modulo Theories Competition* online. 2024. Available at: https://smt-comp.github.io.

[44] Henriksen, J. G.; Jensen, J. L.; Jørgensen, M. E.; Klarlund, N.; Paige, R. et al. Mona: Monadic Second-Order Logic in Practice. In: *TACAS '95.* Springer, 1995, vol. 1019, p. 89–110. LNCS.

[45] Henzinger, M.; Henzinger, T. and Kopke, P. Computing simulations on finite and infinite graphs. In: *Proceedings of IEEE 36th Annual Foundations of Computer Science.* 1995, p. 453–462.

[46] Holík, L.; Janků, P.; Lin, A. W.; Rümmer, P. and Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*, 2018, vol. 2, POPL, p. 4:1–4:32. Available at: https://doi.org/10.1145/3158092.

[47] Holík, L.; Janku, P.; Lin, A. W.; Rümmer, P. and Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*, 2018, vol. 2, POPL, p. 4:1–4:32. Available at: https://doi.org/10.1145/3158092.

[48] Holik, L.; Janků, P.; Lin, A.; Rümmer, P. and Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages*, december 2017, vol. 2, p. 1–32.

[49] Holík, L.; Lengál, O.; Sí č, J.; Veanes, M. and Vojnar, T. Simulation Algorithms for Symbolic Automata. In: Lahiri, S. K. and Wang, C., ed. *Proc. of ATVA'18.* Springer, 2018. ISBN 978-3-030-01090-4.

[50] Holík, L.; Síč, J.; Turoňová, L. and Vojnar, T. Fast Matching of Regular Patterns with Synchronizing Counting. In: Kupferman, O. and Sobocinski, P., ed. *Foundations of Software Science and Computation Structures.* Cham: Springer Nature Switzerland, 2023, p. 392–412. ISBN 978-3-031-30829-1.

[51] Holík, L. and Šimáček, J. Optimizing an LTS-Simulation Algorithm. In: *MEMICS'09.* Faculty of Informatics MU, 2009, p. 93–101.

[52] Hooimeijer, P.; Livshits, B.; Molnar, D.; Saxena, P. and Veanes, M. Fast and Precise Sanitizer Analysis with BEK. In: *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings.* USENIX Association, 2011. Available at: http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf.

[53] Hopcroft, J. E. *An n Log n Algorithm for Minimizing States in a Finite Automaton.* Stanford, CA, USA: Stanford University, 1971.

[54] ISBERNER, M.; HOWAR, F. and STEFFEN, B. *AutomataLib*. Available at: https://learnlib.de/projects/automatalib/.

[55] KAPLAN, R. and KAY, M. Regular Models of Phonological Rule Systems. *Computational Linguistics*, january 1994, vol. 20, p. 331–378.

[56] KARTTUNEN, L.; CHANOD, J.-P.; GREFENSTETTE, G. and SCHILLER, A. Regular Expressions for Language Engineering. *Natural Language Engineering*, march 1997, vol. 2.

[57] KELB, P.; MARGARIA, T.; MENDLER, M. and GSOTTBERGER, C. MOSEL: A Sound and Efficient Tool for M2L(Str). In: GRUMBERG, O., ed. *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*. Springer, 1997, vol. 1254, p. 448–451. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/3-540-63166-6_45.

[58] KERN, C. Securing the tangled web. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery, September 2014, vol. 57, no. 9, p. 38–47. ISSN 0001-0782. Available at: https://doi.org/10.1145/2643134.

[59] KŘETÍNSKÝ, J.; MEGGENDORFER, T. and SICKERT, S. Owl: A Library for ω-Words, Automata, and LTL. In: LAHIRI, S. K. and WANG, C., ed. *Automated Technology for Verification and Analysis*. Cham: Springer International Publishing, 2018, p. 543–550. ISBN 978-3-030-01090-4.

[60] LENGÁL, O.; EK, J. Š. and VOJNAR, T. VATA: A Library for Efficient Manipulation of Non-deterministic Tree Automata. In: *Proc. of TACAS'12*. Springer, 2012, vol. 7214. LNCS.

[61] LIANA HADAREAN. *String Solving at Amazon* https://mosca19.github.io/program/index.html. 2019. Presented at MOSCA'19.

[62] LIN, A. W. and BARCELÓ, P. String solving with word equations and transducers: Towards a logic for analysing mutation XSS. In: BODÍK, R. and MAJUMDAR, R., ed. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016, p. 123–136. Available at: https://doi.org/10.1145/2837614.2837641.

[63] LOMBARDY, S.; MARSAULT, V. and SAKAROVITCH, J. *Awali, a library for weighted automata and transducers (version 2.0)*. 2021. Software available at http://vaucanson-project.org/Awali/2.0/.

[64] MICROSOFT. *Azure Resource Manager documentation*. 2020. Available at: https://docs.microsoft.com/en-us/azure/azure-resource-manager/.

[65] MOHRI, M. Finite-State Transducers in Language and Speech Processing. *Comput. Linguistics*, 1997, vol. 23, no. 2, p. 269–311.

[66] MOURA, L. de and BJØRNER, N. Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM*, september 2011, vol. 54, p. 69–77.

[67] Moura, L. M. de and Bjørner, N. S. Z3: An efficient SMT solver. In: Ramakrishnan, C. R. and Rehof, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* Springer, 2008, vol. 4963, p. 337–340. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-540-78800-3_24.

[68] Møller, A. et al. *Brics automata library.* Available at: https://www.brics.dk/automaton/.

[69] Negrini, L.; Arceri, V.; Cortesi, A. and Ferrara, P. Tarsis: An effective automata-based abstract domain for string analysis. *Journal of Software: Evolution and Process.* Wiley Online Library, 2024, p. e2647. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2647.

[70] Nielsen, J. Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. *Mathematische Annalen.* Springer, 1917, vol. 78, no. 1, p. 385–397.

[71] Nieuwenhuis, R.; Oliveras, A. and Tinelli, C. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *J. ACM.* ACM New York, NY, USA, 2006, vol. 53, no. 6, p. 937–977. ISSN 0004-5411. Available at: https://doi.org/10.1145/1217856.1217859.

[72] OWASP. *Top 10* https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf. 2013.

[73] OWASP. *Top 10* https://owasp.org/www-project-top-ten/2017/. 2017.

[74] OWASP. *Top 10* https://owasp.org/Top10/. 2021.

[75] Ranzato, F. and Tapparo, F. An efficient simulation algorithm based on abstract interpretation. *Information and Computation*, 2010, vol. 208, p. 1–22.

[76] Rungta, N. A billion SMT queries a day (invited paper). In: Shoham, S. and Vizel, Y., ed. *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I.* Springer, 2022, vol. 13371, p. 3–18. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-031-13185-1_1.

[77] Seidl, H.; Schwentick, T.; Muscholl, A. and Habermehl, P. Counting in Trees for Free. In: Díaz, J.; Karhumäki, J.; Lepistö, A. and Sannella, D., ed. *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings.* Springer, 2004, vol. 3142, p. 1136–1149. Lecture Notes in Computer Science. Available at: https://doi.org/10.1007/978-3-540-27836-8_94.

[78] Sipser, M. *Introduction to the Theory of Computation.* 3rdth ed. Cengage Learning, 2013. ISBN 13: 978-1-133-18779-0.

[79] Sun, K. and Ryu, S. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Com. Surv.* New York, NY, USA: Association for Computing

Machinery, 2017, vol. 50, no. 4. ISSN 0360-0300. Available at: https://doi.org/10.1145/3106741.

[80] Thái Minh, T.; Chu, D.-H. and Jaffar, J. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. *Proceedings of the ACM Conference on Computer and Communications Security*, november 2014, p. 1232–1243.

[81] Veanes, M. *A .NET automata library*. Available at: https://github.com/AutomataDotNet/Automata.

[82] Veanes, M.; Halleux, P. de and Tillmann, N. Rex: Symbolic Regular Expression Explorer. In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010, p. 498–507. Available at: https://doi.org/10.1109/ICST.2010.15.

[83] Watson, B. W. Implementing and using finite automata toolkits. *Nat. Lang. Eng.*, 1996, vol. 2, no. 4, p. 295–302. Available at: https://doi.org/10.1017/S135132499700154X.

[84] Wolper, P. and Boigelot, B. On the Construction of Automata from Linear Arithmetic Constraints. In: Graf, S. and Schwartzbach, M., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 1–19. ISBN 978-3-540-46419-8.

[85] Zheng, Y.; Zhang, X. and Ganesh, V. Z3-str: a z3-based string solver for web application analysis, august 2013.

Appendix A

Contents of the Storage Media

The following list lists the contents of the included storage media. Listed are only the few top-level folders in the folder hierarchy.

- `mata/`: The main folder containing the source code for Mata automata library, with the reference implementation of finite transducers.
- `docs/`: The LaTeX source files for this work.
- `benchmarks/`: The benchmarks generated from runs of Z3-Noodler on SMT-LIB benchmarks, used in comparison of Mata with Mona.
- `experiments/`: The experimental pipeline with the source code for running experiments comparing Mata with Mona, source code for Mona.
  - `results/raw/`: The storage of raw generated CSV files when running experiments.
  - `results/processed/`: The CSV files with appropriate names prepared for analysis for experiments shown in this work.
  - `analysis/`: The scripts for analysing run experiments (generate graphs and tables shown in this work).
    - `plots/`: The graphs, tables and plots generated from analysis of the run experiments shown in this work.

Appendix B

Reference Implementation Manual

Here, we describe how our reference implementation for transducers in Mata can be run, tested, and how one can reproduce the experiments shown in this work.

The experiments shown in this work were run on GNU/Linux system Ubuntu 22.04.4 LTS (with the Linux kernel GNU/Linux 5.15.0-106-generic x86_64). The experiments should be runnable on any Unix-like system, provided the following requirements are met.

The following programs are required for the implementation and experimental pipeline to run:

- Python 3.10.12 or higher,
- C++ compiler, experiments run on g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0,
- cmake 3.22.1 or higher,

The experiments can be run as follows:

- Compile and install Mata with finite transducers:
  ```
  make -C mata release
  sudo make -C mata install
  ```
- Compile programs in experimental pipeline:
  ```
  cd experiments/
  make -C mona
  make -C mata
  ```
- Run an experiment:
  ```
  ./run_on_benchmark.py -runs <RUNS> -timeout <TIMEOUT> <OPERATION> <PATH>
  ```
  where `<RUNS>` is to be replaced with the number of runs on each benchmark instance should be run; `<TIMEOUT>` is the requested timeout for each benchmark instance in seconds; `<OPERATION>` is the benchmark operation to run; and `<PATH>` is the path to a file containing a benchmark instance in `.mata` format, or a folder containing (possibly inside additional subfolders) files with the benchmark instances in `.mata` format to run the experiments on.
- Run all experiments from this work:
  ```
  ./run_all_experiments.sh
  ```
- See generated CSV files with results in `results/raw/`.
  Copy the files over to `results/processed/` with descriptive names.

- Analyse the results:
  ```
  cd experiments/analyse
  python -m venv .venv
  source .venv/bin/activate
  pip install -r requirements.txt
  ```
  `analyse.py` where `analyse.py` expects the following CSV files with results in `results/processed/`:

  - `results/processed/<BENCHMARK>_apply_language.csv`
  - `results/processed/<BENCHMARK>_apply_literal.csv`
  - `results/processed/<BENCHMARK>_apply_language_backward.csv`
  - `results/processed/<BENCHMARK>_apply_literal_backward.csv`
  - `results/processed/<BENCHMARK>_projection.csv`
  - `results/processed/<BENCHAMRK>_composition.csv`
  - `results/processed/<BENCHAMRK>_composition_construct_replace.csv`

  where `<BENCHMARK>` is replaced with `transducer-plus` and `webapp`.

  Further, results for benchmark SymbolFromEnd are expected:

  - `results/processed/symbol_from_end_apply_language.csv`
  - `results/processed/symbol_from_end_construct_replace.csv`