TRANSLATION OF MSOL FORMULAS TO FINITE AUTOMATONS

Jarmila Fialová

Bachelor's thesis

Study program: Informatics (Czech)
Specialisation: Theorethical Informatics
Supervisor: RNDr. Radek Hušek Ph.D.

Department of Theorethical Computer Science
Faculty of Information Technology
Czech Technical University in Prague

May 16, 2025

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Překlad MSOL formulí na konečné automaty
Student:	Jarmila Fialová
Vedoucí:	RNDr. Radek Hušek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika 2021
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2025/2026

Pokyny pro vypracování

Tato práce je prvním krokem k moderní reimplementaci programu MONA [1], který překládá formule monadické logiky druhého řádu na konečné automaty ověřující, zda daný řetězec formuli splňuje. (MONA podporuje i logiku se dvěma následníky vedoucí na stromové automaty, těmi se však práce zabývat nebude.)

Hlavní praktickou limitací MONA je neschopnost pracovat s více než 16 miliony uzlů v BDD (binary decision diagram) reprezentujících jeden konečný automat. Implementace je mixem C a C++ z 90. let a obsahuje značné množství maker a globálních proměnných, takže tento limit není snadné odstranit.

Nová implementace bude po vzoru překladačů programovacích jazyků rozdělena na front-end zpracovávající vstup v jazyce MONA, middle-end provádějící optimalizace nad vzniklou mezireprezentací a back-end, který bude konstruovat automaty dle instrukcí middle-endu. Na rozdíl od překladačů budou tyto části výrazněji odděleny (budou spustitelné samostatně), jelikož na ně jsou kladeny velmi rozdílné požadavky a především back-end může potřebovat značné množství výpočetních prostředků.

Cíle této práce:
- Prozkoumat a popsat současné fungování MONA včetně užitých algoritmů a rozdělit ho na front, middle a back-end.
- Prozkoumat a popsat případné alternativy k těmto algoritmům.
- Navrhnout a implementovat prototyp back-endu.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 8. února 2025 v Praze.

[1] https://www.brics.dk/mona/index.html

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 8. února 2025 v Praze.

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Fialová Jarmila. *Translation of MSOL formulas to finite automatons.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

First off, I'd like to thank my absolute beast of a supervisor, RNDr. Radek Hušek Ph.D., for infinite support during both the creation of this here thesis and my academic journey in all its length. You were there for me during the highest of hights and lowest of lows, and I wouldn't have it any other way. I sincerely could not have asked for a better supervisor, and I hope I was a good test subject.

I'd like to thank the other members of the Department of Theoretical Computer Science at FIT CTU, for accepting me into their midst, and tolerating my frequent appropriation of the couch and sleeping during any and all classes, lectures, and the like.

Thank you to all the members of the Faculty of Information Technology at Czech Technical University. The community of both students and teachers is amazing, and for the first time in my life I truly feel like I belong somewhere. I'd also like to thank my family, friends and various communities who have supported me throughout this journey, in ways big and small. Even if you may never read these words of gratitude, I still send them out. In particular: Praise Pravus!

Finally, I'd like to thank you, the reader, for giving this a shot. You reading this thesis now warms my heart with joy that the countless hours poured into this thesis have not been wasted for naught. Thank you.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I declare that I have not used any AI tools during the preparation and writing of the thesis. I am aware of the consequences of apparently unacknowledged use of these tools in the production of any part of my thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act. In accordance with Section 2373(2) of Act No. 89/2012 Coll., Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to use this copyright work, including all computer programs and all their documentation (hereinafter collectively referred to as "the Work"), to all persons who wish to use the Work. Such persons shall be entitled to use the Work in any manner that does not diminish the value of the Work, and for any purpose (including use for profit) but must preserve the validity of the copyleft licence on which the Work was based. This authorization is unlimited in time, territory and quantity.

In Prague on May 16, 2025

Abstract

Using formal logic to verify practical solutions is a well-known method used by a wide variety of fields, ranging from hardware verification to controller synthesis. MONA is a tool that takes in a theory and outputs an automaton that can be used not only to verify, but also find solutions. It is an old tool, whose code is hard to comprehend and harder to modify. This thesis analyzes, recreates and expands MONA's capabilities.

Keywords formal logic, finite automata, MONA, decidability, compiler, binary decision diagram

Abstrakt

Použití formální logiky pro verifikaci praktických řešení je dobře prozkoumaná metoda, využívaná mnoha různými obory od hardwarové verifikace až po syntézu kontrolerů. MONA je nástroj, který dostane na vstupu popis teorie a jehož výstup je automat, který lze použít nejenom pro verifikaci, ale i pro nalezení řešení. Je to starý nástroj, jehož kód je težké pochopit a ještě těžší modifikovat. Tato práce analyzuje, znovuvytváří a rozšiřuje schopnosti MONY.

Klíčová slova formální logika, konečné automaty, MONA, rozhodnutelnost, překladač, binární rozhodovací diagram

Contents

Chapter 1 1

Introduction

There has always been a search for theories that are capable of deciding all statements within the subset they interpret. When Gödel proved with his incompleteness theorems that there are theories that fundamentally cannot prove or disprove all statements of their language, that only limited this search to weaker subgroups. The ability to decide is coveted for practical applications, as if one encodes their problem in a decidable theory, they can use the theory to verify validity. MONA is a tool that goes one step further – not only can it decide sentences, but it is also able to find for which interpretations of free variables a given formula is evaluated as true. This essentially means that with the proper setup, MONA can find solutions.

As a small introductory example, say that we have the formula:

$$X \subseteq \{0, 1, 3, 4\}$$

And we'd like to find for which evaluations of X this formula holds true. We can create a MONA program, described in Figure 1.1.

```
var2 X;
X sub {0, 1, 3, 4};
```

Figure 1.1: Example MONA program.

MONA returns us a description of an automaton identical to the state diagram in Figure 1.2, that when given an evaluation of X in the form of a series of bits denoting if a certain natural number is a part of it – first zero, then one, then two, and so on – can decide whether the formula holds or not. From the diagram we can see that as long as X does not contain the number two or any number higher than four, the automata does not leave an accepting state, signifying that the formula holds, which is the expected result. But, we can also look at it from a different angle – if we take any accepting state, finding a sequence that leads us into this state effectively gives us a possible solution.

First unveiled in 1995, MONA has evolved significantly from its original inception, bringing new features and more efficient methods with each release. It has been successfully used for hardware verification, controller synthesis, computational linguistics, protocol verification, verification of distributed programs and programs in general and theorem provers. However, its development has all but halted, as the last update was in 2020, yet there are still areas for improvement. Its decades-old codebase also isn't kind to attempts to modify or

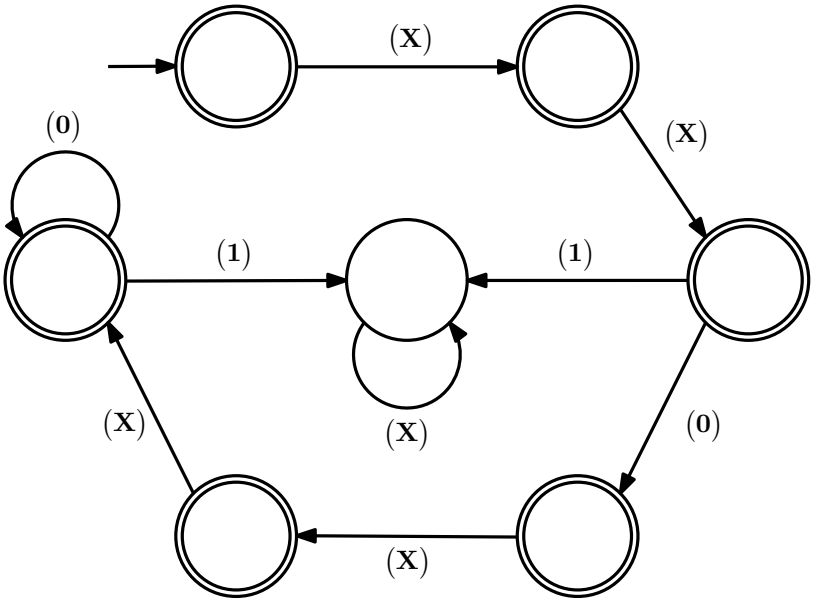


Figure 1.2: Automata for formula $X \subseteq \{0, 1, 3, 4\}$.

extend it. It is written in a mixture of C and C++ and goes to great lengths to optimize performance, usually by utilizing complicated macros. These choices, while certainly providing the desired performance boost, however prove detrimental when trying to analyze the code's function.

The aim of this thesis is to analyze the current state of MONA, search for alternative algorithms that work better and recreate MONA from the ground up with modern, compiler-inspired techniques.

Chapter 2 3

Classical Theoretic Approach

The algorithmic operations done by both MONA and us to achieve our goal aren't selected arbitrarily, but put together with purpose. Thanks to guidance from the following theory, it has been proven that what we are doing is not just feasible, it is also accurate.

2.1 Decidable Theories

Before we begin dissecting our algorithm's inner workings, it is advisable to define our accepted input.

Selecting our accepted formal logic is not a trivial task, as not all are created equal, and choosing one that isn't decidable would lead to, in the best case, automata that do not give the expected output or do not halt, or, in the worst case, to logical constructs that we are unable to construct an automaton for altogether.

Our selected theory "framework" will be a fragment of second-order logic called Weak Monadic Second-order Theory of One Successor (WS1S), as described in [5]. The language of this theory is:

- first-order terms t
  - first-order variables $p_1, p_2, ...$
  - a single-variable function $S(t)$, or $t + 1$, denoting the successor of t
- second-order terms T
  - second-order variables $P_1, P_2, ...$
  - a two-variable function $T_1 \setminus T_2$, denoting the set difference
- formulas ϕ
  - the predicates $t \in T$ (contains), $t_1 < t_2$ (less than), $t_1 = t_2$, $T_1 = T_2$ (equal) and $T_1 \subseteq T_2$ (subset)
  - propositional operators $\phi_1 \wedge \phi_2$ (AND) and $\neg\phi$ (NOT)
  - first-order existential quantifier $\exists t : \phi$
  - second-order existential quantifier $\exists T : \phi$

We also recognize these as syntactic sugar – effectively shorthands:

- first-order terms t

Term	Interpreted as
Constant 0	$(\forall t_2)\neg(t_1 = t_2) \Rightarrow (t_1 < t_2)$
Constants $c = 1, 2, ... \in \mathbb{N}$	$\underbrace{S(S...(0)))}_{repeat\ c\ times}$
Function $t + c, c \in \mathbb{N}$	$\underbrace{S(S...(t)))}_{repeat\ c\ times}$

- second-order terms T

Term	Interpreted as
Constant $\emptyset$	$T \setminus T$

- formulas ϕ

Formula	Interpreted as
Predicate $T = T_1 \cap T_2$	$(\forall t)(t \in T \Leftrightarrow t \in T_1 \wedge t \in T_2)$
Predicate $T = T_1 \cup T_2$	$(\forall t)(t \in T \Leftrightarrow t \in T_1 \vee t \in T_2)$
Predicate $\texttt{singleton}(T)$	$(T \neq \emptyset) \wedge$ $(\forall t_1, t_2)((t_1 \in T \wedge t_2 \in T) \Rightarrow t_1 = t_2)$
Propositional operator $\phi_1 \vee \phi_2$	$\neg(\neg\phi_1 \wedge \neg\phi_2)$
Propositional operator $\phi_1 \Rightarrow \phi_2$	$\neg(\phi_1 \wedge \neg\phi_2)$
Propositional operator $\phi_1 \Leftrightarrow \phi_2$	$(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)$
First-order quantifier $\forall t : \phi$	$\neg(\exists t : \neg\phi)$
Second-order quantifier $\forall T : \phi$	$\neg(\exists T : \neg\phi)$

Now that we have defined the language of the formal logic (not to be confused with the language that our automata are built over, which we will describe later), let us go over the individual parts of the theory's name to dissect the restrictions the theory imposes.

Second-order Variables either denote an element in the domain of discourse (these are first-order) or denote a relation (second-order).

Monadic The theory limits second-order variables to only those that represent unary relations. From now on we will consider these to be representing sets instead – namely, the set of elements which are in the relation.

Weak The theory only allows for second-order terms that are interpreted as sets of *finite* size.

One Successor The only relation of *successor* assigns any first-order variable exactly one successor.

These restrictions by themselves are not enough to specify with what elements we are operating – we have yet to fully define our models, specifically what is our domain of discourse. Thus, we further define that variables are interpreted as either natural numbers, which for our purposes include zero (those are considered first-order) or sets of these numbers (second-order). We'd like to unify how we approach variables, regardless of if they're first or second order. We therefore interpret first-order variables as singleton sets (sets that only contain a single element). This isn't a free change – since we are operating over only second-order variables, if we want to use any first-order functions or predicates, we must recreate them using second-order terms. The singleton interpretation also has several issues, which are further discussed in Section 3.2, but for now, we consider this approach feasible.

All put together, these definitions and restrictions make the theory decidable. This was proven by linking WS1S formulas to regular languages, and by extension, deterministic finite automata, first by a joined effort of Büchi [5] and Elgot [8] and separately discovered by Trahtenbrot [22]. There are stronger decidable theories, the most well known and utilized being S2S (Monadic Second-order Theory of Two Successors) [18], of which WS1S is a fragment.

MONA constructs automata that either decide WS1S formula, referred to as linear mode, or WS2S formula, referred to as tree mode. This thesis shall focus solely on WS1S and MONA's linear mode.

2.2 Finite Automata

As we defined our input, so too must we define what our outputs are.

All our definitions for this section are taken from [12].

▶ **Definition 2.1** (Finite Automaton)**.** *A* finite automaton *(FA) is a 5-tuple* $(Q, \Sigma, \delta, q_0, F)$ *with the following meanings:*

- *A finite set of* states Q*.*
- *A finite set* Σ *of input symbols, also called* letters*. It is sometimes referred to as the* alphabet*.*
- *A* transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ *that takes in a state-symbol pair and returns a set of states.*
- *A* starting *(also called initial)* state q_0 *from* Q*.*
- *A subset* F *of* Q*, which denotes the* accepting *(also called final)* states*. States not belonging to* F *are sometimes referred to as* rejecting states*.*

▶ **Definition 2.2** (String). *A* string $w = w_1w_2...$ *is a finite sequence of symbols. A special case is the* empty string ε*, a string consisting of no symbols.*

The length *of a string is denoted* $|w|$*. The concatenation of string a followed by b is denoted ab and* y^k *is a string where the substring y is repeated k times.*

▶ **Definition 2.3** (Accepted String). *Let* $w = w_1w_2...w_n$ *be a string of symbols from* Σ*. An automaton* accepts *w if there is some sequence of states* $s_0, s_1, s_2, ..., s_n \in Q$ *that satisfies the following conditions:*

1. $s_0 = q_0$
2. $\forall i \in \{0, 1, ..., n-1\} : \delta(s_i, w_{i+1}) \Rightarrow s_{i+1}$
3. $s_n \in F$

If there doesn't exist such a sequence that can satisfy all the conditions, the automaton rejects *w.*

▶ **Definition 2.4** (Language). *A subset of the set of all possible strings over the alphabet* Σ *is called a* language $\mathcal{L}$*.*

▶ **Definition 2.5** (Automaton's Language). *The set of all strings that are accepted by an automaton A is called the* automaton's language*, denoted* $\mathcal{L}(A)$*.*

▶ **Definition 2.6** (Regular Languages). *A language* $\mathcal{L}$ *is* regular *if and only if there exists a finite automaton for which* $\mathcal{L}(A) = \mathcal{L}$*.*

Among other features, all regular languages share a property called the Pumping Lemma. It states that there exists some constant p, and all strings longer than the constant can be "split up" into three strings x, y, z. The middle string is not empty, and the first two put together are no longer than p. Then, if we repeat y any number of times, even zero, and put the strings back together, the resulting string is still in the language. Formally:

▶ **Theorem 2.7** (Pumping Lemma). *Let* $\mathcal{L}$ *be a language. If* $\mathcal{L}$ *is regular, then:*

$$(\exists p \in \mathbb{N}, \forall w \in \mathcal{L}, |w| \geq p) :$$
$$(\exists x, y, z)(w = xyz \wedge |y| \geq 1 \wedge |xy| \leq p)$$
$$(\forall k \in \mathbb{N}_0)(xy^kz \in \mathcal{L})$$

Special Automata Versions

▶ **Definition 2.8** (Complete Automaton). *An automaton is* complete*, if for every combination of* $Q \times \Sigma$ *a non-empty set is returned. Otherwise, it is considered* incomplete*.*

We can always transform an incomplete automaton into a complete one that accepts the same language – we simply add one more rejecting state X, whose transitions always leads to itself on every symbol, and then for all state-symbol pairs that were previously empty, we add in X to the function's result.

▶ **Definition 2.9** (Deterministic Automaton)**.** *A* deterministic finite automaton (DFA) *is an automaton such that for every combination of $Q \times \Sigma$ the resulting set contains at most one element. An automaton not satisfying this property is a* nondeterministic finite automaton (NFA)*.*

As with completeness, we can always transform an NFA into a DFA using an algorithm called the *determinization operation*. The rough idea behind the algorithm is that we start with a set containing only the start state, and then for each set of states and symbol, we find the set of states reachable from these states on this letter. We repeat this process for every state set we find. At the end, the state sets are the new states. Algorithm 2.1 has a possible implementation.

```
   input  : N = (Q_n, Σ_n, δ_n, q_0n, F_n) NFA
   output: D = (Q_d, Σ_d, δ_d, q_0d, F_d) DFA, L(N) = L(D)
 1 Σ_d ← Σ_n
 2 q_0d ← {q_0n}
 3 Queue ← {q_0d}
 4 while Queue not empty do
 5 |   current ← Queue.front
 6 |   foreach symbol ∈ Σ_d do
 7 |   |   destination ← ∅
 8 |   |   foreach old ∈ current do
 9 |   |   |   destination.add(δ_n(old, symbol))
10 |   |   end
11 |   |   δ_d(current, symbol) = destination
12 |   |   if destination not in States then
13 |   |   |   States.add(destination)
14 |   |   |   Queue.push(destination)
15 |   |   end
16 |   end
17 |   if ∃E ∈ current : E ∈ F_n then
18 |   |   F_d.add(current)
19 |   end
20 |   Queue.pop
21 end
```

Algorithm 2.1: Determinization pseudocode.

▶ **Definition 2.10** (Minimal DFA)**.** *A DFA A is* minimal *if no DFA that accepts the same language has less states than A.*

▶ **Definition 2.11** (Minimal Complete DFA)**.** *A complete DFA A is a* minimal, complete *DFA if no complete DFA accepting the same language has less states than A.*

Minimal DFAs of both kinds have the property that they are unique, that is every language will have exactly one minimal DFA that accepts it, and other minimal DFAs that also accept it will be isomorphic with it. We may transform any DFA into a minimal DFA using a *minimization* operation. Algorithm 2.2 contains pseudocode for a possible implementation specifically for complete automata. There are many algorithms to do so, but here we will define Moore's version of the algorithm [2], which is a version of Myhill-Nerode minimization. Methods based on it work with the idea of equivalence classes of states. These are sets of states that "behave the same", in that for any possible string they all accept or reject. Moore's version works as a top-down refinement process. It partitions the states into equivalence classes, first very rough ones, then refines these classes by splitting them into smaller and smaller classes until no more splits can be made.

Automata Operations

There are several other operations we may wish to perform on automata. The three other operations of interest are the *complement, product* (which is, among other things, equivalent to language intersection) and *projection* operations.

The *complement* operation produces an automaton A' from A that accepts a language $\mathcal{L}(A')$ that is the complement to $\mathcal{L}(A)$ – the set of all other possible strings over the alphabet Σ. The 5-tuple for A' $(Q, \Sigma, \delta, q_0, F)$ is the same as A, except it redefines $F_{A'} = Q_A \setminus F_A$.

The *product* operation takes in two automata A, B and creates an automaton C which satisfies $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B)$. It identifies states in the new automaton as pairs of states from the old automata and creates a transition if both states from the old automata had it. A state accepts if both old accepted. A possible implementation is in Algorithm 2.3.

The *projection* operation takes in an automaton A and a function $f : \Sigma \to \Sigma'$. For every string w we define its *projection* w' as the result of applying f on every symbol of w. The resulting automaton A' accepts a language that is the set of all projected $w \in \mathcal{L}(A)$. The implementation of this operation is done by copying A into A', then applying f on all elements of $\Sigma_{a'}$ and all symbol portions of the state-symbol pairs in $\delta_{a'}$.

The resulting automaton that MONA (and thus by extent our implementation) produces are minimal, complete DFAs. However, during the process we may encounter other automaton types.

```
   input  : A = (Q_a, Σ_a, δ_a, q_0a, F_a) complete DFA
   output: A' = (Q_a', Σ_a', δ_a', q_0a', F_a') complete minimal DFA
 1 remove all unreachable states from A – states that cannot be reached
     from the starting state on any string
 2 initial equivalence classes ← F_a, Q_a \ F_a
 3 while can create more classes do
 4 |   foreach equivalence class A do
 5 |   |   if ∃x ∈ Σ_a : at least two states a, b have transitions on x to
 |   |     states in different equivalence classes then
 6 |   |   |   add new equivalence class B
 7 |   |   |   foreach state ∈ A do
 8 |   |   |   |   if δ_a(state, x) leads to same equivalence class as a then
 9 |   |   |   |   |   remove state from A
10 |   |   |   |   |   add state to B
11 |   |   |   |   end
12 |   |   |   end
13 |   |   end
14 |   end
15 end
16 Q_a' ← equivalence classes
17 Σ_a ← Σ_a'
18 δ_a' ← δ_a, but replace all states with their equivalence classes
19 q_0a' ← the class that contains q_0a
20 foreach state ∈ Q_a' do
21 |   if ∃ state in that class that was final then
22 |   |   add state to F_a'
23 |   end
24 end
```

Algorithm 2.2: Minimization pseudocode.

```
   input  : A = (Q_a, Σ_a, δ_a, q_0a, F_a), B = (Q_b, Σ_b, δ_b, q_0b, F_b) DFAs
   output: C = (Q_c, Σ_c, δ_c, q_0c, F_c) DFA, L(C) = L(A) ∩ L(B)
 1 Σ_c ← Σ_a ∩ Σ_b
 2 q_0c ← {q_0a, q_0b}
 3 Queue ← {q_0c}
 4 while not(Queue.empty) do
 5 |   [a, b] ← Queue.front
 6 |   foreach symbol ∈ Σ_c do
 7 |   |   Anew ← δ_a(a, symbol)
 8 |   |   Bnew ← δ_b(b, symbol)
 9 |   |   if both new are defined then
10 |   |   |   δ_c([a, b], symbol) ← [Anew, Bnew]
11 |   |   |   if destination ∉ States then
12 |   |   |   |   States.add(destination)
13 |   |   |   |   Queue.push(destination)
14 |   |   |   end
15 |   |   end
16 |   end
17 |   if a ∈ F_a ∧ b ∈ F_b then
18 |   |   F_c.add([a, b])
19 |   end
20 |   Queue.pop
21 end
```

Algorithm 2.3: Product pseudocode.

2.3 Formula-Automata Connection

Now that we've established our input and output formats, it's time to lay the foundations of how we can correctly transform (or in computer terms, compile) one to the other.

Given a formula ϕ with k free variables, we define the semantics as interpreted by a string w over the alphabet $\mathbb{B}^k$, where $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$. First, we assign each free variable a unique number in the range $1, 2, .., k$, called the variable's *index*. Let P_i denote the variable with index i. Each variable is described by a *track* within the string w, which corresponds to the bits found at the variable's index. The string w then represents an interpretation $w(P_i)$ of variable P_i as the finite set:

$$\{j \mid \text{the } j\text{th bit in the } P_i\text{-track is } \mathbf{1}\}$$

where we index starting from zero. As an example, the following string, whose letters are written horizontally:

$$\begin{matrix} & 0 & 1 & 2 & 3 & 4 & 5 \\ \begin{pmatrix} A \\ B \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{matrix}$$

denotes an interpretation where $A = \{1, 2, 5\}$ and $B = \{2, 4\}$. Thus, each letter can be thought of as a bit array, where a $\mathbf{1}$ means the set representing the variable at that index contains that natural number, a $\mathbf{0}$ means it does not. If we do not care about the inclusion status of a number within a variable, we denote it with $\mathbf{X}$. The meaning of each letter changes depending on where in the string it is located – the first letter denotes what variables contain the number zero, the second the number one, and so on. Several strings can therefore encode the same interpretation of variables, if they are suffixed by letter(s) whose components are all $\mathbf{0}$s. We consider a string to be *minimal* if it doesn't feature such a suffix.

$$\begin{array}{lll}
w \vDash \neg\phi & \text{iff} & w \nvDash \phi \\
w \vDash \phi_1 \wedge \phi_2 & \text{iff} & w \vDash \phi_1 \wedge w \vDash \phi_2 \\
w \vDash \exists P_i : \phi & \text{iff} & \exists \text{ finite } M \subseteq \mathbb{N} : w[P_i \mapsto M] \vDash \phi \\
w \vDash P_i = P_j & \text{iff} & w(P_i) = w(P_j) \\
w \vDash P_i < P_j & \text{iff} & min(w(P_i)) \ < \ min(w(P_j)) \\
w \vDash P_i \subseteq P_j & \text{iff} & w(P_i) \subseteq w(P_j) \\
w \vDash P_i = P_j \setminus P_k & \text{iff} & w(P_i) = w(P_j) \setminus w(P_k) \\
w \vDash P_i = P_j + 1 & \text{iff} & w(P_i) = \{x + 1 \mid x \in w(P_j)\}
\end{array}$$

Figure 2.4: Binary semantics.

We can then define the semantics inductively. We use the notation $w \vDash \phi$ (w satisfies ϕ) iff the interpretation denoted by w makes ϕ true. As a reminder, first-order variables are interpreted as second-order singletons, and thus their representation using the track system is valid. Figure 2.4 defines our semantics, with $w[P_i \mapsto M]$ meaning a minimal string w' that interprets all variables other than P_i in the same way as w does, but interprets P_i as M – essentially replacing the P_i track with M. If w is minimal, then w' must be of the form $w' = ww''$, where w'' is made up of letter(s) where only the ith component can differ from **0**.

Now that we have defined the semantics of a string, for a theory (a set of formulas) ϕ we define its *language* $\mathcal{L}(\phi)$ as the set of all strings that describe a model for the theory.

$$\mathcal{L}(\phi) = \{w : w \vDash \phi\}$$

We can now, once again using induction, construct a deterministic finite automaton A that accepts a language identical to $\mathcal{L}(\phi)$, and thus prove that $\mathcal{L}(\phi)$ is a regular language. Automata for atomic formulas (the last five) are constructed by hand, while automata for composite formulas (the first three) are constructed using automaton operations.

2.4 Atomic Formula Automata

> *Atomic automata are simple, with those required by our theory possessing no more than three states. Therefore, it's easiest just to construct them explicitly.*

We provide two methods of defining these automata. One is the standard transition diagram, the other is a text-based method. The second method is further expanded upon in Section 5.1, but for now we require only two constructs from it, and even those not in full. We define a state with the following BFN-like grammar, whose full syntax can be found in Appendix B:

$$\textit{state} \to \texttt{<}\ \textit{name}\ \texttt{(}\ \textit{stval}\ \texttt{)}\ [\texttt{init}]^{?}$$

The key parts of this grammar are the nonterminals *stval*, which denotes whether the state is accepting or rejecting, and the optional `init` phrase at the end, denoting the starting state. A transition is defined with this grammar:

$$\textit{tran} \to \texttt{>}\ \textit{name}\ [\ \texttt{default} \mid \texttt{@}\ [\textit{ltr}]^{\oplus}\ \texttt{@}\ ]\ \textit{name}$$

The `default` type of transition defines a transition to be used for letters that haven't decided explicitly. The nonterminal *ltr* is used to define the letter for a transition. A letter is defined by a series of pairs of variable and bit value, − for a **0** bit, + for a **1** bit. Any variables not mentioned are automatically assigned an **X** bit for this letter.

For the sake of brevity, all automata are shown for cases where $i = 1, j = 2, k = 3$, and we show only the part of the letter for i, j, k. Automata of the same formula, but with different orderings of variables, would be the same, save for the bits in letters being reordered to mirror the reordered variables. Similarly, any variables not used would be denoted by a **X** in all letters. In addition, automata that deal with first-order variables are made assuming that they are encoded as singleton sets, although the automata do not enforce it. The atomic automata are as follows:

Equality $P_i = P_j$ – The bits encountered must be the same, otherwise we reject. Figure 2.5 contains both the MONA code to construct this automaton and a state diagram of the automaton.

Subset $P_i \subseteq P_j$ – If we encounter a **1** bit for P_i, we also must find **1** for P_j. Figure 2.6 contains both the MONA code to construct this automaton and a state diagram of the automaton.

Less Than $P_i < P_j$ – The **1** bit for P_i must be encountered sooner than for P_j. Figure 2.7 contains both the MONA code to construct this automaton and a state diagram of the automaton.

Set Difference $P_i = P_j \setminus P_k$ – If we encounter a **1** bit for P_i, we also must find **1** for P_j and **0** for P_k. If P_i is **0**, then either both P_j and P_k are also **0**, or P_j is **1**. Figure 2.8 contains both the MONA code to construct this automaton and a state diagram of the automaton.

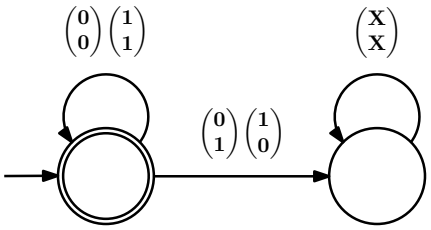
Increment $P_i = P_j + 1$ – When we encounter a **1** bit for P_i, the next letter must contain **1** for P_j. Figure 2.9 contains both the MONA code to construct this automaton and a state diagram of the automaton.

Singleton `singleton`(P_i) – This is not one of the atomic formula directly required by our formal logic, but it is the quality we require of our first order variables and therefore is also included here for the sake of completeness. It checks that we encounter exactly one **1** bit for P_i. Figure 2.10 contains both the MONA code to construct this automaton and a state diagram of the automaton.

```
< accepting (+) init;
< rejecting (-);

> accepting @ i+ & j- @ rejecting;
> accepting @ i- & j+ @ rejecting;
> accepting default accepting;
> rejecting default rejecting;
```

(a) MONA code



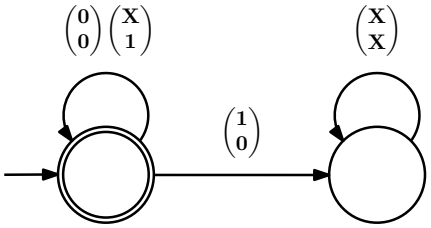
(b) Automata

Figure 2.5: Formula $P_i = P_j$.

```
< accepting (+) init;
< rejecting (-);

> accepting @ i+ & j- @ rejecting;
> accepting default accepting;
> rejecting default rejecting;
```

(a) MONA code



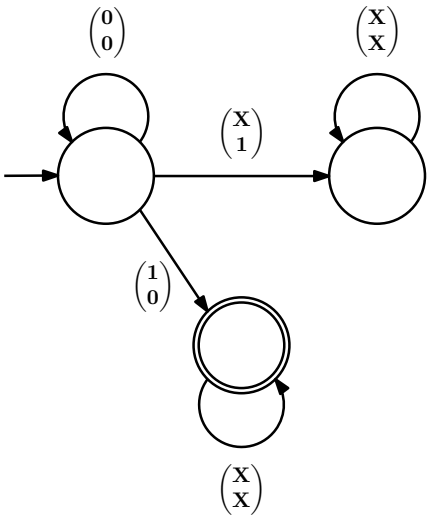
(b) Automata

Figure 2.6: Formula $P_i \subseteq P_j$.

```
< start (-) init;
< accepting (+);
< rejecting (-);

> start @ i- & j- @ start;
> start @ i+ & j- @ accepting;
> start default rejecting;
> accepting default accepting;
> rejecting default rejecting;
```

(a) MONA code



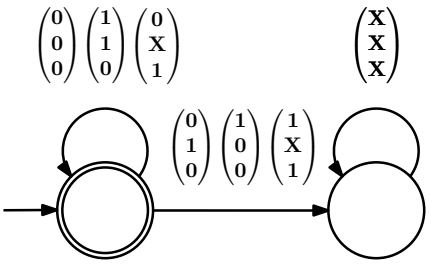
(b) Automata

Figure 2.7: Formula $P_i < P_j$.

```
< accepting (+) init;
< rejecting (-);

> accepting @ i- & j+ & k- @ rejecting;
> accepting @ i+ & j- & k- @ rejecting;
> accepting @ i+ & k+ @ rejecting;
> accepting default accepting;
> rejecting default rejecting;
```

(a) MONA code



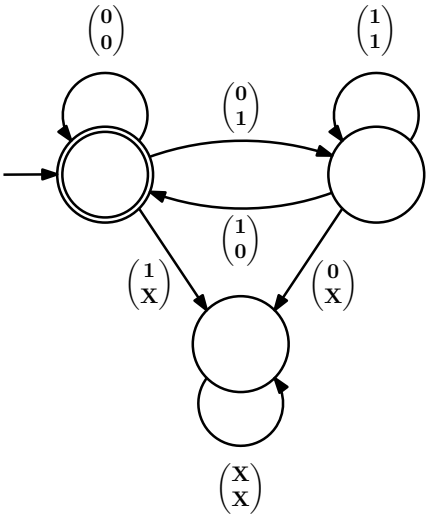
(b) Automata

Figure 2.8: Formula $P_i = P_j \setminus P_k$.

```
< start (+) init;
< increment (-);
< rejecting (-);

> start @ i- & j- @ start;
> start @ i- & j+ @ increment;
> start default rejecting;
> increment @ i+ & j+ @ increment;
> increment @ i+ & j- @ start;
> increment default rejecting;
> rejecting default rejecting;
```

(a) MONA code



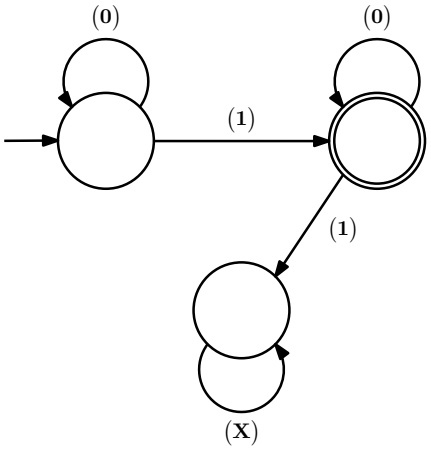
(b) Automata

Figure 2.9: Formula $P_i = P_j + 1$.

```
< start (-) init;
< found (+);
< rejecting (-);

> start @ i+ @ found;
> start default start;
> found @ i+ @ rejecting;
> found default found;
> rejecting default rejecting;
```

(a) MONA code



(b) Automata

Figure 2.10: Formula `singleton`(P_i).

2.5 Composite Formula Automata

Unlike atomic automata, we can't explicitly write out the resulting composite automata. Therefore, we shall provide algorithmic solutions that take in one or two automata and combine them into a single automaton. These go from the simplest, negation, to conjunction, to lastly the existential quantifier, the most complex.

Negation

$\neg\phi$ – Negation corresponds to automata complementation. Therefore, if we already have an automaton A' for which $\mathcal{L}(\phi') = \mathcal{L}(A')$ holds true, then

$$\mathcal{L}(\phi) = \mathcal{L}(\neg\phi') = \texttt{complement}(\mathcal{L}(A')) = \mathcal{L}(\texttt{complement}(A'))$$

and the wanted automaton A is $\texttt{complement}(A')$. The actual implementation is done by flipping accepting and rejecting states.

Conjunction

$\phi_1 \wedge \phi_2$ – Conjunction corresponds to language intersection.

$$\mathcal{L}(\phi) = \mathcal{L}(\phi_1 \wedge \phi_2) = \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_2)$$

The desired automaton A is the automaton product $A_1 \times A_2$. While the result can theoretically be as large as the product state space, it is usually much smaller, in part due to only reachable states being constructed.

Existential Quantifier

$\exists P_i : \phi$ – The existential quantifier should construct an automaton A that functions the same way as A', except that it is allowed to "guess" values on the P_i track. We achieve this by the *projection* operation, where the function passed to the projection removes the P_i track from all letters. This effectively merges letters that only differed on this track, thus allowing the resulting A'' up to two transitions out of each state on each letter, making it non-deterministic. To get A, we simply determinize A''. Unfortunately, just A'' is not enough, as the witness $w[P_i \mapsto M]$ may be longer than w. That corresponds to a rejecting state s that is reachable on w from the start, and that from s there is an accepting state reachable from it on a string consisting of letters of the form

$$\begin{pmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{X} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{pmatrix}$$

where the row occupied by $\mathbf{X}$ is the track belonging to P_i. It's therefore enough to find these states and change them to accepting before the *projection* operation. This can be done using a BFS algorithm on a reversed automaton, starting from a state that neighbors all accepting states. This acceptance *expansion* is done in linear time, thus not negatively impacting the runtime. It has been shown in [1] that while in theory the entire process can be exponential, the usual runtime is linear.

In more theoretic terms, these operations can be defined as: Let the *right-quotient* of a language L with a language L' be the set

$$L/L' = \{w : \exists u \in L', w \cdot u \in L\}$$

We define the *projection* operation E^i as the set

$$E^i(L) = \{w : \exists w', w \text{ is the same as } w' \text{ except for track } i\}$$

Then, take a language L^i of the form

$$L^i = \{w \in (\mathbb{B}^k)^* : \text{the } P_j\text{-track of } w \text{ is of the form } \mathbf{0}^* \ \forall j \neq i\}$$

The desired language and automata can then be found as

$$\mathcal{L}(\phi) = \mathcal{L}(\exists P_i : \phi') = E^i(\mathcal{L}(\phi')/L^i)$$

MONA's Theoretic Approach

While the classical approach just described works for the most part, it is not identical to what MONA utilizes. This is because it brings in problems that are difficult or costly to work around. This chapter discusses the changes to the theory MONA makes to combat these problems.

3.1 Binary Decision Diagrams

Classical transition tables are data structures that grow very large, very quickly. MONA chooses a different method to store its automata.

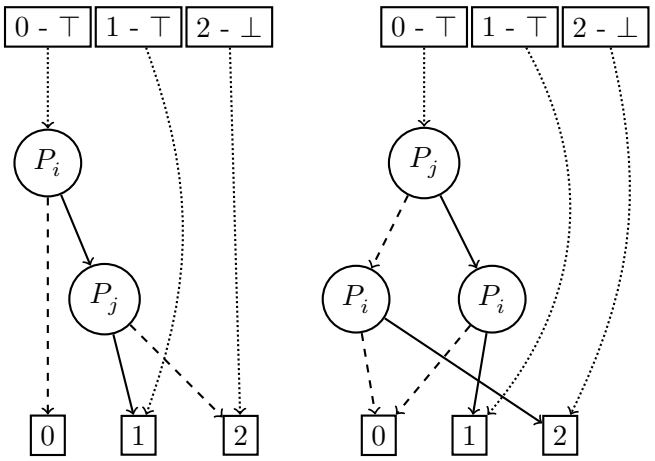
The biggest change MONA employs that differentiates it from tools that preceded it is the method used to store the automata. Since the alphabet of the language grows exponentially with the number of variables, a classic transition table would quickly grow too large to be practical. It would also feature a lot of redundant information, especially for states that only care about a few variables.

To reduce the amount of information stored about the automata to the bare minimum to still describe it, MONA uses a multi-terminal, shared Binary Decision Diagram (BDD) [15] to represent each state, with the resulting BDD forest being the entire automata. BDDs are rooted, directed acyclic graphs, traditionally made up of several internal nodes and two leaf nodes. Each node is made up of three parts – the variable this node decides on, and two child nodes, denoted the *zero child* and *one child*[1]. Their traditional use is to store formulas or boolean functions, with internal nodes representing variables and the two leaf nodes representing whether it evaluates as true or false. In our use-case the leaves of the BDD are not limited to just two, as they represent the next state reached by the transition.

The way a transition is derived is as follows: start at the root node that corresponds to the current state. Then, look at the value of the variable the node decides. If it is **0**, proceed to the *zero child* next, otherwise go to the *one child* next. Repeat this process until you reach a leaf node. If a variable isn't encountered, we do not care about its evaluation – it can be either **0** or **1**. MONA enforces that the variables are encountered in a strictly ascending order, as it makes for easier construction and minimization.

[1] MONA's documentation uses different terminology, calling children *successors* instead, and using the phrases *low* and *high* instead of zero and one.

Figure 3.1 represents denotes a possible BDD. Round nodes are internal nodes, while rectangular nodes are state nodes. In implementation, a leaf node is defined as an internal node, but here for better readability leaf nodes are separated and treated as state nodes instead. Dashed edges represent a *zero child* connection, full a *one child* connection, while dotted edges are link a root to the start of its tree (or a leaf, if the state it represents does not care about the letter it is deciding on). The state labelled 0 is always the starting state.



(a) BDD for $i < j$. **(b)** BDD for $i > j$.

Figure 3.1: Two possible BDD versions for the formula $P_i \in P_j$.

Note that depending on the ordering of variables, the BDD can have either two internal nodes or three. This shows that even the most basic of automata are sensitive to the chosen ordering of variables. This applies to more complex automata as well, and in general, depending on the chosen ordering, the resulting BDD for some formulas can range from a linear number of BDD nodes to exponential [4]. For example, take the following formula:

$$A_1 = B_1 \wedge ... \wedge A_n = B_n$$

If the variables are declared in the order $A_1, B_1, ..., A_n, B_n$, the size of the resulting BDD is linear in the size of n. Intuitively, this is because the BDD only has to "remember" at most one bit of information at a time, and can "forget" the value of a variable once it's compared, which results in a merging of nodes. However, if the order of variables is $A_1, ..., A_n, B_1, ..., B_n$, the size of the BDD is exponential in the size of n, as the BDD has to "remember" the value of all variables A_x before it can start to "forget" them.

Unfortunately, finding an optimal reordering is NP-complete [3], and even worse, polynomial approximation algorithms with any performance ratio have been shown by Sieling [20] to exist only if P = NP.

3.2 First Order Variable Issues

When we initially discussed the formula-automata connection, we glossed over how first-order variables are handled. The method of just using singleton sets has a couple glaring issues, which we will explore and try to remedy in this section.

As said before in Section 2.1, the classical approach interprets first-order variables as singleton sets, where the singular element denotes the term's value. These semantics by themselves are not closed under negation (or in automata terms, under complementation). In simple terms, with negation we lose the singleton property that's required of first-order variables, and we must reestablish it.

To illustrate with an example, a formula $\phi_1 : p = 0$ would be evaluated as $\phi_2 : P = \{0\}$, where p is first-order and P is second-order. However, the complement of ϕ_2 would be $\neg(P = \{0\})$, which is different from the complement of ϕ_1 – that would be $\neg(P = \{0\}) \ \wedge \ \texttt{singleton}(P)$, where $\texttt{singleton}(X)$ is a predicate that evaluates whether X is a singleton set. We could attach this $\texttt{singleton}(X)$ predicate to every free variable and every subformula that variable is a part of. Sometimes this is the preferred method, as singleton encoding has the unique property that every first-order variable has precisely one way of being encoded. The price for this is an additional product and minimization operation, which we have already established are non-trivial procedures we'd like to avoid whenever possible. MONA alters the classical approach in two ways to combat these shortcomings.

1. First-order variables are interpreted as non-empty sets, with the value denoted by them being the smallest number present in the set.

2. Instead of states falling into two categories of *accepting* ($\top$) and *rejecting* ($\bot$), MONA adds in a third *don't-care* ($\sim$) state type that is used to handle formula restrictions.

Let us tackle these in order. There are multiple other methods of encoding first-order values in non-empty sets, such as the maximum of the set, the number of elements in a set, the sum of elements in the set, as bit strings in base 2 (which is the method used when implementing Presburger arithmetic in MONA, further described in Section 3.5), etc. The choice of using the minimum may seem arbitrary at first, but is quickly made apparent by the fact that any automata that interact with first-order variables don't have to "reevaluate" what number that variable is once it is known. This results in slightly more efficient encoding than the singleton method in terms of the complexity of atomic automata. For example, the atomic automata for the formula $x \in Y$

requires three states for both the singleton and the minimum method. However, the minimum method requires one less transition, as it's accepting state is a sink that cannot be exited on any transition. The singleton method's accepting state has a transition on the value x, and goes to the always rejecting state if it ever encounters a **1** for it.

▶ **Lemma 3.1.** *Using the method of bit-encoding in base two for first order variables, the atomic automaton $\in$ cannot be a finite automaton, as it would have to accept a language that is not regular.*

Proof. By contradiction, using the Pumping Lemma (PL) defined in Section 2.2. Without loss of generality, let us assume the automaton is constructed over the formula $f \in S$, with its language denoted $\mathcal{L}$. For a given value of p from PL, let us select w as a string that encodes $2^{(p+1)} \in \{2^{(p+1)}\}$. This is certainly a string that is in $\mathcal{L}$.

Now, no matter the choice of x, y, z, y will always contain some amount of letters which are **0** bits for both f and S. Then, if we select $k = 3$, the values now encoded are:

$$f' = 2^{(p+1)} \cdot 2^{|y^2|}$$
$$S' = \{2^{(p+1)} + |y^2|\}$$

The newly created formula $f' \in S'$ would hold only if the initial selection of value f was the result of the following equation, where $a = |y^2|$:

$$2^{(p+1)} \cdot 2^a \stackrel{?}{=} 2^{(p+1)} + a$$
$$2^{(p+1)} \cdot (2^a - 1) \stackrel{?}{=} a$$
$$2^{(p+1)} \stackrel{?}{=} \frac{a}{2^a - 1}$$

Since $a \geq 2$, the denominator is always larger than the numerator. Therefore, $2^{(p+1)}$ would have to be a value outside the domain of natural numbers, which we can never encode. The new formula is always false, $xy^3z \notin \mathcal{L}$, and as such contradicts PL. □

There are other encoding methods possible that do not put any restrictions on the sets that may encode first-order variables, and thus in theory do not require implementing ternary semantics. One of those defines the empty set as encoding the number zero, and all other sets encode the number equal to their minimum. It is not clear why MONA did not choose this encoding method, as it is very similar to what it already uses. However, even with this encoding ternary semantics would still see usage, namely in differentiating the first boolean-only state in automata that do not interact with zero-order variables, and user-defined restrictions.

3.3 Ternary Semantics

Ternary semantics add in a third type of state. Here we discuss the reasoning and benefits that third type brings, as well as the way it is defined.

This section is based entirely off of the work done in [14]. The switch to ternary logic is done to enable explicit restrictions. These restrictions are used to constrain variables, limiting the values they may take on to some subset. This system was originally created to properly enforce the singleton requirement placed on first-order variables, but has been expanded to enable user-defined restrictions.

All variables P are associated with a formula ρ that restricts the values P can evaluate to some subset. A formula ϕ for a given interpretation can then evaluate to either the standard *true* ($\top$) or *false* ($\bot$), or the new third option of *don't-care* ($\sim$)[2] when any restriction on the free variables of ϕ is not fulfilled. In cases where we wish for no restriction, ρ can simply equate to $P = P$ or any other formula that always evaluates as true.

The semantics of propositional operators with the three-valued interpretations are identical to the semantics described beforehand, with the added rule that a formula is $\sim$ if and only if a subformula is $\sim$. For predicate operators, a formula is $\sim$ iff all of the possible interpretations of the bound variable lead to a $\sim$ formula.

In more theoretical terms, let X be an expression – that is either a formula or a variable. Let $\rho^*(X)$ be the formula that is the conjunction of all $\rho(P_i)$, where P_i is either free in X, or free in $\rho(P_j)$ for some P_j that is free in X, and so on. We refer to smallest set of these free variables that directly or transitively appear in X as $\mathcal{P}(X)$. Then $\rho^*(X)$ is the closure of restrictions on all variables in $\mathcal{P}(X)$, which we assume without prejudice is finite and define as:

$$\rho^*(X) = \bigwedge_{P_i \in \mathcal{P}(X)} \rho(P_i) \quad \left| \begin{array}{c} \text{Provided:} \\ FV(X) \subseteq \mathcal{P}(X) \\ \bigcup_{P \in \mathcal{P}(X)} FV(\rho(P)) \subseteq \mathcal{P}(X) \end{array} \right.$$

where $FV(\alpha)$ is the set of all free variables in α. Tables 3.2 then define the truth tables for the two used propositional operators, while the three-valued semantics are defined in Figure 3.3, with $[\![\phi]\!]w$ signifying w satisfies ϕ in ternary semantic interpretation.

The other atomic formula are treated in the same manner – the value of an atomic formula is $\sim$ if the value of the conjugation of all restrictions is not $\top$, otherwise the value is identical to binary semantics.

[2] MONA's documentation uses a different notation for state types, in which 1 is *true*, 0 is *false* and $\bot$ is *don't-care*.

$\neg^3$	
$\top$	$\bot$
$\bot$	$\top$
$\sim$	$\sim$

(a) Negation operator.

$\wedge^3$	$\top$	$\bot$	$\sim$
$\top$	$\top$	$\bot$	$\sim$
$\bot$	$\bot$	$\bot$	$\sim$
$\sim$	$\sim$	$\sim$	$\sim$

(b) And operator.

Figure 3.2: Truth tables for ternary propositional operators.

$$
\begin{aligned}
[\![\neg\phi]\!]w &= \neg^3[\![\phi]\!]w \\
[\![\phi_1 \wedge \phi_2]\!]w &= [\![\phi_1]\!]w \wedge^3 [\![\phi_2]\!]w \\
[\![\exists\rho(P_i) : \phi]\!]w &= \begin{cases} \top & \exists M : [\![\phi]\!]w[P_i \mapsto M] = \top \\ \bot & (\forall M : [\![\phi]\!]w[P_i \mapsto M] \neq \top) \wedge \\ & (\exists M : [\![\phi]\!]w[P_i \mapsto M] = \bot) \\ \sim & \forall M : [\![\phi]\!]w[P_i \mapsto M] = \sim \end{cases} \\
[\![P_i = P_j]\!]w &= \begin{cases} \top & (w \vDash P_i = P_j) \wedge ([\![\rho^*(P_i) \wedge \rho^*(P_j)]\!]w = \top) \\ \bot & (w \nvDash P_i = P_j) \wedge ([\![\rho^*(P_i) \wedge \rho^*(P_j)]\!]w = \top) \\ \sim & [\![\rho^*(P_i) \wedge \rho^*(P_j)]\!]w \neq \top \end{cases}
\end{aligned}
$$

Figure 3.3: Three-valued semantics.

A keen-eyed reader will have observed that while the restriction will automatically "bubble up" when a propositional operator is used, that does not seem to be the case for predicate operators. This brings us to the only requirement that we have for restriction formulas, specifically for restriction to variables that show up in predicate operators. This restriction simply is that if the restrictions the restriction refers to are satisfied, then this restriction must be satisfiable. Formally, for any formula $\phi = \exists\rho(P_i) : \phi'$, the requirement is:

$$\bigwedge_{P \in \mathcal{P} \setminus P_i} \rho(P) \vDash \exists P_i : \rho$$

With this requirement in place, the restrictions now properly "bubble up". Therefore, it is sufficient to join them to atomic automata, when the product operation is the cheapest, and we don't have to worry about reapplying the restrictions further down the line. The ternary semantics are now equivalent to the classical binary semantics in the following ways:

$$
\begin{aligned}
w \vDash \phi \wedge \rho^*(\phi) &\Leftrightarrow [\![\phi]\!]w = \top \\
w \vDash \neg\phi \wedge \rho^*(\phi) &\Leftrightarrow [\![\phi]\!]w = \bot \\
w \nvDash \rho^*(\phi) &\Leftrightarrow [\![\phi]\!]w = \sim
\end{aligned}
$$

Like we described with WS1S, we now move on how to represent these ternary semantics with finite-state automata. We therefore need to construct

automata A_ϕ that determine $[\![\phi]\!]w$. Like the values a given formula may evaluate to have expanded, so must the possible automata state types expand with a third *don't-care*($\sim$) state. Let $A(w)$ denote the value calculated by A by parsing w, that is determined by the type of final state reached. Let $\mathcal{R}A$ be an automaton created from A by applying the following relabeling:

$$\mathcal{R}A(w) = \begin{cases} \top & A(w) = \top \\ \sim & A(w) \neq \top \end{cases}$$

Then we can construct an automata representing an atomic formula ϕ like so:

$$A_\phi \times \bigtimes_{P \in FV(\phi)} (\mathcal{R}A_{\rho^*(P)})$$

With A_ϕ being the classical automaton for the atomic formula as described in binary semantics, and $\times$ is the automata product using the truth table for $\wedge^3$ to calculate a state type in the result.

The MONA implementation of this is achieved very easily. First, the automaton representing the restriction(s) is constructed as if it was a regular formula. Then, all of the *rejecting* states are changed to *don't-care* states – this switching of states is implemented in MONA as the `restrict` function and we will refer to it under that name from now on. Its opposite, which switches all *don't-care* states to *rejecting*, also exists under the name `unrestrict`. Then, construction of the automaton for the actual formula is done. During this, the restriction automaton is joined to any atomic automaton that feature the variable the automaton was constructed for with an automata product.

3.4 Senary Semantics

> *The three equivalence classes a string may fall into are further refined into six, hence the name.*

In the same paper where ternary semantics were conceived [14], a proposition was made for a system that expands upon it. Consider a formula and its restriction. A string s is deemed to be *interesting* if the following holds:

$$\exists v : s \cdot v \in L \cap R \wedge \exists v' : s \cdot v' \notin L \cap R$$

Where L is the language of the automata for the formula and R is the language of the automata for the restriction. A string that does not satisfy this requirement is considered *non-interesting*. Of note is the fact that all prefixes of an *interesting* string are also *interesting*. The new equivalence classes for strings are then defined not just by the status of the end state they reach, but also by if they are *interesting* or not.

The key advantage realized is that since *non-interesting* strings will not become *interesting* with any further extensions, any strings that feature them as a prefix will always have the same *accepting/rejecting/ don't-care* end state status. Thus, if an automaton has so far read a *non-interesting* string, regardless of what letters it reads in the future, it will not encounter a state of a different type than the type it is currently in – and in a minimized automata, it will not leave the state it is currently in. Borrowing terminology from graph theory, states for *non-interesting* equivalence classes are sinks. Thus, all of the states that handle to *non-interesting* strings can be automatically minimized down to at most three states, one for each state type.

As far as our research led us to believe, senary semantics were never implemented in MONA. While it has been proven that senary semantics wouldn't be worse than ternary, it is unclear how impactful the benefits of explicit denoting of sinks would be over ternary semantics, if any at all.

3.5 Other logics in MONA

> *While primarily designed to work over WS1S, the construction methods of MONA's linear mode are not strictly limited to it, and can be used to emulate different logic fragments.*

In this section, we will explore two other logic fragments that can be emulated by MONA – Presburger Arithmetic and Monadic Second-order Logic on Strings.

Presburger Arithmetic

Presburger arithmetic is a first-order, decidable theory over natural numbers. It abandons all second-order terms and their associated formulas, instead allowing addition of two variables, not just addition by a constant. It can be implemented within WS1S by encoding numbers as second-order bit strings in base-2, with least significant bits first.

This encoding satisfies the requirement of a string denoting the same interpretation if suffixed by letter(s) of all **0**s, and therefore all the composite automata procedures can be used without change. MONA also provides a construct `pconst`(I) for creating an atomic automaton that encodes the constant I using the aforementioned method. However, other atomic automata as defined cannot be used for Presburger arithmetic. The current solution encouraged by MONA is to use user-defined macros or to edit the source code to define the proper atomic automata, both of which are effectively workarounds. Addressing the core issue is one of the motivations for our proposed additions to MONA.

Monadic Second-order Logic on Strings

If instead of interpreting over *infinite* string models, which allow for any string to be elongated by any number of letters without changing the interpretation of the string, we limit ourselves to *finite* string models, we get Monadic Second-order Logic on Strings (M2L-Str).

M2L-Str is built over the same language as WS1S and features the same restrictions. The key difference is that the universe of M2L-Str is not all naturals $\mathbb{N}$, but rather a bounded subset $\{0, ..., n-1\}$, where n is the length of the interpreting sting. This interpretation makes M2L-Str equivalent to regular languages in expressive power, and even preferable over WS1S in some applications, for example those where strings model sequences of states in time, and thus a zero-filled letter still carries meaning and cannot be added or removed at-will. These states can represent for example logical gates [1] or web service controllers [19]. However, WS1S is often more intuitive to work with, as not all elements feature a successor in M2L-Str. In conjunction with ternary semantics, described in Section 3.3, M2L-Str can be emulated in WS1S efficiently.

The standard construction procedures outlined for WS1S can be used without change for M2L-Str, with a single exception: the *acceptance expansion* operation done before a *projection* operation is omitted, as the reason it was included does not hold in M2L-Str – strings of different lengths represent fundamentally different universes, and thus different interpretations. MONA opts for a different approach, instead enforcing this restriction on universe values with a header corresponding to the following list of formulas when M2L-Str is in use:

$$\exists U(\neg(\exists p)(p \notin U \wedge p+1 \in U) \ \wedge \ \texttt{allpos}(U) \wedge \\ (\forall p)(p \in U) \ \wedge \ (\forall P)(P \subseteq U))$$

The `allpos`(U) formula is responsible for enforcing the bounded universe, a feature missing in WS1S and impossible to express, as it has no equivalent when using only already established language constructs. It constructs an automaton that is in an *accepting* state until it meets a **0** on the track belonging to U, which shifts it into a *don't-care* state that it does not leave. Figure 3.4 describes the `allpos`(U) automaton using the syntax previously used in Section 2.4 and further defined in Section 5.1.

MONA used to emulate M2L-Str differently with a variable representing the largest (and therefore last) element in the universe. This option is supported for backwards compatibility, and its automata is defined in Figure 3.5.

```
< start (+) init;
< out_of_bounds (~);

> start @ U- @ out_of_bounds;
> start default start;
> out_of_bounds default out_of_bounds;
```

Figure 3.4: MONA code for `allpos` automaton.

```
< start (~) init;
< last (+);
< out_of_bounds (-);

> start @ U+ @ last;
> start default start;
> last default out_of_bounds;
> out_of_bounds default out_of_bounds;
```

Figure 3.5: MONA code for the largest element in universe automaton.

MONA's Implementation

Now that we have described the theory supporting and in many ways enabling MONA's function, it is time to examine how MONA practically implements this process.

This chapter is based off of the MONA User Manual [15], MONA's implementation "secrets" article [16], and MONA's source code itself [17].

MONA's input is a path to a text file containing first an optional header, denoting which formal logic fragment to use (by default WS1S), followed by formal logic variables, formulas and other supporting features, such as constants, predicates and macros, or imported features from other MONA files.

MONA's output is printed into the standard console output, first denoting what the free variables are and in what order they are evaluated, followed by information about states – which one is the starting state, which are accepting, rejecting, or don't-care. Then, information about the size of the resulting BDD is provided, both in terms of just states and total nodes. Lastly, the transition function is provided, as a series of state-letter-state entries.

MONA's runtime consists of four (or optionally three) phases. They are, in order, the *Parsing* phase, the *Code Generation* phase, the optional *Reduction* phase and last the *Automaton Generation* phase.

4.1 Optimizations

Alongside the changes to theory that MONA implements, there are a number of practical optimizations that don't change the theory that MONA interacts with, but that nevertheless result in improvements in runtime.

First off, the list of accepted atomic formulas is greatly expanded, as is the list of composite formulas – the latter being expanded to all propositional and predicate operators. This eliminates the need for both the decoding of these operators into the very restrictive base language and, more importantly, usually eliminates several costly automata product or projection operations. Secondly, all MONA formulas that feature nested terms are flattened by introducing new variables bound to an exists statement that represent the result of a subterm.

And lastly, since unlike first- and second-order variables, which can take on any countable number of values, zero-order variables can only take on a constant amount of values – that is, *true* and *false* – we can encode them separately,

bypassing some of the issues first- and second-order variables face. Thus, we dedicate a specific letter position in a MONA string to zero-order variables, and zero-order variables only. This could in theory be any position, but for the sake of simplicity it is the first letter of the string. To not make our indexing more confusing, this position is indexed as the -1 position. Therefore, the first state ignores all but zeroth-order variables, and in turn all other states ignore zeroth-order variables.

4.2 Parsing

> *In the first* Parsing *phase, the input formulas are transformed into an Abstract Syntax Tree (AST) representation.*

The input formulas are transformed into an Abstract Syntax Tree (AST) representation using a parser generated via Bison parser generator [9]. An AST is a a tree representation of the input, in our case formula(s), where nodes are individual syntactic constructs of the formula(s), and the (optional) children of nodes are the operands of these constructs. For example, a node could represent the AND operator in a formula, and its two children would be the nodes representing the subformulas it is conjugating. The parser first constructs an initial "declaration" Abstract Syntax Tree (AST) that ignores the order type of terms. Then, this declaration AST is used to generate a typed AST, which gives proper types to some predicates that are different depending on if their variables are first- or second- order. Those are mainly the equality predicate ($=$) and the addition predicate ($+$).

During this construction, additional syntactic checks are preformed that couldn't or would be difficult to do with the generated parser. A symbol table is constructed that is used to track all named variables, constants and parameters. Any constant arithmetic expressions are also evaluated at this time. Both the declaration and typed AST are implemented polymorphically, with each type of formula, expression, term, variable, etc. having its own class that dictates how it generates subsequent structures, which syntactic checks to perform, and what operations to perform on the symbol table.

4.3 Code Generation

> *In the second* Code Generation *phase, the AST is transformed into a directed acyclic graph (DAG).*

This phase turns the AST into a directed acyclic graph (DAG). The phase's operation, referred to as "DAGification", is performed on sets of subformulas that are identical save for the variables they use, and that have an order-preserving

relation between the variables of the subformulas. These subformulas then correspond to identical BDDs, save for the variable renaming, and it is therefore not necessary to construct more than one. This is done as a bottom-up collapsing process, where nodes that represent atomic automata are collapsed if the same atomic formula with variables in the same ordering is encountered, and nodes representing composite automata or calls to predicates and macros are collapsed if they share the same operator (or name in the case of predicates/macros) and their operand subformula nodes are collapsed into the same node.

Even if a node isn't collapsed into another, it is written to a node vector, which effectively stores the AST in prefix notation – that is, operands that take the form of a subformula for any node are encountered before the node itself. This finalizes the transformation into a DAG. Of note is the fact that symmetric atomic automata are always constructed using the same ordering, even if their formulas used a different order. Their resulting nodes are therefore always collapsed into one.

If there are multiple top-level formulas, they are now all joined into one conjugation. This is also the point where the representation switches away from denoting sub-constructs, such as subformulas or terms, via pointers and instead uses the index of the sub-construct in this vector.

4.4 Reduction

> *In the third, optional* Reduction *phase, the DAG is optimized using three primary methods of formula reduction.*

In this phase, the DAG is optimized, using three methods of formula reduction. The first is a family of trivial rewrite rules that are inherent to the formal logic nature of WS1S. Some examples include:

$$\begin{array}{cc} X_i = X_i \rightsquigarrow true & \phi \wedge \phi \rightsquigarrow \phi \\ true \wedge \phi \rightsquigarrow \phi & \neg\neg\phi \rightsquigarrow \phi \\ false \wedge \phi \rightsquigarrow false & \neg true \rightsquigarrow false \end{array}$$

The second type of rewrite rule concerns itself with the potentially exponential blowup caused by quantifiers, specifically by the determinization. In it's original form, the rule is:

$$ex2\ X_i : \phi \rightsquigarrow \phi[T/X_i] \quad \left| \begin{array}{c} \text{Provided:} \\ \phi \Rightarrow (T = X_i) \text{ is true} \\ FV(T) \subseteq FV(\phi) \end{array} \right.$$

where $[T/X_i]$ denotes the replacement of all instances of X_i with a term T and $FV(\alpha)$ is the set of all free variables in α. However, this is not the actual

rewrite rule that MONA uses, for two reasons. Firstly, finding a suitable term T is an expensive task, potentially just as much as constructing the predicate operator itself. Secondly, since MONA requires nested formulas to be flattened, the term substitution can easily cause the quantifier to be recreated. MONA uses a slightly more restrictive rule:

$$ex2\ X_i : \phi \rightsquigarrow \phi[X_j/X_i] \quad \Big| \quad \begin{array}{c}\text{Provided:}\\ \phi \equiv ... \wedge X_j = X_i \wedge ...\\ X_j \neq X_i\end{array}$$

This rule, unlike the family of the first rule type, is not guaranteed to improve performance, since it can cause the DAG reuse rate to decrease, but it will not decrease performance.

The last rule deals with conjunctions, and is of this form:

$$\phi_1 \wedge \phi_2 \rightsquigarrow \phi_1 \quad \Big| \quad \begin{array}{c}\text{Provided:}\\ nonrestr(\phi_2) \subseteq set(\phi_1)\\ restr(\phi_2) \subseteq restr(\phi_1)\end{array}$$

where $set(\alpha)$ is the set of all conjuncts in α, $restr(\alpha)$ is the set of all `restrict` conjuncts, and $nonrestr(\alpha)$ is the set of all non-`restrict` conjuncts (in other words, $set(\alpha) = restr(\alpha) \cup nonrestr(\alpha)$).

The actual implementation of these formula reduction rules in MONA is tied closely to the AST representation. Each class of node types implements a `reduce` method that when called checks if the current node matches any of the rules that target that node type, with the current node as the root of the formula, and replaces itself if applicable.

4.5 Automaton Generation

> *In the final* Automaton Generation *phase, the actual automata construction happens.*

In this final phase, all the buildup from the previous phases finally pays off as the construction itself takes place. Before construction begins, all variables are assigned an index with which they will be referenced for the construction. This assignment is identical to the order of declaration for free variables and order of allocation for not-free variables. Therefore, all free variables will always have a higher index than non-free ones.

MONA's implementation of a BDD consists of three arrays, a *state* array, *transition* and a *node* array. Note that for the first two arrays, elements at the same index belong to the same node. The smaller state array holds information on the states of the automata the BDD represents – that is their *accepting/rejecting/don't-care* type. The transition array, same in size as the state

array, holds the index to the node that is the root of their decision tree. The final, and arguably most important array is the node array, which holds information on the actual nodes of the BDD. Each node contains the index of its children, as well as the index of the variable the node decides on. If the node is a leaf, the variable it decides on is instead the value $2^{16} - 1$, and one of its children contains the index of the next root. Thus, the number of different variables MONA can differentiate is equal to 65535. The maximum number number of nodes is similarly limited, this time to 2^{24}. Both of these limitation stem from the fact that all three indexes are stored in two 32-bit unsigned integers. On top of this BDD automata representation, the managing of the BDD makes heavy use of cache-optimization techniques, as even with the compression the BDD method allows, it makes for by far the most expensive of the phases in terms of time and utilized memory.

The components are constructed in order from the DAG's source to sink, with the final automata being the result.

Primitive automata are constructed by hand, where the BDD is constructed manually to represent the automata representing atomic formula. Some of the automata are described in Section 2.4 – those are the ones that make up the atomic formula core of WS1S theory. The remaining majority come from "syntactic sugar". While they are theoretically constructable just from the core formulas, those formulas would be complex, often involving costly conjunction or existential operations. The resulting "syntactic sugar" automata are relatively simple, usually with a constant amount of states, so it is better to construct them directly.

Negation is done by flipping the states from accepting to rejecting and vice versa. Any don't-care states are not affected. This is done in linear time by simply going over the vector containing all the states, similarly to how the `restrict` and `unrestrict` functions operate.

The remaining propositional operators are done as automata products, while predicate operators are done as an acceptance expansion (described in Section 2.5) operation followed by an automata projection operation and concluded with automata determinization. Both methods are then followed by a minimization as to reduce the size of the result. All of these automata operations are described in Section 2.2. The only difference MONA employs is how it calculates the state type of each state, which depends on the operator that is being constructed. Using these operations, only the necessary states are constructed, which are usually a smaller subset than the entire state space.

Chapter 5 35

Proposed Implementation

Now that we know the current state and capabilities of MONA, we can turn our attention to how we can improve on its design and execution.

For our implementation, we propose to separate MONA's phases into individual executables that may be run together or independently. The frontend will correspond to the *Parsing* phase. The middle end will combine the *Code Generation* and *Reduction* phases into a single phase. The backend, currently the only one for which we have created a prototype, further described in Chapter 6, corresponds to the *Automaton Generation* phase.

Naturally, the split into multiple executables raises the issue on the format of intermediate products. For the sake of backwards compatibility, the form of the input for the frontend and the (default) output of the backend is the same as for MONA – that is, a text file ending in `.mona` for the input and text printed into the console for output. For the intermediate files, we have selected to utilize the JavaScript Object Notation (JSON) file format. The output format for the frontend (which is the middle end's input format) and middle end are both the same, as to enable skipping the middle end's optimizations if desired. We will henceforth refer to these `.json` files as *component* files.

The precise component file schema can be found in Appendix A, but we will roughly outline its requirements here.

The component file contains both information about the current formal logic fragment that the input was constructed from, building upon the header in MONA, and an array of individual components, each representing an AST node. These components make up the AST in prefix notation, where other components are referred to by their index in this "DAG" when needed. Each node contains information about the type of node it is, and any operands it may require, be those constants, variables, or other formulas, represented by a different node. Specifically, they are variables (in the case of components representing what will be atomic automata and predicate operators), a single number literal (for the `MaxSize` and `Plus1` components), array of number literals (for the `Literal` component) or indexes of other, previously constructed automata (in the case of all operators). All operator components also are capable of storing variable renaming schemes for the automata they are constructed out of, provided by the DAGification algorithm (described in Section 4.3). These schemes will naturally be empty for any component files that are constructed by the frontend, as that algorithm is part of the middle end. Two special cases are the `Load` component, whose only operand is the

path to a file containing a BDD to load, and the `Automata` component, whose operand is an object containing states and transitions.

Any variables are already identified by numerical indexes, instead of strings. Those indexes are initially assigned in the same manner as MONA does, that is in order of declaration if they are free, or in order they are encountered when traversing the AST for bound ones.

5.1 Frontend

> *The frontend's job is to take the formal logic formula(s), check them for syntactic validity, and turn them into a representation better suited for further processing – an AST.*

The key new proposed feature of this step is to expand the MONA syntax to enable the ability to define automata within the `.mona` file itself. The following syntax in BNF-like notation defines a new *declaration* for defining an automaton and other key nonterminals (the full proposed syntax can be found in Appendix B):

$$
\begin{array}{lcl}
decl & \rightarrow & \texttt{automaton}\ name\ [(par|con)]^{?}\ \texttt{=}\ \texttt{[}\ [stmnt]^{\circledast}\ \texttt{]} \\
stmnt & \rightarrow & state \mid tran \mid com \mid ifHead\ \texttt{\{}\ stmnt\ \texttt{\}} \\
 & \mid & forHead\ \texttt{\{}\ stmnt\ \texttt{\}} \\
 & & \\
state & \rightarrow & \texttt{<}\ stName[\texttt{\{}int \mid int\ \texttt{,}\ \ldots\ \texttt{,}\ int\texttt{\}}]^{?}\ \texttt{(}stval\texttt{)}\ [\texttt{init}]^{?} \\
tran & \rightarrow & \texttt{>}\ id\ [\ \texttt{default} \mid \texttt{@}\ [ltr]^{\oplus}\ \texttt{@}\ ]\ id \\
 & & \\
com & \rightarrow & \texttt{bool}\ contName\ [\ \texttt{<-}\ bool\ ]^{?} \\
 & \mid & \texttt{var}\ contName\ [\ \texttt{<-}\ \ int\ ]^{?} \\
 & \mid & \texttt{arr}\ contName\ [\ \texttt{<-}\ field]^{?} \\
 & \mid & contVal\ \ \texttt{<-}\ \ [bool|int|field]
\end{array}
$$

The optional *variables* nonterminal is used to define names for local parameter variables that are not logical variables used in the automaton, but rather used in computing the automaton itself, with the string preceding it denoting the order of the variable, which is used for syntactic checks. This string is purposefully different for the one used to denote the order of logical variables.

The nonterminal *states* defines a name and the state for one or more states. The state types are accepting (+), rejecting (−) or don't-care (~). Multiple states can be defined under a single name using the `( )` construct, effectively constructing a state array. If a state array is declared again but with a range disjoint from the previous declaration, the range is extended – other forms of redefinition are syntactic errors. Individual states can later be accessed by the operator used for indexing into a field. The first declared state is also

the default starting state, if no state was marked by `init` after it – then that state is the starting state. If a field is marked, the first state of the field is the starting state.

The nonterminal *stmnt* is used both to define states and their transitions, but also to control which state transitions are constructed by providing variables and control flow similar to a primitive programming language.

The *tran* nonterminal defines the actual state transition from one state to another, referred either by name or by index. The `default` transition defines a transition used for letters without an explicit transition defined, and one default transition must be defined for every state. Otherwise, one or more letters separated by `&` are specified to be used for a specific transition. Only states that have already been declared can be used in transitions.

The nonterminal *ltr*, in combination with the nonterminal *paramval* are used to define the letter for a transition. A letter is defined by a series of pairs of variable and bit value, − for a **0** bit, + for a **1** bit. Any variables not mentioned are automatically assigned a **X** bit for this letter.

Checks provided for this declaration guarantee that any names used are unique, that only declared variables are used for state transitions, that there are no states defined for which there is an unknown state type and only one starting state is present. The definitions of parameters in letter nor state transitions don't have to follow any ordering. Checks for a realized automaton guarantee that the number and order of variables matches, that all states have a defined transition for every letter and that the resulting automaton is deterministic. Some of the checks performed on realized automata are performed by the backend, as they are easiest to detect when constructing the automaton itself. As an example of this automaton syntax, Figure 5.1 contains a definition for an automaton that checks a second-order variable is a specific constant set.

An automata defined with this system can then be parametrized and utilized later in code or exported with the same syntax as a macro or predicate. It is worth noting that these automata "blueprints", unless parametrized by defining all variables, both logical and constructive, do not denote an automaton that can be constructed, and thus will not show up in the frontend's output.

The long-term goal of this addition is to eliminate the need for explicit functions used to construct atomic automata and instead replace them with a generic function that takes in an automata representation in a MONA file and constructs its BDD on demand. This then enables both an easy method to define new atomic automata for the user without relying on complicated importing/exporting workarounds, as well as the existence of a "standard atomic automata library", where the user may simply change the definitions of these automata to facilitate a change to a different formal logic fragment, while retaining the same syntax. The implementation of this step does not require any major changes from MONA's current rendition of these steps beyond those

```
automaton const_set (var2 x, arr C) = [
    var curr <- 1;
    var max_ele <- max C;

    < fail (-);
    > fail default fail;
    < field{0 , ... , max_ele} (-) *;

    for(i in {0, ... , max_ele - 1})[
        > field[i] default fail;
        if(i = C[curr]) [
            > field[i] @ x+ @ field[i + 1];
            curr <- curr + 1 ;
        ];
        if(i ~= C[curr]) [
            > field[i] @ x- @ field[i + 1];
            curr <- curr + 1 ;
        ];
    ];
    < succ (+);
    > succ default succ;


    > field[max_ele] @ x+ @ succ;
    > field[max_ele] @ x- @ fail;
]
```

Figure 5.1: Automaton definition example.

needed to humor the proposed new additions and produce the expected output component file. The two generation passes can be blended into a single pass, but that shouldn't result in significantly faster runtime, as it doesn't eliminate any actual construction steps, just reduces the number of traversals across the AST.

5.2 Middle end

The middle end is responsible for performing various optimizations over the DAG. Therefore, its existence is not strictly required for MONA to function, but it makes the backend significantly faster.

Variable Reordering

The main new feature we'd like to implement in the middle end is an investigation into possible variable reordering. As discussed in Section 3.1, we cannot find an optimal ordering nor approximate it in an acceptable amount of time. Therefore, our hopes turn to greedy and heuristic algorithms. We are looking for relatively cheap algorithms, as technically any ordering is a valid ordering, and BDDs using it will eventually be constructed. Also, we have reasons to suspect that for most formulas, multiple equally "good enough" orderings exist, and thus it is sufficient to find just one from this group. We present three possible greedy algorithms of our own creation, employing the following observations:

1. Variables that show up in the same atomic formulas should be ordered close to each other, as the BDD needs to retain information about the evaluation of the first until the evaluation of the last.

2. Some formulas lead to equal results when the ordering is reversed (notably, this includes all symmetric formulas), but this is not the case for all of them. For example, the formula $i < j$ has a preference for i coming before j, as the resulting BDD has one less node.

All of these algorithms start by constructing a directed weighted clique graph, where nodes are variables, and the weight of an edge is calculated as the number of atomic formulas the two variables are in together. If no such formulas exist, then the edge weight is 0. The goal is to find a Hamiltonian path, which then describes the ordering used in ascending order. If variables are in a formula that favors a specific direction of ordering (like the previously mentioned $<$), the edge(s) that corresponds to this order is increased in weight by some positive integer amount.

Algorithm 1: Max Weight Edge The algorithm simply starts from the highest weighted edge to the lowest and adds it to the resulting path if possible (ie. no loop would form and no branching). This may result in multiple disconnected paths during the construction, but we are guaranteed to have a single path by the end of the construction. This is a relatively fast approach if the edges are stored in a maximum heap.

Algorithm 2: Min-Cut The algorithm uses a combination of finding minimum cuts (min-cut) and a divide and conquer method. First we create a second, undirected graph by collapsing edges between the same vertexes together and summing their weights. Then we find a min-cut, which separates the vertexes into two sets. Then, select the most expensive edge between the sets whose vertexes do not have an edge already selected for the resulting path, unless that vertex is part of a singleton set, and add the edge to the resulting path.

Repeat the process of finding a min-cut and selecting an edge until all vertexes are in singleton sets. We can find a min-cut in polynomial time using the Stoer-Wagner algorithm [21], and we will do at most $V - 1$ iterations of finding the cut, where V is the number of vertexes.

Algorithm 3: Max Weight Vertex This algorithm first calculates three numbers from each vertex – the first is the sum of all weights of edges leading to it (the *to weight*), the second of all weights of all edges leading from it (the *from weight*), and the third is the sum of all weights of all edges (effectively the sum of the first and second number, also the *total weight*). Then, the vertexes are added to the resulting ordering in order of their *total weight*. All vertexes beyond the first are added to the start of the ordering if their *from weight* is higher than their *to weight*, otherwise they are added to the end. In case of equal weights, they can be added at either end.

Antiprenexing

The second, arguably just as important, proposed new addition to the middle end's optimizations is the concept of *antiprenexing*, first described for MONA in [11]. It is the notion of "pushing" predicate operators deeper into formulas, causing their possible state space explosions to trigger sooner on smaller automata, potentially resulting in less damage as they are minimized away. This is done using a new set of rewrite rules, similar to those already described in Section 4.4. First, the primary rules that perform scope narrowing:

$$\exists X : \phi_1 \vee \phi_2 \rightsquigarrow (\exists X : \phi_1) \vee (\exists X : \phi_2)$$

$$\exists X : \phi_1 \wedge \phi_2 \rightsquigarrow \phi_1 \wedge \exists X : \phi_2 \quad \Big| \quad \begin{array}{c}\text{Provided:}\\ X \text{ is not free in } \phi_1\end{array}$$

Second, we use De Morgan's rules as support rules that enable the previous two rules to be used more often.

$$\neg(\phi_1 \wedge \phi_2) \rightsquigarrow \neg\phi_1 \vee \neg\phi_2$$
$$\neg(\phi_1 \vee \phi_2) \rightsquigarrow \neg\phi_1 \wedge \neg\phi_2$$

The third rule is of the form:

$$\exists X : \phi_1 \wedge (\phi_2 \vee \phi_3) \rightsquigarrow \exists X : (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$$

This is the only rule that may be detrimental to apply in some scenarios, as it causes the number of operations performed to increase by one, and the resulting products may be bigger. An example of when not to use this rule is with this formula:

$$A_1 \wedge ... \wedge A_n \wedge (B_1 \vee ... \vee B_n)$$

Repeat applications of this rule would result in the formula:

$$((A_1 \wedge B_1) \vee ... \vee (A_1 \wedge B_n)) \wedge ... \wedge ((A_n \wedge B_1) \vee ... \vee (A_n \wedge B_n))$$

which requires a quadratic number of operations and cannot reuse the large chunk $B_1 \vee ... \vee B_n$.

This problem is solved by only using the rule if the estimated cost of the formula does not exceed some threshold value set at the start. The estimated cost proposed by [11] is the result of two linear functions, one which approximates the minimum size, the other the total weight, with both trained on samples of MONA formulas and their resulting automata sizes. The value of these functions is calculated from the bottom up, with atomic automata assigned minimum sizes and total weights equal to their actual size, and composite automata have estimated minimum sizes and total weights, calculated as the result of a linear function that takes in the size of product space of all operand(s) and optionally number of shared variables between the operands. The intercept for this function is equal to zero, while the coefficients range in values, depending on the operator, from 0.0583 to 1.337.[1] The selected threshold value for these estimations was 5000. These values have been acquired by taking various formula and the sizes of their resulting automata and using linear regression to find a model that best fits the values, and as such they are very open to change.

The last rule is a multi-step process that applies on formulas where multiple existential quantifiers are applied on a large conjunction. The rule works as follows:

[1]The exact weights can be found in the code implementation of the tool the paper's authors created, found at `https://github.com/vhavlena/lazy-wsks`

1. Reorder the quantifiers to be best applied for the following steps.
2. Reorder the topmost conjuncts using laws of associativity and commutativity to enable the last step to have as much effect as possible.
3. Apply the second scope narrowing rule as much as is possible.

In addition to these new rules, we would like our implementation to have its rules less "baked into" the AST structure. Instead of each node class having rules explicitly defined with the idea that only this node class would be applicable for the application of the rule, we aim to create a Rule class that would implement two functions – an `IsApplicable` function that takes in a node that is the root of a formula and checks if that rule can be applied to this node, and a `Modify` function that modifies the node and any other nodes it may rely on according to the rewrite rule it represents. This enables us to experiment with measuring the impact of specific rules by turning them on or off, the impact of the order of application rules, or if it is more desirable to first apply one rule to all nodes or all rules to one node.

5.3 Backend

> *The backend's purpose is to turn the DAG (either an unoptimized version provided straight by the frontend, an optimized one created by the middle end or one provided by the user) into the final output that is a DFA.*

One of the first areas we've identified for improvement is the ability to leverage multithreading to speed up the construction process, and take advantage of the now-commonplace multi-core CPUs. There are two possible places in the process where we could multithread:

1. We can create a thread pool and allocate a thread from it to each automata being constructed. This has the advantage of being very easy to implement, as not much coordination is required between automata constructions, as long as automata wait for construction until the automata they are constructed from finish. The downside of this approach is that the process may require more memory at some point in time than if all automata constructions happened sequentially.
2. We can again create a thread pool and allocate a thread from it when a new state is discovered during construction. This has the benefit of not requiring more memory than sequential mode, as the state would take up the same memory anyways, but would require much more care in proper access to the automata being created. The subsequent wait for mutexes may also cause slowdown, though unlikely to be so significant to provide worse results than sequential mode.

We'd also like to experiment with other methods of minimization, as that is still the operation that takes the longest. The first is an $O(n \log n)$ algorithm that was proposed specifically for MONA [13], but seemingly was never implemented. The second is Brzozowski's algorithm, which inverts the automaton twice, each time only constructing reachable states. This algorithm was tested back in 1997 in [10] and quickly discarded, as it has a worst-case exponential behavior, however, the implementation used back then was not using BDDs to store their automata. This second method was successfully implemented in the prototype and its testing results can be found in Section 6.6.

Implemented Prototype

This chapter goes into detail on a prototype implementation of the backend.

The prototype[1] currently consists of the backend without multithreading, but prepared for its addition. Like the original, our prototype constructs the resulting automaton from the bottom up, with the final BDD representing the resulting automaton. All of the algorithms used in this prototype, except for Brzozowski's minimization, are also used in the original MONA and discussed more in depth in previous chapters, albeit their implementation is different.

6.1 Atomic and Negation Automata

Here we detail the methods used for constructing atomic and negation automata.

The atomic automata are crafted "by hand", with all of them having their own functions to generate them that only take the index(es) of variables they are constructed over. Due to the requirement of variables being resolved in a strictly ascending order, the BDD representations for the same type of automata may differ more than just in the variable indexes, but the automata they represent are still identical (just with a different variable order).

The negation operator is done by copying the original BDD and simply flipping accepting states to rejecting and vice versa.

6.2 Composite Automata Construction

The remaining operators are much more complex and thus require care. Surprisingly, they can be constructed using the same overarching method, which can be extended to seemingly unrelated operations like minimization.

The rough idea is to take an implicit representation of the automata being constructed and make it explicit in the form of a BDD. We create ourselves an oracle, which holds information on the implicit automata, and can be queried for information about states and transitions.

[1] The latest version of the prototype can be found at `https://gitlab.fit.cvut.cz/fialojar/mona-2`

Had we stored our automata using a standard transition table, the process then would only require starting from the starting state, then finding to what states it transitions on which symbols, and repeating the process for newly discovered states while they form. But, since we use BDDs to store state transitions, we have to create these as well. Fortunately, we can query the oracle about information required for individual nodes too. If we give it BDD(s), we can ask it about the deciding variable of a node, the children of a node, or if it is an internal or state node. Using all the queried information from the oracle, we have all we need to explicitly construct the result.

Guide Algorithm

The guide algorithm is used to transform an implicit automaton defined by an oracle into an explicit BDD representation, with Algorithm 6.1 containing pseudocode of it. It is a combination of a Breadth First Search (BFS) algorithm [6] when finding which state nodes to construct, and a Depth First Search (DFS) algorithm [6] when constructing a single state node. The usage of BFS ensures that only reachable state nodes are constructed, and that all state nodes are eventually constructed. We could also use DFS instead of BFS, as it also has these properties. Conversely, the choice of using DFS for the single state construction guarantees that a node is constructed only when the information it relies on is already known and not subject to further changes. The pseudocode in Function 6.2 describes the algorithm recursively, but the actual implementation in our prototype is iterative.

```
   input  : Oracle
   output: Explicit Automaton
1  StatesQueue ← {startState} while StatesQueue not empty do
2  |  currentState ← StatesQueue.front
3  |  StateConstruct(Oracle, currentState, StatesQueue)
4  |  StatesQueue.pop
5  end
```

Algorithm 6.1: Guide Algorithm.

Starting with the start state node, and continuing with further state nodes in a queue, each state node proceeds with the following state construction. We could allocate a node at this point and fill in information about its children later, but this would result in unnecessary allocations. If the node is a leaf node that hasn't been previously encountered, it is constructed and added to the queue of state nodes. The function for all leaf nodes then ends.

For all other nodes, we first ask the oracle for its variable and children, then run the state construction function for them if they aren't constructed.

Then, if internal, the node constructs itself with the children and the variable it receives. If if is a state node, it asks for its state type, then also proceeds with state construction with the variable, children and type, finally removing itself from the queue. The automaton construction ends when the queue is empty.

```
1  Function StateConstruct(Oracle, currentNode, StatesQueue):
2      if currentNode is leaf then
3          if currentNode not yet constructed then
4              construct leaf with values of currentNode
5              StatesQueue.push(currentNode)
6          end
7          return
8      else
9          zeroChild ← get zero child index of currentNode from oracle
10         oneChild ← get one child index of currentNode from oracle
11         if zeroChild isn't constructed then
12             StateConstruct(Oracle, zeroChild, StatesQueue)
13         end
14         if oneChild isn't constructed then
15             StateConstruct(Oracle, oneChild, StatesQueue)
16         end
17         var ← get deciding variable of currentNode from oracle
18         if currentNode isn't a state node then
19             construct current node
20         else
21             type ← get state type of currentState from oracle
22             construct current state
23         end
24     end
```

Algorithm 6.2: StateConstruct Function.

Since only nodes that are considered not constructed proceed with construction, this reduces the number of constructions down to a bare minimum. For internal nodes, checking for an already constructed node is easy, as a node with the same deciding variable and children is always the same node. However, this check does not work for state nodes, as we are unable to easily detect equivalent states. To truly minimize the resulting automaton, a minimization operation is carried out after the construction concludes, which makes heavy use of the fact that it only needs to minimize the states.

Some further optimizations do happen during construction. If both children of an internal node are the same, the child node is used in place of the

parent. If the node being constructed is a state node, it instead "pretends" that it is that child, then asks again for its children and variable. This is because a state must be constructed as a state, and cannot reuse a different node.

6.3 Oracles

> *Now that we have defined the guide algorithm and how it operates, the last missing piece is the format and function of the oracle.*

The above algorithm asks for an as-of-now-undefined oracle that it uses to query about information in the form of nodes from the old BDD(s) we are constructing the new BDD from. The overall representation of these nodes is what distinguishes the construction of propositional operators from predicate operators.

We define a node from an old BDD as *active* in a representation when that node's variable is equal to the variable of the node being constructed. Otherwise, we consider the node *inactive*. In addition, we always consider the deciding variable of a leaf node to be ∞, as to prevent leaf nodes from being considered *active*.

Oracle Interface

All valid oracles must be capable of providing the following interface. They may additionally employ other functions to enable their workings.

Variable For a given node, provide the current variable.

Children For a given node, provide the nodes that are its zero and one child.

Start Node Provide the start state.

State Type For a given state, provide the state type.

Node Type For a given node, answer whether the node is an internal node or a state node.

Node Equality For two given nodes, provide whether they are the same node. Does not cover two state nodes that have the same behavior.

Is Leaf For a given node, answer whether the node is a leaf.

Propositional Operator Oracle

This oracle is used to construct propositional operators, namely and, or, implication and equivalence. It carries out the automata product operation.

Node Format Propositional nodes represent the nodes of a product automaton. They consist of a pair of states, one from each automaton. In implementation the node is an ordered pair of indexes of nodes, the first from the left-hand side BDD and second from the right-hand side BDD.

Variable The current variable is the minimum of the variables of the two nodes from the pair.

Children The children are the children of the node if it is *active*, or the node itself if *inactive*.

Start Node The start node is represented by a pair of start nodes from both BDDs.

State Type The state of a new state node is the result of the propositional operator applied on the pair of nodes, where a node is evaluated as true if it is accepting and false otherwise. For example, a node for an `Implication` component is rejecting if the left-hand side node was accepting and the right-hand side node was rejecting, otherwise it is accepting.

Node Type The node is an internal node if either node from the old BDDs is an internal node, otherwise it is a state node.

Node Equality Two nodes are equal if both nodes from the old BDDs are equal.

Is Leaf Leaf nodes are those nodes where both nodes in the old BDDs are leaf nodes.

Predicate Operator Oracle

This oracle is used to construct predicate operators, namely Exists and For All. It first carries out the automata expansion operation, followed by automata projection and determinization operations at the same time.

Node Format Predicate nodes represent the nodes of a projected automaton. They consist of a set of indexes, representing reached nodes in the old BDD. For the sake of performance, the set is implemented as an array of ordered, unique indexes.

Variable The current variable is the minimum of the variables of the nodes from the set.

Children The children are the children of the node if it is *active*, or the node itself if *inactive*.

Start Node The starting node set is a set only containing the start state.

State Type The state of a new state node depends on the states of the nodes in the set. For the `Exists` component, if at least one of the nodes in the set is accepting, then the new node is accepting, otherwise it is rejecting. For the `ForAll` component, if all of the nodes in the set are accepting, then the new node is accepting, otherwise it is rejecting.

Node Type The node is an internal node if any node from the set is an internal node, otherwise it is a state node.

Node Equality Two nodes are equal if there exists an equality bijection between the sets.

Is Leaf Leaf nodes are those nodes where all nodes in the set are leaf nodes.

Before Construction The old BDD has to be altered to account for the fact that some states should have a different state type, as described in Section 2.3. For the `Exists` component this is the expansion of accepting states, for the `ForAll` component it is instead an expansion of rejecting states. This is done in the same way as in the original MONA with a BFS on a reversed BDD from all nodes of the expanded type at the same time, resulting in a $O(n)$ pre-processing time.

Additional Actions When the current variable is one of the bound variables of the operator, the current node set is replaced by a set where all the active nodes replace themselves with both of their children.

6.4 Minimization

Eager minimization is one of the tools employed by MONA to keep runtime to an acceptable amount, and we follow in its footsteps in that regard.

After the construction of an automaton corresponding to a non-trivial operator (any except for negation), we minimize the result. We employ both the method utilized by MONA, which is a modified version of Moore's algorithm, defined in Section 2.2, and Brzozowski's minimization method. Moore's algorithm has a worst case complexity of $O(n^2)$, it has been proven that in most cases the complexity is no worse than $O(n \log \log n)$ [7]. Brzozowski's is worse off, having a worst case exponential complexity.

Moore's Minimization

Our implementation of Moore's minimization starts by splitting up state nodes into equivalence classes based on their state type. Then, for each state node, we run the StateConstruct function in Algorithm 6.2 described above. This time, the oracle is very simple, representing nodes as only a single index. It returns all information as it is in the BDD being minimized unchanged, with one exception – when asked for a leaf node, it returns the index of the equivalence class the leaf node belongs to.

Once all nodes are constructed, they are compared against each other within the same equivalence class. We compare the indexes of their children and the variable they decide on. We can also compare via state classes, but since those were used for the initial split, they will not yield additional information. We split up any differing nodes into different equivalence classes and repeat the entire process while more classes are created in every cycle. The BDD created in the last cycle is the resulting minimized BDD.

Brzozowski's Minimization

Brzozowski's algorithm is conceptually very simple. It boils down to constructing a *reversed* automaton, for which all transitions of the form $\delta(Q_1, \Sigma) \to Q_2$ are replaced with $\delta(Q_2, \Sigma) \to Q_1$, and the starting and final state(s) are swapped. This results in a nondeterministic automaton with possibly multiple starting states. For the minimization to work, only reachable states are constructed. Reachable states are the start state, and all states for which there exists some other reachable state from which a transition exists to the state. The process of reversing and determinization is repeated twice.

Even without the usage of BDD for storage, this method is potentially expensive, due to the determinization. BDDs throw in another hurdle to overcome – they cannot be simply reversed, as the property of each node deciding on a single variable is broken. This requires us to reconstruct the BDD during the guide algorithm's run, with nodes not always being fully resolved. Instead, if a reversed node decides on at least two different variables, the nodes that were reached on the first variable are marked as active and resolved accordingly, while others are marked as inactive, and copied into partial nodes to be resolved later. In addition, the first reversal forced us to construct the BDD with variables in a descending order, instead of the ascending order used everywhere else so far, as the process would otherwise break down completely.

Our implementation of this minimization method constructs the minimized version with two passes of the Guide Algorithm 6.1, using the result of the first as the input for the second run. The oracle for this minimization implementation is as follows:

Node Format Minimized nodes consist of a set of indexes, representing nodes or their parts in the old BDD. These nodes can be either full, yet to be resolved, or partial, in the process of being resolved. Full nodes differentiate between being a root, a leaf, or neither. Partial nodes contain information on which variable they will fully resolve, as well as to which child to be added. Like with the projection oracle, the sets are implemented as an array of ordered, unique indexes due to performance.

Variable The current variable is either the maximum (for the first run) or minimum (for the second run) of the variables of the nodes.

Children If a node is *inactive*, its children are the node itself. Otherwise, the children of an active full node are full nodes reachable on the active variable, and other reachable nodes of this node are added to both children as partial nodes. Active partial nodes add their node as a full node only to their respective child. If a full node is added and is a state node in the old BDD, it is added as a leaf.

Start Node The starting node set is a set containing all accepting states from the old BDD.

State Type The state of a new state node is accepting if it contains the node at index zero from the old BDD, otherwise it is rejecting.

Node Type The node is a state node if all nodes are full nodes and either all roots or all leaves. Otherwise, it is an internal node.

Node Equality Two nodes are equal if there exists an equality bijection between the nodes.

Is Leaf Leaf nodes are those nodes where all nodes are leaf nodes.

Before Construction The old BDD has to be reversed. This is done by going over all nodes and adding to both child (or one, if it only has one) nodes the index of the parent and which variable it reached them on. Additionally, all nodes then calculate the first variable they will decide on.

6.5 BDD Format

Our BDD representation differs slightly from MONAs. Here we detail the differences.

Instead of three arrays, we condense them into two – the *state* and *node* arrays, with the same function as in MONA. Instead of requiring a transition array to locate the root node, the root node is always found at the same index as the

index of the state. This poses the requirement that all states are defined before any internal node, but allows us to also get rid of dedicated leaf nodes, and instead detect them by checking that the index falls within the start interval that contains states. In addition, the variable index zero has a special meaning of "no variable", and is used for states which have the same transition no matter the letter. Nodes featuring a zero variable index will have both children point to the same index.

6.6 Benchmarking

Now that we have a working prototype backend in hand, we can put it through its paces.

Due to the frontend not being implemented yet, we are unable to conduct a more traditional version of benchmarking against the original MONA. Therefore, we tested our prototype by itself on two categories of atomic automata. Both were motivated by the existence of those beyond the core necessary five, as we asked ourselves: is their explicit construction truly that big of a time save? The resulting automata are linear in size in relation to a constant they're defined with, but constructing them requires usage of many additional variables that need to be projected away. The tests were run on a 64-bit Linux Mint GNU/Linux desktop with AMD(R) Ryzen(R) 5 5600X 6-Core Processor CPU running at 3.8 GHz with 48 GiB of RAM. All time measurements were taken using Linux's `time` utility. A value of N/A denotes that the program did not finish due to running out of memory. For all graphs, the exact values can be found in Appendix C.

Addition

While the increment operation is one of the core atomic formulas, the addition of a constant $c \neq 1$ is not. To construct the formula $x = y + c$, we used two methods – a naive one, which chains together c increment operations, and a smarter one, where previously constructed increments are reused, reaching the desired constant in a logarithmic number of steps rather than linear. If $y = x_0 \wedge x = x_c$, then the naive formula is of the form:

$$\exists x_1, ..., x_{c-1} : \bigwedge_{i=0}^{c-1} (x_{i+1} = x_i + 1)$$

And its eager minimization version is:

$$R_n = \begin{cases} n = 2 & \exists x_1 : x_1 = x_0 + 1 \wedge x_2 = x_1 + 1 \\ n > 2 & \exists x_{n-1} : R_{n-1} \wedge x_n = x_{n-1} + 1 \end{cases}$$

While the smart formula is of the form:

$$R_n = \begin{cases} n = 2 & x_1 = x_0 + 1 \land x_2 = x_1 + 1 \\ n > 2 & R_{n/2} \land R_{n/2}[x_0 \mapsto x_{2^n}, ..., x_{2^n} \mapsto x_{2^{2n}}] \\ & \exists x_1, ..., x_{\log(c)-1} : R_c \end{cases}$$

And its eager minimization version is:

$$R_n(x, y) = \begin{cases} n = 2 & \exists z : z = x + 1 \land y = z + 1 \\ n > 2 & \exists z : R_{n/2}(x, z) \land R_{n/2}(z, y) \\ & R_c(x_0, x_c) \end{cases}$$

For the sake of simplicity, only powers of two were tested. For both of these approaches, we tested a version that applies quantifiers at the end, and a version that eagerly pushes the quantifiers as much inward as is possible, manually mimicking the Antiprenexing method discussed in Section 5.2. Both methods of minimization were also tested.

From the resulting values, we can see that generally eagerly applying operators results in more states created, but the total number of nodes drops significantly, often by an order of magnitude. Surprisingly, Brzozowski minimization outperforms Moore's version on this particular formula. This is likely due to both the final and intermediate automata being in shape very close to what in graph theory would be a path – a series of $c + 1$ states, the first starting the transfer if the smaller variable is detected, the ones in between transferring to the next state if the value for the bigger variable has not been detected yet, with the final state transferring to an always accepting state if the bigger variable is detected. This indicates that it can be preferable in some situations.

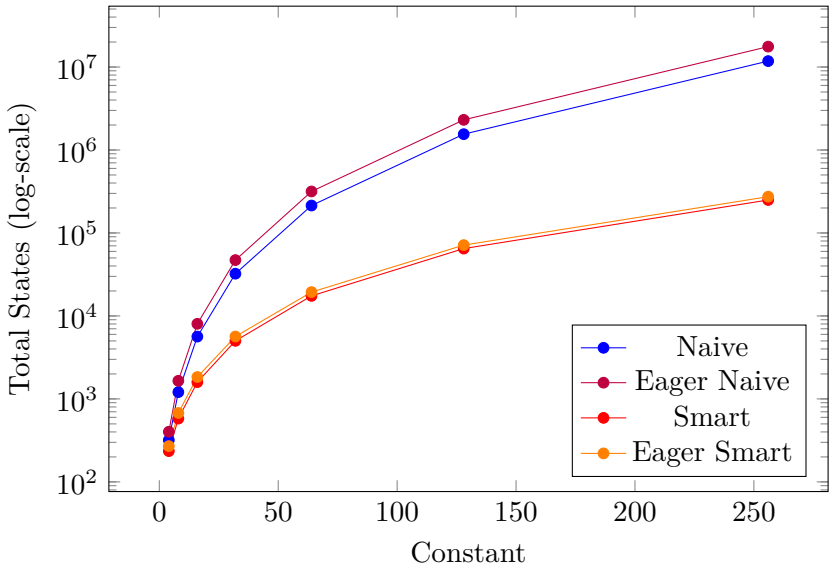


Figure 6.3: Total states for addition, using Moore minimization.

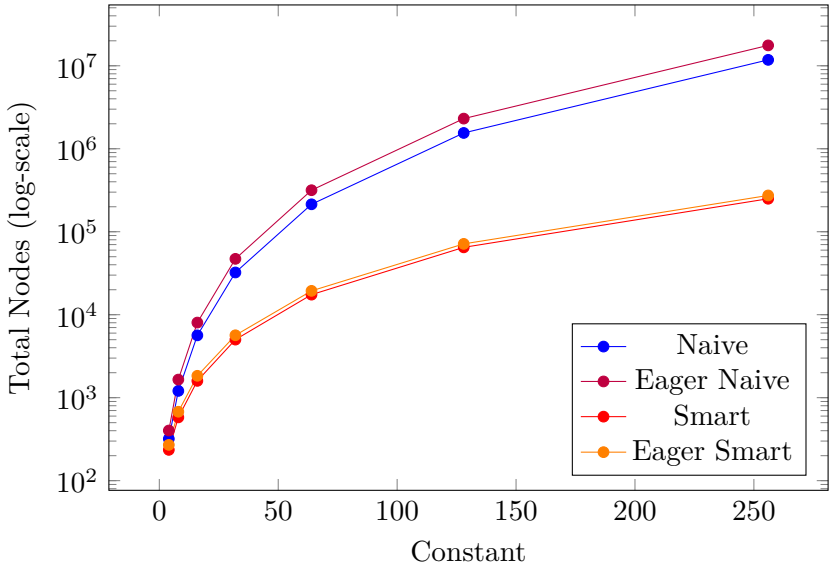


Figure 6.4: Total nodes for addition, using Moore minimization.

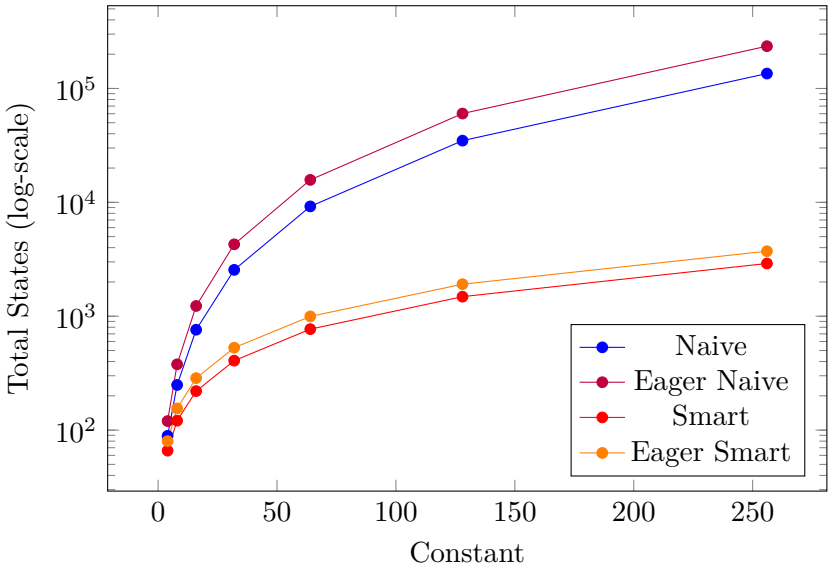


Figure 6.5: Total states for addition, using Brzozowski minimization.

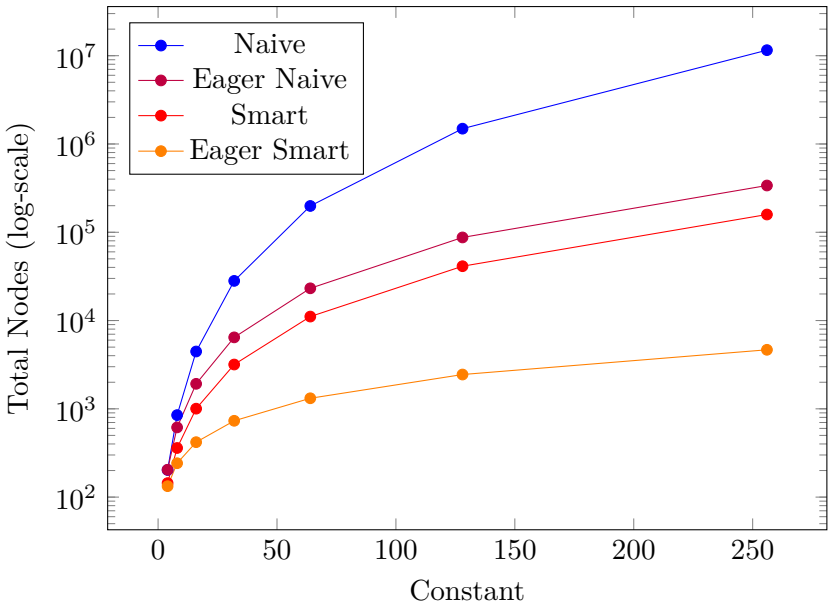


Figure 6.6: Total nodes for addition, using Brzozowski minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	0.002s	0.002s	0.003s	0.003s
8	0.003s	0.003s	0.004s	0.003s
16	0.012s	0.007s	0.009s	0.005s
32	0.079s	0.017s	0.039s	0.007s
64	1.089s	0.115s	0.310s	0.015s
128	16.715s	0.692s	5.083s	0.037s
256	4m 57.107s	5.540s	1m 25.061s	0.094s

Table 6.7: Total duration for addition, using Moore minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	0.003s	0.003s	0.003s	0.003s
8	0.005s	0.004s	0.004s	0.003s
16	0.014s	0.008s	0.006s	0.004s
32	0.059s	0.014s	0.013s	0.004s
64	0.337s	0.049s	0.040s	0.006s
128	3.551s	0.128s	0.143s	0.008s
256	48.357s	0.453s	0.866s	0.016s

Table 6.8: Total duration for addition, using Brzozowski minimization.

Set Size

This operation isn't implemented in the original MONA, nor directly in our prototype, instead an atomic automaton exists to check the maximum size of a set. Once again we employ two methods of constructing the formula $size(X) = c$. The naive one compares all pairs of elements in the set for inequality, while the smart one instead uses the less than operator to cut down the quadratic number of comparisons down to linear. The naive formula is of the form:

$$\exists x_1, ..., x_c : \bigwedge_{i=1}^{c}(x_i \in X) \wedge \bigwedge_{i=1}^{c}\bigwedge_{j=i}^{c}(x_i \neq x_j) \wedge \forall y : y \in X \Rightarrow \bigvee_{i=1}^{c}(y = x_i)$$

And its eager minimization version is:

$$R_n(m) = \begin{cases} n = 1 & \exists x_1 : \bigwedge_{j=2}^{m}(x_1 \neq x_j) \wedge x_1 \in X \wedge \\ & \forall y : y \in X \Rightarrow \bigvee_{i=1}^{m}(y = x_i) \\ n > 1 & \exists x_n : R_{n-1}(m) \wedge x_n \in X \wedge \bigwedge_{j=n+1}^{m}(x_i \neq x_j) \end{cases}$$

$$R_c(c)$$

While the smart formula is of the form:

$$\exists x_1, ..., x_c : \bigwedge_{i=1}^{c}(x_i \in X) \wedge \bigwedge_{i=2}^{c}(x_{i-1} < x_j) \wedge \forall y : y \in X \Rightarrow \bigvee_{i=1}^{c}(y = x_i)$$

And its eager minimization version is:

$$R_n(m) = \begin{cases} n = 1 & \exists x_1 : x_1 \in X \wedge (x_1 < x_2) \wedge \forall y : y \in X \Rightarrow \bigvee_{i=1}^{m}(y = x_i) \\ n > 1 & \exists x_n : R_{n-1}(m) \wedge x_n \in X \wedge (x_n < x_{n+1}) \\ n = c & \exists x_c : R_{c-1}(c) \wedge x_c \in X \end{cases}$$
$$R_c(c)$$

Again, only powers of two were tested, and versions utilizing eager quantifiers and/or Brzozowski minimization were tested.

The results of these tests massively favor Moore's minimization, as the "flipped" automaton BDD likely undergoes a state space explosion. Another interesting thing to note is that when using the smart method of formula construction, eager quantifiers perform worse in all metrics.

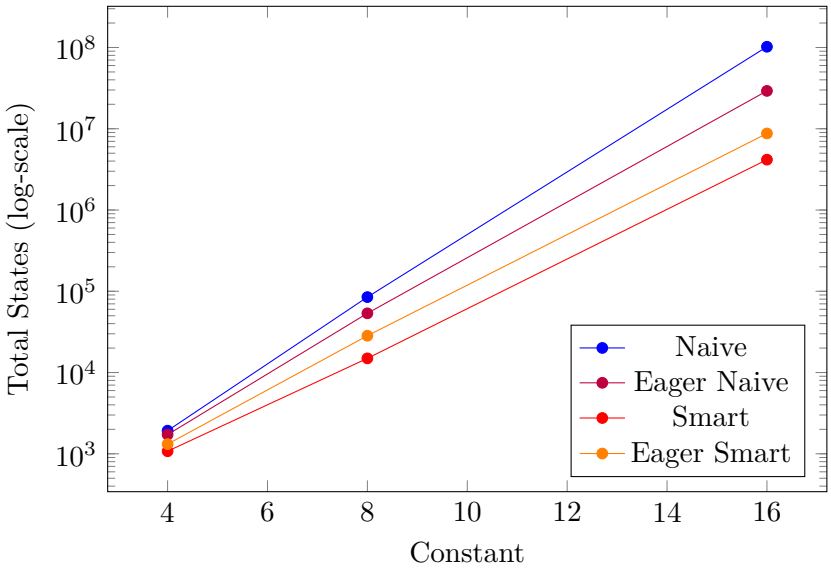


Figure 6.9: Total states for set size, using Moore minimization.

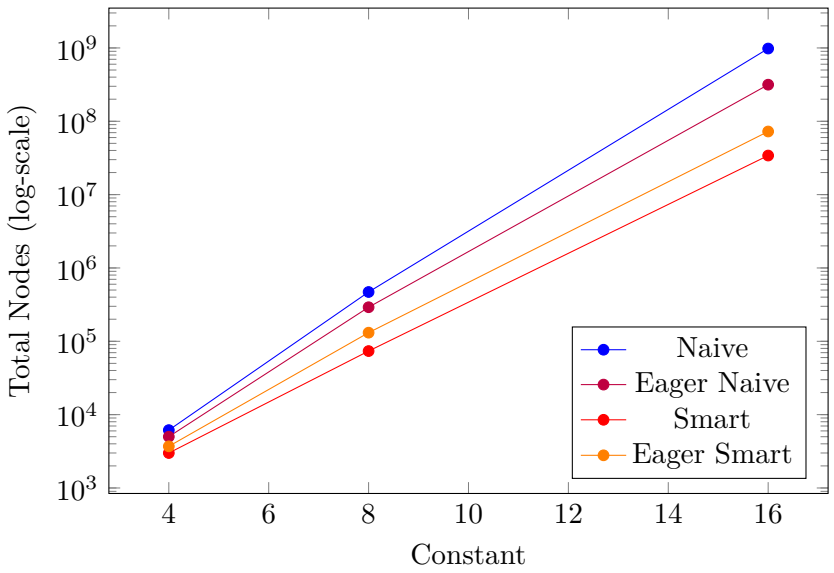


Figure 6.10: Total nodes for set size, using Moore minimization.

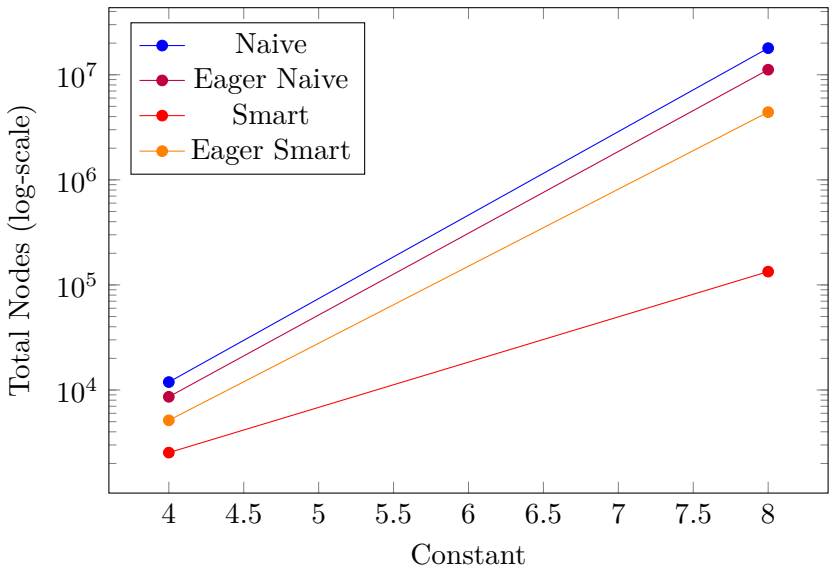


Figure 6.11: Total states for set size, using Brzozowski minimization.

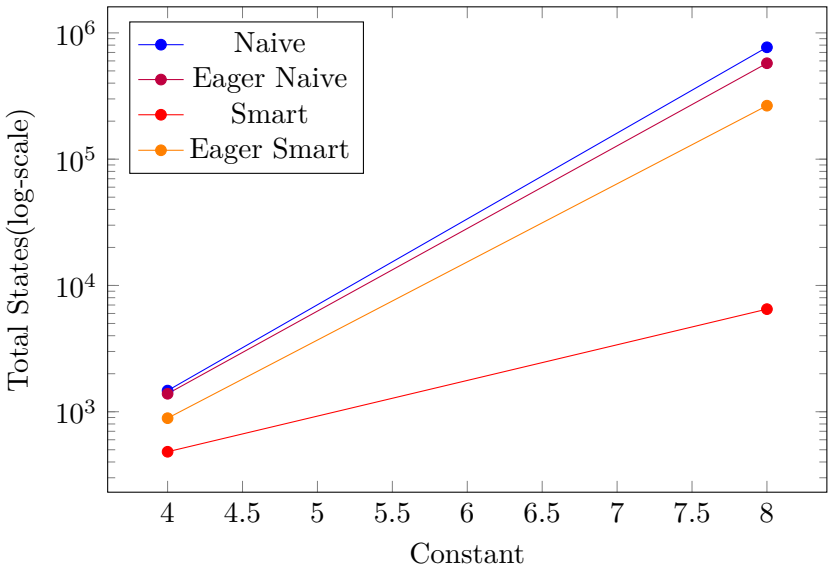


Figure 6.12: Total nodes for set size, using Brzozowski minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	0.009s	0.008s	0.006s	0.007s
8	0.270s	0.158s	0.064s	0.098s
16	24m 12.669s	5m 52.291s	58.862s	1m 45.849s

Table 6.13: Total duration for set size, using Moore minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	0.061s	0.030s	0.017s	0.022s
8	12m 29.360s	6m 26.025s	3.426s	3m 9.987s
16	N/A	N/A	N/A	N/A

Table 6.14: Total duration for set size, using Brzozowski minimization.

Conclusion

The goal of analyzing MONA's current state has been carried out in full for its linear mode. Tree mode has not been studied by this thesis, as that wasn't a goal, but the new improvements we propose can also be applied to it in the future. Since most of the algorithms in use are very basic and very well studied, there wasn't much room for improvement. We did still find some new ones for implementation, especially in the realm of automata minimization. As for the actual implementation, the backend has been recreated in a far cleaner and more modular manner, which is readily extendable.

The most immediate future goal is the full implementation of MONA 2.0, with at least the features described in Chapter 5, to create a fully working tool that can be used in place of MONA. The plans are laid out, thus it is only a matter of putting in the work. After a fully working tool, our efforts can be directed to optimizing and extending the foundation. Secondly, we have only begun to scratch the surface on possible variable reordering methods. Approximation algorithms are a massive field and we have no illusion about our algorithms being the best – they're simply a start. Lastly, we would like to further explore the need for a large number of predefined atomic automata – not from a language point, where their role as syntactic sugar is undisputable, but from the automata construction side. Given the preliminary results observed during our benchmarks, we suspect their inclusion may not be strictly required, but only further research can tell.

Bibliography

[1] David Basin and Nils Klarlund. "Automata based symbolic reasoning in hardware verification". In: *Formal Methods In System Design* 13.3 (1998), pp. 253–286.

[2] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. "Minimization of Automata". In: *CoRR* abs/1010.5318 (2010). arXiv: 1010.5318. URL: http://arxiv.org/abs/1010.5318.

[3] Beate Bollig and Ingo Wegener. "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on computers* 45.9 (1996), pp. 993–1002.

[4] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". In: *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691.

[5] J. Richard Büchi. "Weak Second-Order Arithmetic and Finite Automata". In: *Mathematical Logic Quarterly* 6.1–6 (Jan. 1960), pp. 66–92. ISSN: 1521-3870. DOI: 10.1002/malq.19600060105. URL: http://dx.doi.org/10.1002/malq.19600060105.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[7] Julien David. "Average complexity of Moore's and Hopcroft's algorithms". In: *Theoretical Computer Science* 417 (2012). Mathematical Foundations of Computer Science (MFCS 2010), pp. 50–65. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2011.10.011. URL: https://www.sciencedirect.com/science/article/pii/S0304397511008814.

[8] Calvin C. Elgot. "Decision problems of finite automata design and related arithmetics". In: *Transactions of the American Mathematical Society* 98.1 (1961), pp. 21–51. ISSN: 1088-6850. DOI: 10.1090/s0002-9947-1961-0139530-9. URL: http://dx.doi.org/10.1090/S0002-9947-1961-0139530-9.

[9] Free Software Foundation. *GNU Bison*. URL: https://www.gnu.org/software/bison/.

[10] James Glenn and William Gasarch. "Implementing WS1S via finite automata: Performance issues". In: *International Workshop on Implementing Automata*. Springer. 1997, pp. 75–86.

[11] Vojtech Havlena et al. “Antiprenexing for WSkS: A Little Goes a Long Way.” In: *LPAR*. 2020, pp. 298–316.

[12] John E Hopcroft and Jeffrey D Ullman. *An introduction to automata theory, languages, and computation*. en. Addison-Wesley series in computer science. Upper Saddle River, NJ: Pearson, Jan. 1979.

[13] Nils Klarlund. “An n log n algorithm for online BDD refinement”. In: *International Conference on Computer Aided Verification*. Springer. 1997, pp. 107–118.

[14] Nils Klarlund. “A theory of restrictions for logics and automata”. In: *Computer Aided Verification, CAV '99*. Vol. 1633. LNCS.

[15] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus. Jan. 2001. URL: http://www.brics.dk/mona/.

[16] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. “MONA Implementation Secrets”. In: *International Journal of Foundations of Computer Science* 13.4 (2002), pp. 571–586.

[17] Nils Klarlund et al. *MONA*. URL: https://github.com/cs-au-dk/MONA.

[18] Michael O. Rabin. “Decidability of Second-Order Theories and Automata on Infinite Trees”. In: *Transactions of the American Mathematical Society* 141 (July 1969), p. 1. ISSN: 0002-9947. DOI: 10.2307/1995086. URL: http://dx.doi.org/10.2307/1995086.

[19] Anders Sandholm and Michael I Schwartzbach. “Distributed safety controllers for Web services”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 1998, pp. 270–284.

[20] Detlef Sieling. “The Nonapproximability of OBDD Minimization”. In: *Information and Computation* 172.2 (2002), pp. 103–138. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.2001.3076. URL: https://www.sciencedirect.com/science/article/pii/S0890540101930767.

[21] Mechthild Stoer and Frank Wagner. “A simple min-cut algorithm”. In: *Journal of the ACM* 44.4 (July 1997), pp. 585–591. ISSN: 1557-735X. DOI: 10.1145/263867.263872. URL: http://dx.doi.org/10.1145/263867.263872.

[22] B. A. Trahtenbrot. “Finite automata and the logic of one-place predicates”. In: *Twelve Papers on Logic and Algebra* (1966), pp. 23–55. ISSN: 2472-3193. DOI: 10.1090/trans2/059/02. URL: http://dx.doi.org/10.1090/trans2/059/02.

List of Algorithms, Figures and Tables

List of Abbreviations

AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BFS	Breath First Search
DAG	Directed Acyclic Graph
DFA	Deterministic Finite Automaton
DFS	Depth First Search
FA	Finite Automaton
JSON	JavaScript Object Notation
MSOL	Monadic Second Order Logic
M2L-Str	Monadic Second-order Logic on Strings
NFA	Nondeterministic Finite Automaton
PL	Pumping Lemma
WS1S	Weak Monadic Second-order Theory of One Successor

Contents of Attachments

```
/
├── README.txt ..... description of contents
├── thesis.pdf ..... thesis text in PDF format
├── implementation ..... directory with source code of implementation
├── thesis ..... directory with source code of thesis text
└── other
    └── schema.json ..... ready to use JSON schema from Appendix A
```

MONA 2.0 Component File Schema

The following JSON schema fully establishes our requirements on the JSON component file used to carry information between the different executables of MONA 2.0.

```
"title": "Component",
"type": "object",
"properties":{
    "header":{
        "$ref": "#/$defs/header"
    },
    "components":{
        "type": "array",
        "description": "Individual automata definitions.",
        "items":{
            "$ref": "#/$defs/component",
        },
        "minItems": 1
    }
},
"$defs":{"header":{
    "type": "object",
    "description": "Other required construction information.",
    "properties":{
        "logic":{
            "type": "string",
            "description": "Used formal logic, like ws1s."
        },
        "use_bool":{
            "type": "boolean",
            "description": "Whether to generate the initial
            state that decides only zero-order variables."
        }
    },
    "required": ["logic"]
},
```

```
"component":{
    "type": "object",
    "description": "Single component definition.",
    "properties":{
        "type":{
            "type": "string",
            "description": "Component type. ex. EXISTS"
        },
        "index":{
            "type": "integer",
            "description": "Index of automata component required
                for the NOT operator.",
        },
        "indexes":{
            "$ref": "#/$defs/indexes"
        },
        "variable":{
            "type": "array",
            "description": "Index of variable required for
                an operator.",
        },
        "variables":{
            "$ref": "#/$defs/variables"
        },
        "rename":{
            "$ref": "#/$defs/rename"
        },
        "amount":{
            "type": "integer",
            "description": "Literal value for construction
                (used by MaxSize and Plus1)."
        },
        "literals":{
            "$ref": "#/$defs/literals"
        },
        "path":{
            "type": "string",
            "description": "Path to file location (Load)."
        },
        "automata":{
            "$ref": "#/$defs/automata"
```

```
        }
    },
    "required": ["type"]
},
"indexes":{
    "type": "array",
    "description": "Indexes of automata components required for
        an operator.",
    "items":{
        "type":"integer"
    },
    "minItems": 1
},
"variables":{
    "type": "array",
    "description": "Index(es) of variables required for
        an operator.",
    "items":{
        "type":"integer"
    },
    "minItems": 1
},
"rename":{
    "type": "object",
    "description": "A mapping of variable to another variable.
        Provided alongside any automata component operands.",
    "patternProperties":{
        "^[0-9]*$":{"type":"integer"}
    }
},
"literals":{
    "type": "array",
    "description": "Literal values for construction
        (used by Literal).",
    "items":{
        "type":"integer"
    }
},
"automata":{
    "type": "array",
    "description": "Single automata definition
        (used by Automata).",
```

```
    "items":{
        "$ref": "#/$defs/state"
    },
    "minItems": 1
},
"state":{
    "type": "object",
    "description": "Single state definition.",
    "properties":{
        "state_type":{
            "type":"integer",
            "description": "A state's type.
                Accepting, rejecting or don't-care."
        },
        "default":{
            "type":"integer",
            "description": "Index of state all letters not
                explicitly defined transition to."
        },
        "transitions":{
            "type":"array",
            "description": "Other transitions besides default.",
            "items":{
                "$ref": "#/$defs/state"
            }
        }
    },
    "required": ["state_type", "default"]
},
"transition":{
    "type": "object",
    "description": "Single transition definition.",
    "properties":{
        "destination":{
            "type":"integer",
            "description": "Index of state the transition
                leads to."
        },
        "variables":{
            "type":"array",
            "description": "Values of variables the letter
                must match.",
```

```
            "items":{
                "$ref": "#/$defs/var"
            },
            "minItems": 1
        }
    },
    "required": ["destination", "variables"]
},
"var":{
    "type": "object",
    "description": "Single variable value definition.",
    "properties":{
        "id":{
            "type":"integer",
            "description": "Index of variable to decide on."
        },
        "value":{
            "type":"integer",
            "description": "Whether the variable's bit value
            encountered in the letter must be 0 or 1."
        }
    },
    "required": ["id", "value"]
}}
```

MONA 2.0 Syntax

The overwhelming majority of this syntax is identical to that of MONA in linear mode, but we provide it here in full for completeness. The grammar is described in BNF-like format, with the following meta-syntax:

$[X\|Y]$	either X or Y
$[X]^?$	optional X
$[X]^*$	zero or more repeats of X
$[X]^\odot$	zero or more repeats of X, separated by commas
$[X]^+$	one or more repeats of X
$[X]^\oplus$	one or more repeats of X, separated by commas

B.1 MONA syntax

MONA program

program → $[$*header*`;`$]^?[$*declaration*`;`$]^+$
header → `ws1s`
| `m2l-str`

Declarations

declaration → *fml*
| `include "` *filename* `"`
| `assert` *fml*
| `execute` *fml*
| `const` *c* `=` *I*
| `defaultwhere1 (` *p* `) =` *fml*
| `defaultwhere2 (` *P* `) =` *fml*
| `var0` $[b]^\oplus$
| `var1` $[varwhere1]^\oplus$
| `var2` $[varwhere2]^\oplus$
| `macro` *name*$[params]^?$ `=` *fml*
| `pred` *name*$[params]^?$ `=` *fml*
| `allpos` *P*
| `automaton` *name* $[(par|con)]^?$ `= [` $[stmnt]^\circledast$ `]`

Formulas

$$
\begin{array}{rcl}
fml & \rightarrow & \texttt{true} \mid \texttt{false} \mid \texttt{(}\ fml\ \texttt{)} \\
& \mid & b \mid \texttt{\~{}}\ fml \mid fml_1\ \texttt{\&}\ fml_2 \\
& \mid & fml_1\ \texttt{|}\ fml_2 \mid fml_1\ \texttt{=>}\ fml_2 \mid fml_1\ \texttt{<=>}\ fml_2 \\
& \mid & name[expres]^{?} \mid t_1\ \texttt{=}\ t_2 \mid t_1\ \texttt{\~{}=}\ t_2 \\
& \mid & t_1\ \texttt{<}\ t_2 \mid t_1\ \texttt{>}\ t_2 \mid t_1\ \texttt{<=}\ t_2 \\
& \mid & t_1\ \texttt{>=}\ t_2 \mid T_1\ \texttt{=}\ T_2 \mid T_1\ \texttt{\~{}=}\ T_2 \\
& \mid & T_1\ \texttt{sub}\ T_2 \mid t\ \texttt{in}\ T \mid t\ \texttt{notin}\ T \\
& \mid & \texttt{empty(}\ T\ \texttt{)} \mid \texttt{restrict(}fml\texttt{)} \mid \texttt{prefix(}fml\texttt{)} \\
& \mid & \texttt{ex0}\ [b]^{\oplus}\ \texttt{:}\ fml \\
& \mid & \texttt{all0}\ [b]^{\oplus}\ \texttt{:}\ fml \\
& \mid & \texttt{ex1}\ [varwhere1]^{\oplus}\ \texttt{:}\ fml \\
& \mid & \texttt{all1}\ [varwhere1]^{\oplus}\ \texttt{:}\ fml \\
& \mid & \texttt{ex2}\ [varwhere2]^{\oplus}\ \texttt{:}\ fml \\
& \mid & \texttt{all2}\ [varwhere2]^{\oplus}\ \texttt{:}\ fml \\
& \mid & \texttt{let0}\ [b\ \texttt{=}\ fml]^{\oplus}\ \texttt{in}\ fml \\
& \mid & \texttt{let1}\ [p\ \texttt{=}\ t]^{\oplus}\ \texttt{in}\ fml \\
& \mid & \texttt{let2}\ [P\ \texttt{=}\ T]^{\oplus}\ \texttt{in}\ fml \\
& \mid & \texttt{import(}\ \texttt{"}\ filename\ \texttt{"}\ [v\texttt{->}var]^{\odot}\ \texttt{)} \\
& \mid & \texttt{export(}\ \texttt{"}\ filename\ \texttt{"}\ fml\ \texttt{)}
\end{array}
$$

First-order Terms

$$
\begin{array}{rcl}
t & \rightarrow & p \mid I \mid \texttt{(}\ t\ \texttt{)} \mid t\ \texttt{+}\ I \mid t\ \texttt{-}\ I \\
& \mid & \texttt{max}\ T \mid \texttt{min}\ T \mid t_1\ \texttt{+}\ I\ \texttt{\%}\ t_2 \mid t_1\ \texttt{-}\ I\ \texttt{\%}\ t_2
\end{array}
$$

Second-order Terms

$$
\begin{array}{rcl}
T & \rightarrow & P \mid \texttt{\{}\ [elems]^{\odot}\ \texttt{\}} \mid \texttt{(}\ T\ \texttt{)} \mid \texttt{empty} \\
& \mid & \texttt{pconst(}I\texttt{)} \mid T_1\ \texttt{union}\ T_2 \mid T_1\ \texttt{inter}\ T_2 \mid T_1 \setminus T_2 \\
& \mid & T\ \texttt{+}\ I \mid T\ \texttt{-}\ I
\end{array}
$$

Statements

$$
\begin{array}{lcl}
\mathit{stmnt} & \rightarrow & \mathit{state} \mid \mathit{tran} \mid \mathit{com} \mid \mathit{ifHead}\ \texttt{\{}\ \mathit{stmnt}\ \texttt{\}} \\
 & \mid & \mathit{forHead}\ \texttt{\{}\ \mathit{stmnt}\ \texttt{\}}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{state} & \rightarrow & \texttt{<}\ \mathit{stName}[\texttt{\{}\mathit{int} \mid \mathit{int}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{int}\texttt{\}}]^{?}\ \texttt{(}\mathit{stval}\texttt{)}\ [\texttt{init}]^{?} \\
\mathit{tran} & \rightarrow & \texttt{>}\ \mathit{id}\ [\ \texttt{default} \mid \texttt{@}\ [\mathit{ltr}]^{\oplus}\ \texttt{@}\ ]\ \mathit{id} \\
\mathit{ltr} & \rightarrow & \mathit{paramval} \mid \mathit{ltr}\ \texttt{\&}\ \mathit{paramval}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ifHead} & \rightarrow & \texttt{if (}\ \mathit{bool}\ \texttt{)} \\
\mathit{forHead} & \rightarrow & \texttt{for (}\ \mathit{contName}\ \ \texttt{in}\ \ \mathit{fld}\ \texttt{)}
\end{array}
$$

Automata Commands

$$
\begin{array}{lcl}
\mathit{com} & \rightarrow & \texttt{bool}\ \mathit{contName}\ [\ \texttt{<-}\ \mathit{bool}\ ]^{?} \\
 & \mid & \texttt{var}\ \mathit{contName}\ [\ \texttt{<-}\ \ \mathit{int}\ ]^{?} \\
 & \mid & \texttt{arr}\ \mathit{contName}\ [\ \texttt{<-}\ \mathit{fld}]^{?} \\
 & \mid & \mathit{contVal}\ \ \texttt{<-}\ \ [\mathit{bool}|\mathit{int}|\mathit{fld}]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bool} & \rightarrow & [\ \mathit{contName} \mid \mathit{Be}\ ] \\
\mathit{int} & \rightarrow & [\ \mathit{contName} \mid \mathit{I} \mid \mathit{Ie}\ ] \\
\mathit{fld} & \rightarrow & [\ \mathit{contName} \mid \texttt{\{}[\mathit{int}]^{\odot}\texttt{\}} \mid \texttt{\{}\mathit{int}\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{int}\texttt{\}}\ ]
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Be} & \rightarrow & \texttt{true} \mid \texttt{false} \mid \texttt{(}\ \mathit{bool}\ \texttt{)} \\
 & \mid & \texttt{\~{}}\ \mathit{bool} \mid \mathit{bool}_1\ \texttt{\&}\ \mathit{bool}_2 \mid \mathit{bool}_1\ \texttt{|}\ \mathit{bool}_2 \\
 & \mid & \mathit{bool}_1\ \texttt{=>}\ \mathit{bool}_2 \mid \mathit{bool}_1\ \texttt{<=>}\ \mathit{bool}_2 \mid \mathit{int}_1\ \texttt{=}\ \mathit{int}_2 \\
 & \mid & \mathit{int}_1\ \texttt{\~{}=}\ \mathit{int}_2 \mid \mathit{int}_1\ \texttt{<}\ \mathit{int}_2 \mid \mathit{int}_1\ \texttt{>}\ \mathit{int}_2 \\
 & \mid & \mathit{int}_1\ \texttt{<=}\ \mathit{int}_2 \mid \mathit{int}_1\ \texttt{>=}\ \mathit{int}_2 \mid \mathit{fld}_1\ \texttt{=}\ \mathit{fld}_2 \\
 & \mid & \mathit{fld}_1\ \texttt{\~{}=}\ \mathit{fld}_2 \mid \mathit{int}\ \texttt{in}\ \mathit{fld} \mid \mathit{int}\ \texttt{notin}\ \mathit{fld} \\
 & \mid & \texttt{empty(}\ \mathit{fld}\ \texttt{)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Ie} & \rightarrow & \texttt{(}\ \mathit{int}\ \texttt{)} \mid \mathit{int}_1\ \texttt{+}\ \mathit{int}_2 \mid \mathit{int}_1\ \texttt{-}\ \mathit{int}_2 \\
 & \mid & \mathit{int}_1\ \texttt{*}\ \mathit{int}_2 \mid \mathit{int}_1\ \texttt{/}\ \mathit{int}_2 \mid \mathit{int}_1\ \texttt{\^{}}\ \mathit{int}_2 \\
 & \mid & \mathit{int}_1\ \texttt{\%}\ \mathit{int}_2 \mid \mathit{fld}\ \texttt{[}\ \mathit{int}\ \texttt{]} \mid \texttt{min}\ \mathit{fld} \\
 & \mid & \texttt{max}\ \mathit{fld}
\end{array}
$$

Other

$$
\begin{array}{lcl}
\mathit{elems} & \rightarrow & t \mid t_1\texttt{,}\ \ldots\ \texttt{,}t_2 \\
\mathit{expres} & \rightarrow & \texttt{(}\ [\mathit{exp}]^{\odot}\ \texttt{)} \\
\mathit{exp} & \rightarrow & \mathit{fml} \mid t \mid T \\
\mathit{var} & \rightarrow & b \mid p \mid P \\
 & & \\
\mathit{params} & \rightarrow & \texttt{(}\ [\mathit{par}]^{\odot}\ \texttt{)} \\
\mathit{par} & \rightarrow & \texttt{var0}\ [b]^{\oplus} \\
 & \mid & \texttt{var1}\ [\mathit{varwhere1}]^{\oplus} \\
 & \mid & \texttt{var2}\ [\mathit{varwhere2}]^{\oplus} \\
\mathit{con} & \rightarrow & [\texttt{bool}|\texttt{val}|\texttt{arr}]\ \mathit{name} \\
 & & \\
\mathit{varwhere1} & \rightarrow & p\ [\texttt{where}\ \ \mathit{fml}]^{?} \\
\mathit{varwhere2} & \rightarrow & P\ [\texttt{where}\ \ \mathit{fml}]^{?} \\
 & & \\
\mathit{id} & \rightarrow & \mathit{stName} \\
 & \mid & \mathit{stName}\texttt{[}\mathit{int}\texttt{]} \\
\mathit{contVal} & \rightarrow & \mathit{contName} \\
 & \mid & \mathit{contName}\texttt{[}\mathit{int}\texttt{]} \\
\mathit{paramval} & \rightarrow & \mathit{name}\ [\texttt{+}|\texttt{-}] \\
\mathit{stval} & \rightarrow & [\texttt{+}|\texttt{-}|\sim]
\end{array}
$$

B.2 Restrictions

- b, p, P are names of zeroth-, first- and second-order variables respectively.
- I is an constant integer expression. c is the name of a constant.
- *name* is the name of a macro, predicate or automata. *stName* is the name of a state. *contName* is the name of a control variable.
- Any name is a string of letters, digits, underscores, dollar symbols, and single quotes.
- Variables, macros, predicates and automata must be defined before they are used. If any `defaultwhere` declarations are used, they must be placed before all variable, predicate and macro declarations.
- For macro, predicate and automata invocations, the argument types must match.
- At most one `allpos` declaration is allowed.

- A line comment can be declared with `#` – anything after the symbol until the end of line is ignored. A block comment is denoted as anything between `/*` and `*/`.

B.3 Precedence and Associativity

The table below shows the precedence and associativity of MONA operators, both their logical formula and control statements versions. This denotes in which order the operations are performed, from highest precedence to lowest. If, for example, the operator op_1 has higher precedence (lower precedence number) than operator op_2, then the expression $E_1 op_1 E_2 op_2 E_3$ is interpreted as $(E_1 op_1 E_2) op_2 E_3$. If the precedences are equal, then the interpretation is decided by the associativity – so if op is right-associative, then $E_1 op E_2 op E_3$ is interpreted as $E_1 op (E_2 op E_3)$.

Precedence	Operator	Associativity
1	`()`	not
2	`[]`	not
3	`^`	right
4	`* / %`	left
5	`\`	left
6	`inter`	left
7	`union`	left
8	`min max`	not
9	`= ~= < > <= >=`	not
10	`in notin sub`	not
11	`~`	not
12	`&`	left
13	`|`	left
14	`=>`	right
15	`<=>`	right
16	`:`	not

Benchmark Results

This section contains the actual numerical values measured for the graphs present in Section 6.6.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	319	402	235	270
8	1 207	1 652	579	677
16	5 639	8 040	1 595	1 836
32	32 295	47 120	5 011	5 651
64	214 119	316 512	17 451	19 386
128	1 550 567	2 308 352	64 835	71 393
256	11 784 679	17 609 280	249 691	273 672

Table C.1: Total states for addition, using Moore minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	468	511	338	336
8	2 137	2 023	963	794
16	13 215	9 575	3 340	2 046
32	110 235	55 355	14 725	6 086
64	1 170 003	369 831	80 478	20 502
128	14 723 267	2 693 543	515 063	74 870
256	203 381 475	20 542 375	3 645 200	285 942

Table C.2: Total nodes for addition, using Moore minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	89	120	66	80
8	249	378	121	155
16	761	1 230	220	286
32	2 553	4 278	407	529
64	9 209	15 750	770	996
128	34 809	60 198	1 485	1 911
256	135 161	235 110	2 904	3 722

Table C.3: Total states for addition, using Brzozowski minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	203	204	144	133
8	849	616	360	242
16	4 461	1 920	1 004	419
32	28 133	6 448	3 176	732
64	198 613	23 184	11 092	1 317
128	1 490 869	87 376	41 248	2 446
256	11 550 581	338 640	158 892	4 663

Table C.4: Total nodes for addition, using Brzozowski minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	1 925	1 722	1 075	1 315
8	84 791	53 528	14 939	28 374
16	102 109 051	29 235 188	4 163 803	8 750 317

Table C.5: Total states for set size, using Moore minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	6 152	5 005	2 997	3 698
8	470 102	291 415	73 594	131 090
16	981 373 174	315 511 555	34 111 112	72 570 790

Table C.6: Total nodes for set size, using Moore minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	1 470	1 388	482	889
8	769 312	575 116	6 494	265 236
16	N/A	N/A	N/A	N/A

Table C.7: Total states for set size, using Brzozowski minimization.

Constant	Naive		Smart	
	Lazy	Eager	Lazy	Eager
4	11 896	8 613	2 535	5 145
8	17 920 134	11 170 679	133 821	4 407 834
16	N/A	N/A	N/A	N/A

Table C.8: Total nodes for set size, using Brzozowski minimization.