Mona Reimplemented: WS1S Logic with Mata

Michal Šedý

Email: xsedym02@stud.fit.vutbr.cz;

Abstract

This paper focuses on the reimplementation of the decision procedure for WS1S logic, a second-order logic that can be decided using finite automata. The well-known tool for WS1S logic decision, Mona, employs automata with transitions represented through binary decision diagrams (BDDs). Due to the integration of BDDs into automata, operations like reversal cannot be designed in the conventional manner of reverting individual edges. Instead, the reversal of each BDD must be computed, potentially resulting in an exponential blowup. Motivated by these limitations, Pavel Bednář reimplemented Mona using a pure automata-based approach with the Mata library. This work optimizes this approach, resulting in a significant speedup, up to ten times faster, compared to Bednář's original reimplementation.

Keywords: Finite Automata, Binary Decision Diagrams, WS1S, Mona, Mata

1 Introduction

The most well-known decision procedures are SAT and SMT [1], which are widely used in various applications such as verification (e.g., predicate abstraction), test generation, hardware synthesis, minimization, artificial intelligence, etc. The SAT (satisfiability) problem is a decision problem that asks whether a given propositional formula is satisfiable. The SMT (satisfiability modulo theories) problem extends the SAT problem to the satisfiability of first-order formulas with equality and atoms from many first-order theories. There are also higher-order decision procedures such as WS1S, WS2S, WSkS, S1S, etc.

This work focuses on WS1S, the weak monadic second-order theory of the first successor. The term "weak" refers to finite sets, "monadic" indicates unary relations, "second-order" allows the usage of quantifiers over the relations, and "first successor" means that there is only one successor (e.g., the structure is linear). WS1S [2] has an extremely simple syntax and semantics: it is a variation of predicate logic with first-order variables that denote natural numbers and second-order variables that denote finite sets of natural numbers, it has a single function symbol, which denotes the successor function, and it has comparison operators such as $\leq$, $=$, $\in$, and $\supseteq$. Richard Büchi presented an approach on how to decide WS1S using finite automata in [3]. The main idea is to recursively transform each subformula of the main WS1S formula into deterministic finite automaton (DFA) representing feasible interpretations and to simulate boolean operations via the automata operations.

The most commonly used tool for deciding WS1S and WS2S is Mona[1], which employs Büchi's recursion approach for the construction of finite automata with binary

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

decision diagram (BDD) to represent all automaton transitions. The use of BDD makes the decision faster, but at the cost of making some automata operations, such as reversion, expensive (potentially resulting in exponential blowup). Despite this limitation, Mona is widely utilized in various fields of program verification, including the verification of programs with complex dynamic data structures [4, 5], string analysis [6], parametrized systems [7], distributed systems [8], automatic synthesis [9], hardware verification [10], and many others.

The previously mentioned problem with hard-to-compute automata operations when using BDDs motivated Master's thesis by Bc. Pavel Bednář [11]. He reimplemented Mona's decision of WS1S by using a pure automata-based approach with the Mata automata library[2]. The special type of edge, the *skip edge* has been introduced. The skip edge contains information about how many variables can be jumped over. The primary idea behind introducing the skip edge was to enable jumps not only over inner states (representing the BDD structure) but also over automaton states, with no upper limit on the maximal jump. However, despite this innovation, the skip edges did not produce significant improvements in terms of space or time compression. Furthermore, it appears that skip edges led to an overcomplication of algorithms.

In our approach, we reimplemented Bednář's solution by enhancing each automaton state with an index corresponding to the variable id, mirroring the indexing strategy used for inner nodes in the ordered BDD employed by Mona. This index information allows us to determine the length of a jump based on the indices of the source and destination states in the automaton's transition. Due to the indexing sequence following a pattern of $0, 1, \ldots, n-1, 0, 1, \ldots, n-1, 0, \ldots$, the longest jump can only reach the next state with an index of 0. Although this might appear to be a step backward from Bednář's solution, the limitation on the jump length simplifies all algorithms. Surprisingly, it results in a significantly faster decision of the input formula, up to ten times faster compared to the variant with skip edges.

The second section introduces the definitions of finite automata, binary decision diagram, and WS1S. In the third section, we will dive into different automata representations. The fourth section provides a detailed description of intersection, union, complement, determinization, and minimization of automata with state indexing. Moving to the fifth section, we present a comparison of decision times between the Mona tool, automata with skip edges, and automata with indexes. The experiments are divided into two parts. Initially, automata operations are tested separately on the automata generated during Mona computation. Subsequently, the comparison is executed on the entire input formula.

2 Preliminaries

In this section, we briefly introduce the definitions of nondeterministic and deterministic automata, binary decision diagram, and the WS1S logic.

2.1 Automata

Definition 1. *A deterministic finite automaton is a 5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$*, where its components are:*

- Q *is a finite nonempty set of states,*
- Σ *is an alphabet,*
- $\delta : Q \times \Sigma \to Q$ *is a transition function,*
- $q \in Q$ *is an initial state, and*
- $F \subseteq Q$ *is a set of final states.*

Definition 2. *A nondeterministic finite automaton is a 5-tuple* $M = (Q, \Sigma, \delta, Q_0, F)$*, where* Q*,* Σ*, and* F *are defined identically as for the DFA. The transition function* δ *is defined as* $\delta : Q \times \Sigma \to 2^Q$ *and* $Q_0 \subseteq Q$ *is a nonempty set of initial states.*

[2] available at: https://github.com/VeriFIT/mata

Nondeterminism allows the automaton to make transitions to more than one successor based on the combination of current state and input symbol. In contrast, its deterministic variant can transition to at most one state. Nondeterminism keeps the automaton more compact, but certain operations such as complementation cannot be performed directly. Therefore, determinization is required beforehand.

2.2 Binary Decision Diagram

Representation of the boolean function ϕ with n logical variables leads to 2^n transitions for each automaton state in order to cover every possible combination of logical values. This exponential number of transitions can be reduced using Binary Decision Diagrams (BDDs). Binary Decision Diagrams provide a compact and, most importantly, canonical representation of logical functions of the form $\phi : \{0,1\}^* \to \{0,1\}$.

Definition 3. *The binary decision diagram [12] is rooted, directed, connected, and acyclic graph defined as a 7-tuple* $G = (N, T, var, low, high, root, val)$ *where:*

- N *is a finite set on non-terminal (inner) nodes,*
- T *is a finite set of terminal nodes (leaves) such that* $N \cap T = \emptyset$*,*
- $var : N \to N \cup T$ *defines the low and high successors of the inner nodes,*
- $root \in N \cup T$ *is root node, and*
- $val : T \to \{0,1\}$ *assignes logical values to the leaves.*

The size of the BDD is not determined only by the number of logical variables used within the function ϕ but also by the ordering of the variables in the BDD. The best variable ordering can result in a BDD with a linear (in the number of variables) number of nodes, while the worst ordering can lead to an exponential size.

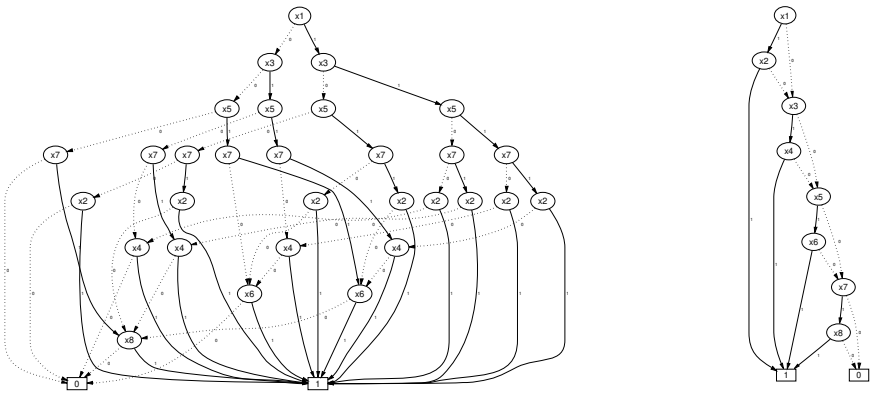


Fig. 1: Two BDDs for the function $f(x1, \ldots, x_{2n}) = x_1x_2 + x_3x_4 + \cdots, x_{2n-1}x_{2n}$ with bad variable ordering (on the left) and good variable ordering (on the right).

Consider the Boolean function $f(x_1, \ldots, x_{2n}) = x_1x_2 + x_3x_4 + \cdots + x_{2n-1}x_{2n}$. Using the variable ordering $x_1 < x_3 < \cdots < x_{2n-1} < x_2 < x_4 < \cdots < x_{2n}$, BDD requires 2^{n+1} nodes to represent the function. Using the ordering $x_1 < x_2 < x_3 < x_4 < \cdots < x_{2n-1} < x_{2n}$, the BDD consists of $2n + 2$ nodes. An example of such orderings is shown in Figure 1. The problem of finding the best variable ordering is NP-hard [13]. However, there are various heuristics to address this challenge [14].

Definition 4. *Let* $\prec$ *be a given ordering on logical variables* Var*, a Binary Decision Diagram (BDD)* G *is ordered (OBDD) with respect to* $\prec$ *if, for every* $n \in \mathbb{N}$*, the following conditions hold:*

1. $low(n) \in N \implies var(n) \prec var(low(n))$
2. $high(n) \in N \implies var(n) \prec var(high(n))$

Definition 5. *The OBDD $G = (N, T, var, low, high, root, val)$ is a Reduced OBDD (ROBDD) if the following conditions are satisfied:*

1. $\forall t_1, t_2 \in T : val(t_1) \neq val(t_2)$
2. *There are no isomorphic subgraphs in G.*
3. $\forall n_1, n_2 \in N : low(n_1) \neq low(n_2) \vee high(n_1) \neq high(n_2)$

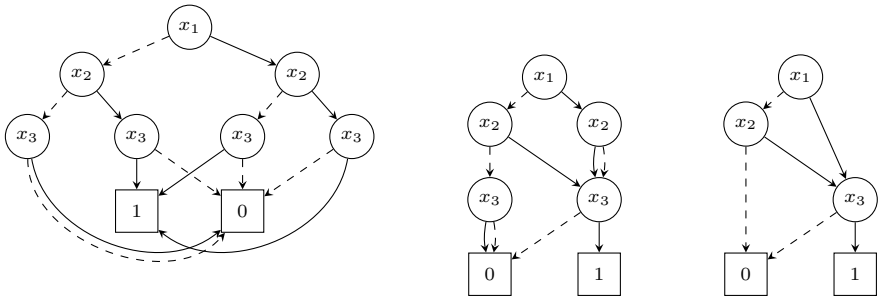


Fig. 2: From left to right, an OBDD satisfying the first, second, and third conditions as specified by the definition of ROBDD.

Theorem 1. *For every boolean function ϕ over some set of variables Var and every variable ordering $\prec$ on Var, there is a unique (up to isomorphism) reduced OBDD (with respect to $\prec$) G_ϕ which represents ϕ. [12]*

Based on Theorem 1, checking the equivalence of two functions, ϕ_1 and ϕ_2, represented by Reduced OBDDs (ROBDDs) G_1 and G_2 is equivalent to checking the isomorphism of G_1 and G_2.

Moreover, if several Boolean functions are represented with one shared ROBDD with multiple roots, as Mona does, the equivalence checking is reduced from isomorphism checking to simply checking the identity of the BDD roots.

2.3 WS1S

Richard Büchi showed, in [3], that WS1S is equivalent to regular expressions and can therefore be represented by finite automat. In this subsection, the simplification of the WS1S formula and its semantics will be presented, followed by the transformation of atomic formulae into automata. The main source for this subsection was [15].

Formula simplification

First-order terms are encoded as second-order terms, since a first-order value can be seen as a singleton second-order value. Also, booleans can be encoded using the first position in the input automaton string.

All second-order terms are "flattened" by introducing new variables that contain the values of all subterms. For example, the formula $A = (B \cup C) \cap D$ will be transformed into the form $\exists V : A = V \cap D \wedge V = B \cup C$, where V is a new variable.

Subformulae are simplified to contain fewer operators. As a result, the solver must implement only basic operations. The abstract syntax for simplified WS1S formulae can be defined by the following grammar:

$$\phi := \neg\phi' | \phi' \wedge \phi'' | \exists P_i : \phi' | P_i \subseteq P_j | P_i = P_j \setminus P_k | P_i = P_j + 1$$

Semantic

Given the main formula ϕ_0, we define its semantic inductively relative to a string w over the alphabet $\mathbb{B}^k$, where $\mathbb{B} = \{0, 1\}$ and k is the number of variables in ϕ_0. Assume that every variable of ϕ_0 is assigned a unique number in the range $1, 2, \ldots, k$, called the *variable index*. The string w now determines an interpretation $w(P_i)$ of P_i,

defined as the finite set $\{j \mid \text{the } j\text{-th bit in the } P_i\text{-track is 1}\}$. For example, the formula $\phi_0 \equiv \exists C : A = B \setminus C$ has variables A, B, and C, which are assigned the indices 1, 2, and 3, respectively. A typical string w over $\mathbb{B}^3$ looks like:

$$\begin{array}{ll} A & \begin{pmatrix}1\\0\\1\end{pmatrix}\begin{pmatrix}0\\0\\1\end{pmatrix}\begin{pmatrix}1\\1\\0\end{pmatrix}\begin{pmatrix}0\\0\\0\end{pmatrix} \\ B & \\ C & \end{array}$$

It's important to note that w with the suffix $(0^*)^T$ defines the same interpretation as w. Therefore, the minimal w is a string where there is no such nonempty suffix. The semantic of a formula ϕ is defined inductively:

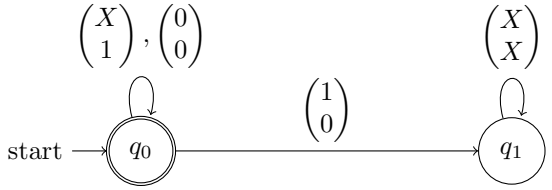
$$\begin{array}{lll} w \vDash \neg\phi' & \text{iff} & w \nvDash \phi' \\ w \vDash \phi' \wedge \phi'' & \text{iff} & w \vDash \phi' \wedge w \vDash \phi'' \\ w \vDash \exists P_i : \phi' & \text{iff} & \exists finite(M) \subseteq \mathbb{N} : w[P_i \mapsto M] \vDash \phi' \\ w \vDash P_i \subseteq P_j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

where we use the notation $w[P_i \mapsto M]$ for the shortest string w' that interprets all variables P_j, $j \neq i$, as w does, but interprets P_i as M. Note that if we assume that w is minimal, then w' decomposes into $w' = w \cdot w''$, where w'' is a string of letters of the form $(0^*X0^*)^T$, and the i-th component is the only one that may be different from 0.

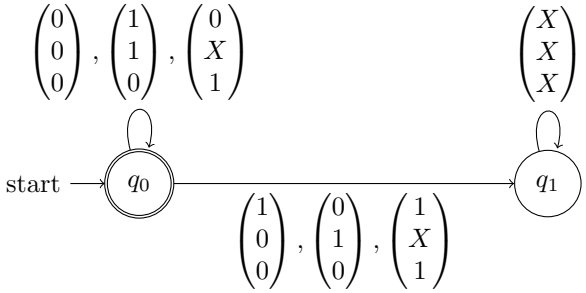
Automata construction

The input formula ϕ is recursively transformed into the deterministic finite automaton that represents the set of satisfying strings $L(\phi) = \{w \mid w \vDash \phi\}$. The translation of atomic and composite formulae to deterministic finite automata follows:

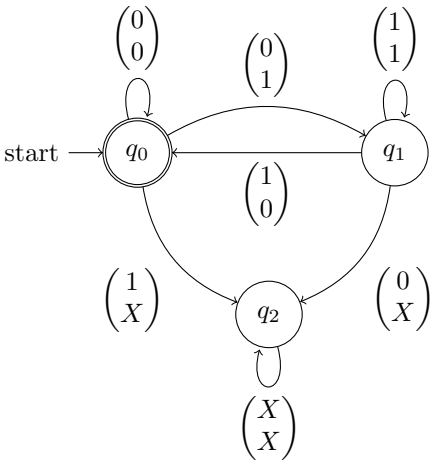
- $\phi = P_1 \subseteq P_2$:



- $\phi = P_1 = P_2 \setminus P_3$:



- $\phi = P_1 = P_2 + 1$:



- $\phi = \neg\phi'$: Negation of a formula corresponds to automaton complementation. If we have already calculated A' such that $L(\phi') = L(A')$, then $L(\neg\phi') = \complement L(\phi') = \complement L(A') = L(\complement A')$, where $\complement$ denotes both language complementation and automata complementation. If the automaton is complete and deterministic, then complementation can be done by swapping accepting and non-accepting states.
- $\phi = \phi' \wedge \phi''$: Conjunction corresponds to language intersection, $L(\phi' \wedge \phi'') = L(\phi') \cap L(\phi'')$. So, the resulting automaton A is obtained by the product of automata $A' \times A''$, where $L(\phi') = L(A')$ and $L(\phi'') = L(A'')$.
- $\phi = \exists P_i : \phi'$: Intuitively, the desired automaton A acts as the automaton A' for ϕ' except that it is allowed to guess the bits on the P_i-track. The resulting automaton A is nondeterministic. It is necessary to apply determinization and adjust the automaton A in such a way that each $w \in L(A)$ is minimal.

3 Automata representation

This section presents three different approaches on how to incorporate BDDs into finite automaton. First, Mona's approach, which uses shared BDDs is shown, followed by approaches that utilize skip edges or state indexing.

3.1 Mona

Mona represents the transition function not only using a single BDD (as in the case of Kripke structures) but with a *shared multi-terminal* BDD (SMTBDD). The main difference from standard BDDs is that the leaves of SMTBDD do not contain boolean values 0 or 1 but rather states of the automaton.

Definition 6. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be DFA. The SMTBDD is defined as a 7-tuple $G = (N, T, var, low, high, R, val)$ where:*

- *N is finite set on non-terminal (inner) nodes,*
- *T is finite set of terminal nodes (leaves) such that $N \cap T = \emptyset$,*
- *$var : N \to N \cup T$ defines the low and high successors of the inner nodes,*
- *$R \subseteq N \cup T$ is nonempty set of roots, and*
- *$val : T \to Q$ assigns states to the leaves.*

For example, let the formula $\phi \equiv \exists p, q : p \neq q \wedge p \in X \cap Y \wedge q \in X \cap Y$ be represented by the automaton M in Figure 3, where each state r, s, and t contains information about whether it is accepting or not and points to its root node in the SMTBDD describing its transition relation. The data structure also contains a pointer to the initial state.

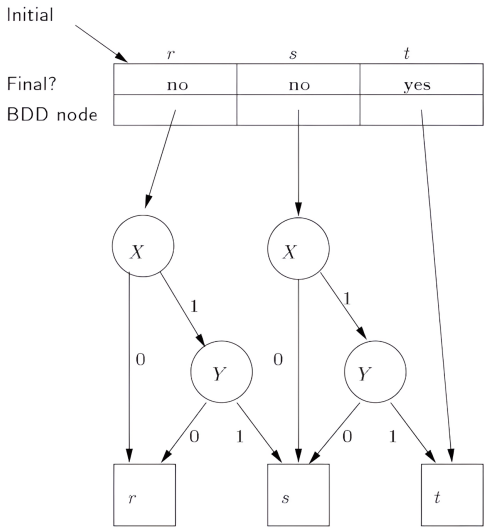


Fig. 3: Mona's automaton for the formula $\phi \equiv \exists p, q : p \neq q \wedge p \in X \cap Y \wedge q \ \in X \cap Y$.

3.2 Automaton with Skip Edges

The main idea of Bednář's skip edges, presented in [11], is to integrate BDD nodes directly into the transitions of the automaton. This can be easily done as the BDD has an automata-like structure. To simulate the functionality of the BDD, where each node has an assigned variable, it is necessary to use skip edges.

Definition 7. *Let $M = (Q, \Sigma, \delta, Q_0, F)$ be an NFA, then the automaton with skip edges $N = (Q, \Sigma, \delta', Q_0, F)$ has a transition function defined as $\delta' : Q \times \Sigma \to 2^{\mathbb{N} \times Q}$.*

The transition function in the automaton with skip edges provides information about the target states and the number of variables that have been jumped over. The skip edge with a length of 1 is a special case that has the same functionality as the edge in NFA. $(n, r) \in \delta'(q, a)$ denotes that the automaton can move from state q to r after reading a symbol a and then jump over $n - 1$ variables.

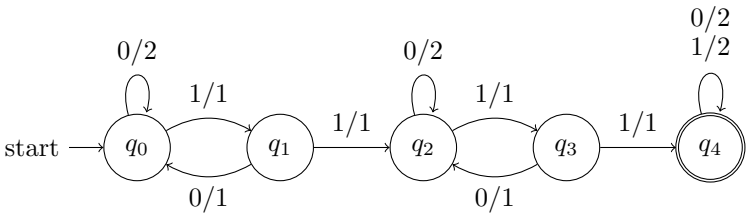


Fig. 4: An automaton with skip edges representing the language from Figure 3. A skip edge labeled with a/n can be interpreted as a transition of length n over a symbol a.

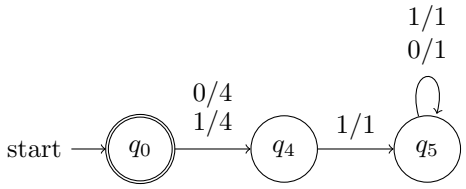


Fig. 5: Automaton with skip edges and a variable X representing formula $3 \in X$.

The potential benefit of skip edges is derived from the fact that its length has no upper limit. Therefore, the skip edge can traverse over many BDDs and automaton states. The demonstration of this benefit is shown in Figure 5. However, it should be mentioned that this approach overcomplicated algorithms and did not show improvement in space or time.

3.3 Automaton with State Indexing

The approach of an automaton with state indexing has been the main focus of this reimplementation. Such an automaton integrates BDDs directly while maintaining the indexes on its nodes (automaton states).

Definition 8. *Let $M = (Q, \Sigma, \delta, Q_0, F)$ be an NFA. The automaton M is called an automaton with state indexing if there exists an index function $\iota : Q \to \mathbb{N}_0$ such that the following conditions hold:*

1. $\forall q \in Q_0 : \iota(q) = 0$
2. $\forall q \in F : \iota(q) = 0$
3. $\forall q, r \in Q : \nexists a \in \Sigma : r \in \delta(q, a) \wedge \iota(r) \neq 0 \wedge \iota(q) \geq \iota(r)$

The first and second conditions reflect the fact that only roots or leaves in the BDD can be initial and final states, respectively. The third condition demands that the part of the automaton simulating the BDD must be acyclic, with the exception of the root nodes/states with index 0.

The index information determines the length of a jump based on the indices of the source and destination states in the automaton's transition. Due to the indexing sequence following a pattern of $0, 1, \ldots, n-1, 0, 1, \ldots, n-1, 0, \ldots$, the longest jump can only reach the next state with an index of 0. Although it might seem like a step back from Bednář's method, the restriction on jump length actually simplifies all algorithms. Interestingly, this leads to a noticeably faster evaluation of the input formula, up to 10 times faster when compared to the version incorporating skip edges.

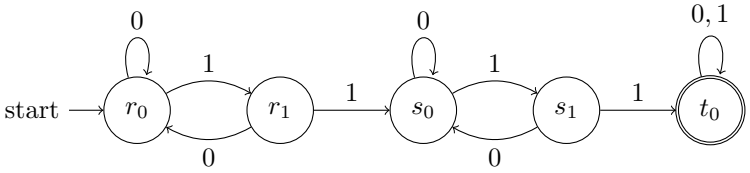


Fig. 6: Automaton with state indexing representing the language from Figure 3.

4 Automata operations

This section presents operations on automata with state indexing, which are utilized in the reimplementation of Mona using the Mata library. The primary concept of these operations is derived from Mata and modified to adapt state indexing.

4.1 Complement

The complementation method is the most straightforward operation on automata. It involves taking a complete deterministic automaton and swapping its accepting and nonaccepting states. Unlike classical complementation, only the accepting property of states with index 0 is altered.

Definition 9. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a complete DFA with an index function $\iota : Q \to \mathbb{N}_0$. The complement of M is defined as $\complement M = (Q, \Sigma, \delta, q_0, F')$, where:*

$$F' = \{q \in Q \mid \iota(q) = 0 \wedge q \notin F\}$$

4.2 Intersection

The standard intersection algorithm generates the resulting automaton using product construction. It begins with the pair of initial states (one from each automaton) and moves synchronously to the pair of successors based on the transition symbol. The key idea is inspired by the *Apply* method [16], commonly used in BDD operations. In this adaptation of the standard intersection algorithm, the successor pair for the analyzed product pair is generated based on the state (or states in the case of identical indices) with the lowest index. Meanwhile, the state with the higher index is duplicated.

Algorithm 1: An Intersection Algorithm

Data: Two DFA $M = (Q_M, \Sigma, \delta_M, i_M, F_M)$ and $N = (Q_N, \Sigma, \delta_N, i_N, F_N)$ and tow index functions $\iota_M : Q_M \to \mathbb{N}_0$ and $\iota_N : Q_N \to \mathbb{N}_0$
Output: Resulting DFA $A = (Q \subseteq Q_M \times Q_N, \Sigma, \delta, i, F) = M \cap N$ and index function $\iota : Q \to \mathbb{N}_0$

Procedure Index(s_M, s_N)
  if $\iota_N(s_N) = 0$ **then**
    return $\iota_M(s_M)$
  if $\iota_M(s_M) = 0$ **then**
    return $\iota_N(s_N)$
  return $min(\iota_M(s_M), \iota_N(s_N))$

Procedure CreateState(x, y)
  $worklist \leftarrow worklist \cup \{(x, y)\}$
  $Q \leftarrow Q \cup \{(x, y)\}$
  $\iota((x, y)) \leftarrow Index(x, y)$
  if $x \in F_M \wedge y \in F_N$ **then**
    $F \leftarrow F \cup (x, y)$

$i \leftarrow (i_M, i_N)$
$\iota(i) \leftarrow 0$
$worklist \leftarrow \{i\}$
$Q \leftarrow \{i\}$
$F \leftarrow \emptyset$
while $worklist \neq \emptyset$ **do**
  $s \leftarrow worklist.pop()$
  $(s_M, s_N) \leftarrow s$
  if $\iota_M(s_M) = \iota_N(s_N)$ **then**
    forall $a \in \Sigma \cap \Sigma : \delta_M(s_M, a) \in Q_M \wedge \delta_N(s_N, a) \in Q_N$ **do**
      $\delta(s, a) \leftarrow (\delta_M(s_M, a), \delta_N(s_N, a))$
      $CreateState(\delta_M(s_M, a), \delta_N(s_N, a))$
  else if $(\iota_M(s_M) < \iota_N(s_N) \wedge \iota_M(s_M) \neq 0) \vee \iota_N(s_N) = 0$ **then**
    forall $a \in \Sigma : \delta_M(s_M, a) \in Q_M$ **do**
      $\delta(s, a) \leftarrow (\delta_M(s_M, a), s_N)$
      $CreateState(\delta_M(s_M, a), s_N)$
  else
    forall $a \in \Sigma : \delta_N(s_N, a) \in Q_N$ **do**
      $\delta(s, a) \leftarrow (s_M, \delta_N(s_N, a))$
      $CreateState(s_M, \delta_N(s_N, a))$
return A, ι

4.3 Determinization

The standard determinization algorithm constructs the resulting automaton through the subset construction. It initiates the process with a set of initial states and creates transitions to the set of targets, which consists of successors of states from the examined set. This method requires modification for compatibility with the index function,

similarly to the intersection algorithm. Specifically, the set of successors of the examined state (representing the set of states) is formed by the states with the lowest index, while the remaining states are duplicated.

Algorithm 2: A Determinization Algorithm

Data: A NFA $M = (Q_M, \Sigma, \delta_M, I_M, F_M)$ and an index function $\iota_M : Q_M \to \mathbb{N}_0$
Output: A DFA $A = (Q \subseteq 2^{Q_M}, \Sigma, \delta, i, F)$ such that $L(M) = L(A)$ and an index function $\iota : Q \to \mathbb{N}_0$

```
1  i ← I_M
2  Q ← {i}
3  F ← ∅
4  ι(i) ← 0
5  worklist ← {i}
6  while worklist ≠ ∅ do
7      s ← worklist.pop()
8      waiting ← {q ∈ s | ι_M(q) ≠ ι(s)}
9      cont ← s \ waiting
10     symbols ← {a ∈ Σ | ∀q ∈ cont : δ(q, a) ≠ ∅}
11     forall a ∈ symbols do
12         s_next ← waiting ∪ ⋃_{q∈cont} δ(a, q)
13         δ(s, a) ← s_next
14         Q ← Q ∪ {s_next}
15         worklist ← worklist ∪ {s_next}
16         if ∀q ∈ s_next : ι_M(q) = 0 then
17             ι(s_next) ← 0
18         else
19             ι(s_next) ← min({q ∈ s_next | ι_M(s) ≠ 0})
20         if ι(s_next) = 0 ∧ F_M ∩ s_next ≠ ∅ then
21             F ← F ∪ {s_next}
22     if waiting ≠ ∅ then
23         forall a ∈ Σ \ symbols do
24             δ(s, a) ← waiting
25             Q ← Q ∪ {waiting}
26             worklist ← worklist ∪ {waiting}
27             if ∀q ∈ waiting : ι_M(q) = 0 then
28                 ι(waiting) ← 0
29             else
30                 ι(waiting) ← min({q ∈ waiting | ι_M(s) ≠ 0})
31             if ι(waiting) = 0 ∧ F_M ∩ waiting ≠ ∅ then
32                 F ← F ∪ {waiting}
33 return A, ι
```

4.4 Union

The only distinction between the union of two NFAs and the union of two NFAs with index functions is the necessity of unifying the index functions as well.

Definition 10. *Let $M = (Q_M, \Sigma_M, \delta_M, I_M, F_M)$ be the first NFA with index function $\iota_M : Q_M \to \mathbb{N}_0$ and $N = (Q_N, \Sigma_N, \delta_N, I_N, F_N)$ be the second NFA with an index function $\iota_N : Q_N \to \mathbb{N}_0$, where $Q_M \cap Q_N = \emptyset$. The union of M and N is an NFA $A = (Q_M \cup Q_N, \Sigma_M \cup \Sigma_N, \delta, I_M \cup I_N, F_M \cup F_N)$ where:*

$$\delta(a, q) = \begin{cases} \delta_M(a, q) & q \in Q_M \\ \delta_N(a, q) & q \in Q_N \end{cases}$$

with an index function $\iota : Q_M \cup Q_N \to \mathbb{N}_0$ *defined as:*

$$\iota(q) = \begin{cases} \iota(q) & q \in Q_M \\ \iota(q) & q \in Q_N \end{cases}$$

4.5 Projection

The basic idea of a projection is to remove all transitions going from the states with the index of the variable being removed and for every such a state to redirect all incoming edges into its successors. The only exception is the projection of the variable with index 0. In that case, the redirection cannot be done due to the restriction that states with index 0 cannot be jumped over. The solution to this problem is to create an edge for every letter of the alphabet and for every successor of the examined state.

Algorithm 3: A Projection Algorithm

Data: A NFA $M = (Q_M, \Sigma, \delta_M, I, F)$, an index function $\iota_M : Q_M \to \mathbb{N}_0$, and $id \in \mathbb{N}_0$ of a variable being projected.
Output: A NFA $A = (Q \subseteq Q_M, \Sigma, \delta, I, F)$ and new index function $\iota : Q \to \mathbb{N}_0$

```
1  A ← M
2  if id = 0 then
3      ι ← ι_M
4      forall q ∈ Q_M do
5          if ι_M(q) ≠ id then
6              continue
7          succ ← {r ∈ Q_M | ∃a ∈ Σ : r ∈ δ_M(q, a)}
8          forall a ∈ Σ do
9              δ(q, a) ← succ
10     return A, ι;
11 forall q ∈ Q_M do
12     if ι_M(q) ≠ id then
13         continue
14     succ ← {r ∈ Q_M | ∃a ∈ Σ : r ∈ δ_M(q, a) ∧ ι_M(r) ≠ id}
15     forall a ∈ Σ, r ∈ Q_M : q ∈ δ_M(r, a) ∧ ι_M(r) ≠ id do
16         δ(r, a) ← δ(r, a) ∪ succ
17     A.remove(q)
18 forall q ∈ Q do
19     if ι_M(q) > id then
20         ι(q) = ι_M(q) − 1
21     else
22         ι(q) = ι_M(q)
23 return A, ι
```

After the projection of the variable. There exist paths over symbol 0 leading from nonaccepting states into accepting. This changes the language represented by the automaton. To repair the language, its necessary to mark every such state as accepting with the exception to the initial state. The automaton $M = (Q, \Sigma, \delta, I, F_M)$ is transformed into the automaton of the form $A = (Q, \Sigma, \delta, I, F)$, where the set of final states is defined as $F = F_M \cup \{q \in Q \setminus I \mid \text{exist path over 0 leading to } f \in F_M\}$.

4.6 Revert

The standard reversion algorithm switches the initial and final states of the automaton and flips the direction of all transitions. This practice can be used on automata with indexed states only when there is a single variable. Otherwise, each transition of length

$1 < n$ over a symbol $a \in \Sigma$ must be reverted and divided into two sections. The first section contains transitions of length $n-1$ over all symbols of Σ and the second section consists of the transition of length 1 over the symbol a.

Definition 11. *Let $n \in \mathbb{N}$ be the number of variables and $\iota : Q \to \mathbb{N}_0$ be the index function. The length of a transition between states $q, r \in Q$ over a letter $a \in \Sigma$ is determined as:*

$$len(q \xrightarrow{a} r) = \begin{cases} \iota(r) - \iota(q) & \iota(r) \neq 0 \\ n - \iota(q) & otherwise \end{cases}$$

Definition 12. *Let $M = (Q_M, \Sigma, \delta_M, I, F)$ be NFA, $n \in \mathbb{N}$ be the number of variables used in the automaton, and $\iota : Q_M \to \mathbb{N}_0$ be index function. The reverted automaton is defined as $A = (Q := Q_N \cup Q_M, \Sigma, \delta, I, F)$ with index function $\iota : Q \to \mathbb{N}_0$ where:*

- $Q_N = \{(q, a, r) \in Q_M \times \Sigma \times Q_M \mid r \in \delta(q, a) \wedge len(q \xrightarrow{a} r) \neq 1\}$ *is a set of new states.*
- $\delta(q, a) = \begin{cases} \{r \in Q_M \mid q \in \delta_M(r, a) \wedge len(r \xrightarrow{a} q) = 1\} \cup & \\ \quad \{(r, b, q) \in Q_N \mid \exists b \in \Sigma : q \in \delta_M(r, b)\} & q \in Q_M \\ \{(r, a, q) \in Q_N \mid r \in \delta(a, q)\} & otherwise \end{cases}$
- $\iota(q) = \begin{cases} n - 1 - \iota_M(q) & q \in Q_M \\ \iota_M(q') + 1 & q := (q', a, r') \in Q_N \end{cases}$

4.7 Minimization

The fast minimization algorithm for incomplete deterministic automata [17] that performs in $\mathcal{O}(m \log n)$ time has been implemented. Unfortunately, benchmarks have revealed that this algorithm somehow does not produce the minimal form of the automaton when using state indexing.

Since this algorithm is unsuitable for automata minimization and the minimization algorithm based on the simulation provided by the Mata library is too slow, the naive Brzozowski algorithm [18] has been chosen. This algorithm minimizes DFA by reverting and determinizing the input automaton, and then reverting and determinizing it again.

Automata minimization performed by the Brzozowski algorithm corresponds to the elimination of isomorphic subgraphs in BDD reduction. Eliminating nodes with equivalent *high* and *low* successors can be also performed on automata by simply redirecting all incoming transitions from such a state to all its successors. The only exception is the elimination of states with index 0. Those states will have to remain in the automaton even if their *high* and *low* successors are the same state.

Definition 13. *Let $M = (Q, \Sigma, \delta, i, F)$ be the DFA with index function $\iota : Q \to \mathbb{N}_0$. The automaton M is called reduced DFA if it is in minimal form and $\forall q \in Q$: $\iota(q) = 0 \vee (\nexists r \in Q \setminus \{q\} : \forall a \in \Sigma : r = \delta(q, a))$.*

5 Experimental results

This section is dedicated to comparing the performance of Mona, its reimplementation using skip edges, and its reimplementation using state indexing. The experiments were performed on a machine equipped with an AMD Ryzen 7 3800XT 8-Core processor and 32 GB of memory. The comparison begins with an analysis of intersection and projection operations, with the focus on execution times. Subsequently, the focus shifts to the complex evaluation of all WS1S formulae. The benchmark formulae have been sourced from the Gaston project[3], and Bednář's Master thesis [11].

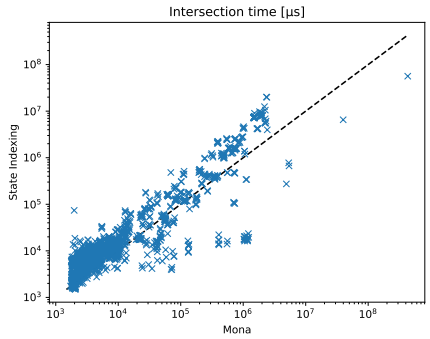
[3] available at: https://www.fit.vutbr.cz/research/groups/verifit/tools/gaston/.cs

5.1 Operations performance

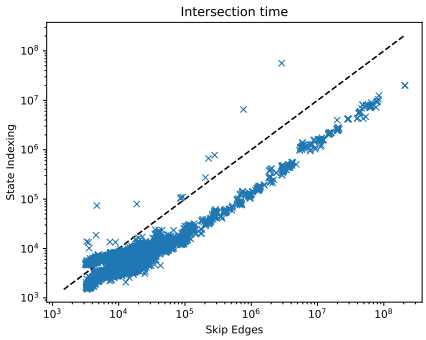
Individual operations, such as intersection and projection, have been tested on automata generated during the decision of the WS1S procedures from [11]. Complementation was omitted from testing due to its simplicity, and minimization was excluded since the Brzozowski algorithm is widely known for its slow performance. Determinization was not tested because the Mona tool does not produce nondeterministic automata. However, the impact of determinization can be indirectly observed during the projection of the first variable. Because each projection must be followed by a determinization, and by projecting the first variable, the automaton attains the highest level of nondeterminism.

Intersection

Figures 7 and 8 reveal that the State Indexing reimplementation was, on average, approximately ten times faster than the Skip Edges. When comparing State Indexing to the Mona tool, it becomes evident that Mona is still faster, on average, than State Indexing. However, there are instances where State Indexing surpassed Mona by more than 50 times in speed. Additionally, State Indexing performs intersection at its worst case (without outliers) only up to ten times slower than Mona.



Comparing State Indexing and Mona: logarithmic time performance [μs].



Comparing State Indexing and Skip Edges: logarithmic time performance [μs].

Fig. 7: Comparison of intersection time for the tool Mona, reimplementation using Skip Edges, and reimplementation using State Indexing.

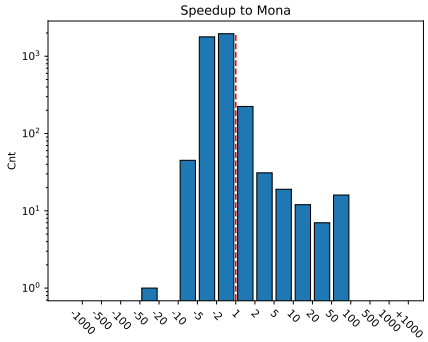
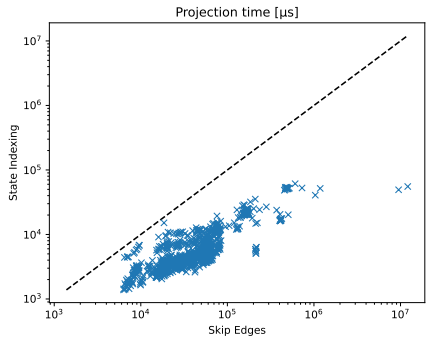


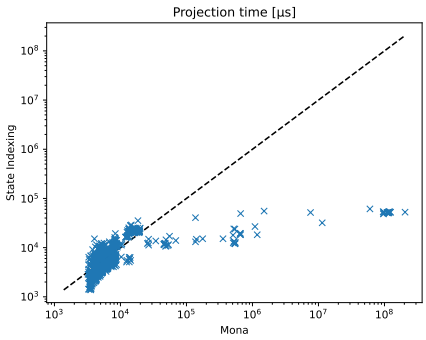
Fig. 8: State Indexing speedup compared to Mona intersection time. The vertical axis represents the number of occurrences (logarithmic scale), while the horizontal axis indicates the speedup (positive) or slowdown (negative).

Projection of the last variable

It is interesting to observe, as depicted in Figures 9 and 10, that the reimplementation with State Indexing was, on average, faster not only than the Skip Edges version but also than Mona. For certain inputs, State Indexing proved to be more than a thousand times faster than the tool Mona. Despite this, the maximal slowdown for State Indexing was only a factor of 5.



Comparing State Indexing and Mona: logarithmic time performance [μs].



Comparing State Indexing and Skip Edges: logarithmic time performance [μs].

Fig. 9: Comparison of projection and determinization time for the tool Mona, reimplementation using Skip Edges, and reimplementation using State Indexing.

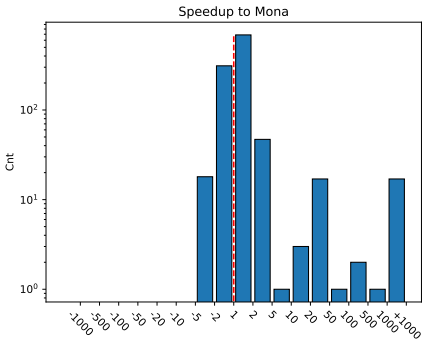
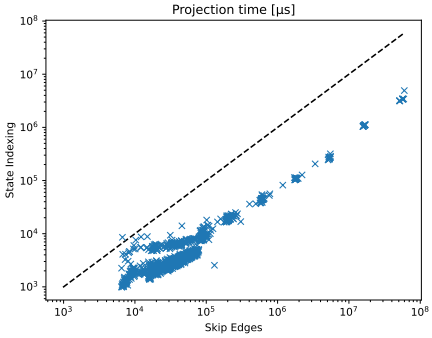


Fig. 10: State Indexing speedup compared to Mona projection time. The vertical axis represents the number of occurrences (logarithmic scale), while the horizontal axis indicates the speedup (positive) or slowdown (negative).

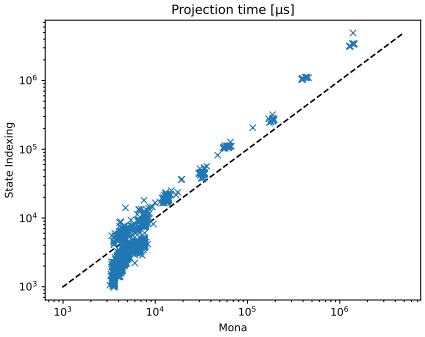
Projection of the first variable

The projection of the first variable (the variable with the lowest id) was included in the experiments because projecting such a variable creates an automaton with the highest level of nondeterminism. And followed by the determinization, it can effectively substitute experiments focused on determinization.

Unsurprisingly, the State Indexing reimplementation is about ten times faster than Skip Edges. However, it should be noted that Mona and State Indexing performed almost identically. This similarity might be due to a less optimized projection algorithm in the Mona tool for the first variable or a potentially suboptimal determinization algorithm.



Comparing State Indexing and Mona: logarithmic time performance [μs].



Comparing State Indexing and Skip Edges: logarithmic time performance [μs].

Fig. 11: Comparison of projection and determinization time for the tool Mona, reimplementation using Skip Edges, and reimplementation using State Indexing.

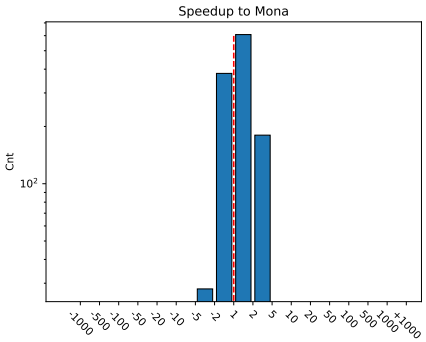


Fig. 12: State Indexing speedup compared to Mona projection time. The vertical axis represents the number of occurrences (logarithmic scale), while the horizontal axis indicates the speedup (positive) or slowdown (negative).

5.2 WS1S formulae

This subsection presents the results of experiments performed on a set of parametric families of WS1S formulae. The first benchmark (`horn-in`) consists of formulae used to evaluate the Toss tool, as presented in [19]. The second set of families (`horn-leq`) comes from the work of D'Antoni et al. [20], artificially enhanced by added quantifier alternations. The last set of benchmarks (`horn-trans`, `set-closed`, and `set-singletons`) is from the dWiNA tool [21].

Benchmark	SE–Mona [ms]	SI–Mona [ms]	Mona [ms]
`horn-in`	∞(7)	∞(6)	∞(**16**)
`horn-leq`	209	**101**	∞(18)
`horn-leq` (+3)	198	**88**	∞(18)
`horn-leq` (+4)	193	**88**	∞(18)
`horn-trans`	∞(15)	∞(**20**)	∞(16)
`set-closed`	∞(**5**)	∞(4)	∞(**5**)
`set-singletons`	∞(4)	∞(4)	∞(**5**)

Table 1: Results for parametrized benchmarks are presented up to $k = 20$. SE–Mona represents a Mona reimplementation using Skip Edges, while SI–Mona represents the reimplementation utilizing State Indexing. $\infty(n)$ indicates that the cumulative time for the computation of formulas $\leq n$ exceeded the 10-second limit.

Experiments focused on the cumulative time of computing parametrized formulae from the easiest to the most difficult. Once the cumulative time exceeded the 10-second limit during the computation of the n-th formula, the tool was stopped with the result $\infty(n)$. Otherwise, the result was the value of the cumulative time.

Surprisingly, Mona completely failed on `horn-leq` benchmarks. Mona consumed more than 10 seconds, while both of its reimplementations, which use DFAs instead of BDDs, completed calculations in 200 milliseconds.

On the other hand, Mona outperformed both reimplementations on `horn-in` benchmarks. This could be caused by using Brzozowski minimization algorithm, as on these benchmarks, minimization takes up to 98% of the computation time.

As expected, the State Indexing reimplementation was faster than Skip Edges and Mona on most benchmarks. However, the speedup was not as significant as one might expect based on the performance of reimplemented automata operations. This behavior is caused by the high number of minimizations performed, which utilizes inefficient Brzozowski algorithm.

6 Conclusion

This paper introduces a new reimplementation of the Mona tool for decision procedures in WS1S. The primary motivation was to introduce a pure automata-based approach instead of the combination of BDDs and automata. The reimplementation is based on the Master's thesis of Bc. Pavel Bednář, where skip edges were used to mimic the behavior of the BDDs employed by Mona. Our reimplementation utilizes state indexing with the Mata automata library to simulate indexes of inner nodes in BDDs.

Using the pure automata-based approach, the state indexing method resulted in a tenfold faster computation of determinization, intersection, and projection. On benchmark formulae designed to stress test Mona, both reimplementations performed better, with only one exception that can be attributed to using an inefficient Brzozowski minimization algorithm. Our reimplementation outperformed Bednář's approach on most benchmarks, achieving a speedup ranging from two to ten.

The work can be further improved by implementing a better simulation-based minimization, utilizing abstraction, or by performing operations on nondeterministic finite automata instead of deterministic ones.

References

[1] Vojnar Tomáš: Lecture notes in Static Analysis and Verification: SAT and SMT Solving. BUT - Faculty of Information Technology (2023). https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-10.pdf

[2] Klarlund, N.: A theory of restrictions for logics and automata. In: Computer Aided Verification, CAV '99. LNCS, vol. 1633

[3] Büchi, J.R.: Weak second-order arithmetic and finite automata. Mathematical Logic Quarterly **6**(1-6), 66–92 (1960) https://doi.org/10.1002/malq.19600060105

[4] Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. PLDI '01, pp. 221–231. Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/378795.378851

[5] Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. SIGPLAN Not. **46**(1), 611–622 (2011) https://doi.org/10.1145/1925844.1926455

[6] Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Trans. Softw. Eng. Methodol. **22**(4) (2013) https://doi.org/10.1145/2522920.2522926

[7] Baukus, K., Bensalem, S., Lakhnech, Y., Stahl, K., Equation, V.: Abstracting ws1s systems to verify parameterized networks. (2001). https://doi.org/10.1007/3-540-46419-0_14

[8] Klarlund, N., Nielsen, M., Sunesen, K.: A case study in verification based on trace abstractions. In: Broy, M., Merz, S., Spies, K. (eds.) Formal Systems Specification, pp. 341–373. Springer, Berlin, Heidelberg (1996)

[9] Sandholm, A., Schwartzbach, M.I.: Distributed safety controllers for web services. In: Astesiano, E. (ed.) Fundamental Approaches to Software Engineering, pp. 270–284. Springer, Berlin, Heidelberg (1998)

[10] Basin, D., Klarlund, N.: Automata based symbolic reasoning in hardware verification. Form. Methods Syst. Des. **13**(3), 255–288 (1998) https://doi.org/10.1023/A:1008644009416

[11] Pavel, B.: Rozhodování ws1s pomocí symbolických automatů [online]. Diplomová práce, Vysoké učení technické v Brně (2023 [cit. 2024-01-20]). https://theses.cz/id/kq7t5j/

[12] Vojnar Tomáš: Lecture notes in Static Analysis and Verification: Binary Decision Diagrams. BUT - Faculty of Information Technology (2023). https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-07.pdf

[13] Bollig, B., Wegener, I.: Improving the variable ordering of obdds is np-complete. IEEE Transactions on Computers **45**(9), 993–1002 (1996) https://doi.org/10.1109/12.537122

[14] Rice, M., Kulhari, S.: A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction, Technical Report. http://www.cs.ucr.edu/~skulhari/StaticHeuristics.pdf (2008)

[15] Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, Aarhus University, (2001). BRICS, Department of Computer Science, Aarhus University. Notes Series NS-01-1. Available from http://www.brics.dk/mona/. Revision of BRICS NS-98-3

[16] Bryant: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**(8), 677–691 (1986) https://doi.org/10.1109/TC.1986.1676819

[17] Béal, M.-P., Crochemore, M.: Minimizing incomplete automata. In: Finite-State Methods and Natural Language Processing (FSMNLP'08). Joint Research Centre, pp. 9–16, United States (2008). https://hal.science/hal-00620274

[18] Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. (1962). https://api.semanticscholar.org/CorpusID:118363215

[19] Ganzow, T., Kaiser, Ł.: New algorithm for weak monadic second-order logic on inductive structures. In: Dawar, A., Veith, H. (eds.) Computer Science Logic, pp. 366–380. Springer, Berlin, Heidelberg (2010)

[20] D'Antoni, L., Veanes, M.: Minimization of symbolic automata. SIGPLAN Not. **49**(1), 541–553 (2014) https://doi.org/10.1145/2578855.2535849

[21] Fiedor, T., Holík, L., Lengál, O., Vojnar, T.: Nested antichains for ws1s. Acta Informatica **56**(3), 205–228 (2019) https://doi.org/10.1007/s00236-018-0331-z