Monadic Second-Order Logic on Finite Sequences

Loris D'Antoni
University of Wisconsin-Madison
loris@cs.wisc.edu

Margus Veanes
Microsoft Research
margus@microsoft.com

Abstract

We extend the weak monadic second-order logic of one successor for finite strings (M2L-STR) to symbolic alphabets by allowing character predicates to range over decidable quantifier free theories instead of finite alphabets. We call this logic, which is able to describe sequences over complex and potentially infinite domains, symbolic M2L-STR (S-M2L-STR). We present a decision procedure for S-M2L-STR based on a reduction to symbolic finite automata, a decidable extension of finite automata that allows transitions to carry predicates and can therefore model complex alphabets. The reduction constructs a symbolic automaton over an alphabet consisting of pairs of symbols where the first element of each pair is a symbol in the original formula's alphabet, while the second element is a bit-vector. To handle this modified alphabet we show that the Cartesian product of two decidable Boolean algebras, e.g., the product of formula's algebra and bit-vector's algebra, also forms a decidable Boolean algebra. To make the decision procedure practical, we propose two efficient representations of the Cartesian product of two Boolean algebras, one based on algebraic decision diagrams and one on a variant of Shannon expansions. Finally, we implement our decision procedure and evaluate it on more than 10,000 formulas. Despite the generality, our implementation has comparable performance with the state-of-the-art M2L-STR solvers.

Categories and Subject Descriptors F.2.2 [*Theory of Computation*]: Automata over infinite objects, Regular languages

Keywords Symbolic automata, SWS1S, MSO logic

1. Introduction

Logics for describing strings and sequences are ubiquitous and appear in applications such as program verification, string processing, program monitoring, and personalized education [4, 7, 27, 36, 40]. These logics are typically equipped with operators that can describe the order between events appearing in a given sequence and operators for describing the kind of events that can appear. Notable examples are linear temporal logic (LTL) [18] and the weak monadic second-order logic of one successor (WS1S) [9]. For example, an LTL formula can specify that, in a given string, the symbol a should always be followed by a symbol b. WS1S is more expressive than LTL and one can write a WS1S formula specifying that every b is preceded by an even number of as. The success of these logics is largely due to their good properties and the decidability of checking satisfiability. In this paper, we present S-M2L-STR as a decidable extension of M2L-STR (WS1S for *finite* strings, c.f. [29, 30]) for describing *finite* sequences over arbitrary domains.

S-M2L-STR: a logic for sequences over arbitrary domains. Although used in many practical contexts, logics like M2L-STR and LTL can only describe sequences over finite and typically small domains. For example, neither LTL nor M2L-STR provide an elegant and natural way to describe the set of all sequences of integers such that *every odd number is eventually followed by a number greater than 4*. In this paper we present symbolic M2L-STR (S-M2L-STR), an extension of M2L-STR that can naturally describe this property while retaining decidable satisfiability checking. S-M2L-STR formulas are parametric in an underlying alphabet theory (e.g., linear integer arithmetic), which operates over a potentially infinite domain (e.g., integers). To retain decidability, the underlying theory is required to form a decidable Boolean algebra, i.e., it is decidable to check whether a predicate is satisfiable and the set of predicates is closed under Boolean operations.

The following S-M2L-STR formula, ψ, captures the property we informally described: $\forall x.[\varphi_{odd}](x) \rightarrow \exists y.x < y \wedge [\varphi_{>4}](y)$. Here φ_{odd} and $\varphi_{>4}$ are unary integer linear arithmetic predicates, and ψ describes all sequences of integers such that if a position x contains an odd element ($[\varphi_{odd}](x)$), then there exists a position y appearing after x ($x < y$) that contains an element greater than 4 ($[\varphi_{>4}](y)$). The variables x and y represent positions in the sequence.

From S-M2L-STR to M2L-STR. M2L-STR can only describe sequences over finite domains because it only supports (encoding of) unary predicates of the form $[a](x)$ where a is a symbol from a finite alphabet. Despite this limitation, there is a way to convert every S-M2L-STR formula into an equi-satisfiable M2L-STR one. Although the predicates appearing in a given S-M2L-STR formula φ operate over an infinite domain, the set of maximal satisfiable Boolean combinations $M(\varphi)$—also called minterms—of such predicates induces a finite set of equivalence classes.[1] For example, the set of equivalence classes of the S-M2L-STR formula ψ is:

$$M(\psi)=\{\varphi_{odd}\wedge\varphi_{>4}, \neg\varphi_{odd}\wedge\varphi_{>4}, \varphi_{odd}\wedge\neg\varphi_{>4}, \neg\varphi_{odd}\wedge\neg\varphi_{>4}\}.$$

Intuitively, using only these predicates there is no way to, e.g., distinguish the number 1 from the number 3, i.e., given any sequence l, if one replaces any element 1 in l with the element 3, the new sequence l' is a model of ψ if and only if l is a model of ψ. Using this argument, every S-M2L-STR formula ψ can be compiled into an equi-satisfiable M2L-STR formula over the alphabet $M(\psi)$. Unfortunately, computing the set of formulas $M(\varphi)$ is an expensive procedure and requires numerous satisfiability checks over potentially large predicates. Moreover, since there can be exponentially many minterms, such a reduction may result in an alphabet whose size is exponential in the size of the S-M2L-STR formula.

[1] This property was already observed in the context of symbolic finite automata [16].

POPL '17, January 18 - 20, 2017, Paris, France

ACM 978-1-4503-4660-3/17/01…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/3009837.3009844

***A symbolic decision procedure for* S-M2L-STR.** We propose a decision procedure for S-M2L-STR that avoids the reduction to M2L-STR. Our decision procedure extends the following result: given a M2L-STR formula φ, one can construct a finite automaton that accepts the same set of sequences accepted by φ [9]. In this reduction, if the M2L-STR formula φ has k free variables and operates over a finite alphabet Σ, the resulting automaton operates over the alphabet $\Sigma \times \{0,1\}^k$, where k bits are used to represent variable values—i.e., what positions in the string are attached to what variables. We show that, given an S-M2L-STR formula φ, one can construct a *symbolic finite automaton* (s-FA) [16] that accepts the same set of sequences accepted by φ. Similarly to S-M2L-STR, s-FAs extend finite automata to sequences over symbolic alphabets by allowing transitions to carry predicates from an underlying alphabet theory. This theory is required to form a decidable Boolean algebra—i.e., the theory is closed under Boolean operations and, given a predicate in the theory, it is decidable to check whether the predicate is satisfiable. Given a S-M2L-STR formula φ over an (infinite) alphabet σ and with k free variables, we construct a symbolic automaton A_φ over the alphabet $\sigma \times \{0,1\}^k$. We do so by showing that if the alphabet theory used by φ forms a decidable Boolean algebra, the theory resulting from building the product alphabet $\sigma \times \{0,1\}^k$ still forms a decidable Boolean algebra. Since it is decidable to check whether an s-FA accepts the empty language, we can then test whether the formula φ is satisfiable by checking whether the corresponding s-FA accepts some sequence. If the formula is satisfiable, we can use the s-FA to produce a model for it.

***Implementing the algebra* $\sigma \times \{0,1\}^k$** To implement our S-M2L-STR decision procedure we propose two efficient representations of the product Boolean algebra over the alphabet $\sigma \times \{0,1\}^k$. The first representation is based on algebraic decision diagrams or ADDs [6],[2] by allowing the leaves to be predicates themselves, from another effective Boolean algebra. The second representation is based on a variant of Shannon expansions, that utilizes If-Then-Else or ITE expressions whose nonterminals are predicates of one algebra and whose terminals are predicates of another algebra.

We implement our decision procedure using both of these representations and evaluate on more than 10,000 existing M2L-STR formulas, and on new benchmark S-M2L-STR formulas over the theories of bit-vector arithmetic and linear integer arithmetic. Our experiments show the following. (1) Our solver has comparable performance to existing M2L-STR solvers in the case of M2L-STR formulas. (2) Our solver has better performance than the reduction from S-M2L-STR to M2L-STR in the case of formulas over complex alphabet theories. (3) The representation based on ADDs is more efficient than the representation based on ITEs.

Contributions. In summary our contributions are the following.

(1)The logic S-M2L-STR for describing sets of finite sequences over infinite alphabets (Section 5).

(2) A symbolic decision procedure for S-M2L-STR based on a reduction to symbolic finite automata over a product Boolean algebra, with two efficient representations of this algebra (Section 6).

(3) A *predicate trie* algorithm for efficiently detecting equivalence of predicates. (Section 4)

(4) An efficient implementation of S-M2L-STR and its decision procedure together with an integration with external SMT solvers to support complex alphabet theories (Section 7).

(5) A comprehensive evaluation on more than 10,000 existing M2L-STR benchmarks and on new S-M2L-STR formulas over the theories of bit-vectors and linear integer arithmetic (Section 8).

[2] ADDs are an extension of BDDs whose leaves can be elements of another finite domain, not just 0 and 1.

```
(* Caesar cipher over a list *)
let rec map_caesar (x: int list) : int list =
  match x with
    [] -> []
  | h::t -> ((h + 5) mod 26) :: map_caesar t
(* Remove odd numbers *)
let rec filter_ev (x: int list) : int list =
  match x with
    [] -> []
  | h::t when (h mod 2=0) -> h :: filter_ev t
  | h::t when (h mod 2=1) -> filter_ev t
(* Composition of the two functions *)
let map_filt (x: int list) : int list =
  filter_ev (map_caesar x)
(* Composition of the functions *)
let filt_map_filt (x: int list) : int list =
  map_filt (filter_ev x)
(* Contracts *)
{list_of_even x} y := map_caesar x {list_of_odd y}
{list_of_even x} y := map_filt x {empty_list y}
{any_list x} y := filt_map_filt x {empty_list y}
```

Figure 1. Example of contracts for list manipulating programs.

2. Motivating Example

We use a simple example to illustrate the need for a decidable logic for describing properties involving lists over arbitrary domains.

Figure 1 contains simple ML programs operating over lists of integers: map_caesar replaces each value x of an integer list with $(x + 5)\ mod\ 26$, filter_ev removes all the odd elements from a list, while map_filt and and filt_map_filt are different functional compositions of map_caesar and filter_ev.

A programmer might want to verify that such functions adhere to the contracts given at the end of the figure. For example, the first contract specifies that the function map_caeser, when given as input a list of even numbers always produces a list of odd numbers. Similarly, the last contract specifies that the function filt_map_filt always outputs the empty list regardless of its input. Although arbitrary contracts over arbitrary programs are hard to verify, contracts like the ones presented in Figure 1 have been successfully verified using tools such as Fast [17], an automaton-based language for verifying contracts in programs that operate over lists and trees.

The reader at this point might wonder how constructs such as list_of_even and any_list can be programmed by the user. One option is to define them using programs of type (int list $\rightarrow$ bool). While this option provides generality, it has two main drawbacks:

- The user can write arbitrary programs that are therefore hard to reason about.

- Writing contracts using programs might be challenging, error-prone, and lengthy.

Another option, is to restrict the type of predicates appearing in the contract to restricted classes of programs that have decidable properties. The language Fast [17] adopts this option and only allows contracts to be specified using symbolic finite automata. While this option addresses the first problem, it still poses a burden on the programmer who has to think in terms of automata rather than declaratively.

We argue that a good option that fits both our requirements is to write such contracts using a decidable logic and we propose S-M2L-STR as a possible choice for such a logic. For example, the predicate list_of_even can be expressed using S-M2L-STR as

$$\text{list\_of\_even } l \stackrel{\text{def}}{=} l \in \mathscr{L}(\forall p.[\lambda x.x\ mod\ 2 = 0](p))$$

where $\mathscr{L}(\varphi)$ is the set of lists that are models of the S-M2L-STR formula φ. Informally, the S-M2L-STR formula states that all the positions p in the list l contain a number satisfying the predicate $\lambda x.x \ mod \ 2 = 0$. The other predicates can be defined smilarly.

$$\begin{array}{rcl} \text{list\_of\_odd}\ l & \stackrel{\text{def}}{=} & l \in \mathscr{L}(\forall p.[\lambda x.x \ mod \ 2 = 1](p)) \\ \text{any\_list}\ l & \stackrel{\text{def}}{=} & l \in \mathscr{L}(true) \\ \text{empty\_list}\ l & \stackrel{\text{def}}{=} & l \in \mathscr{L}(\neg \exists p.true) \end{array}$$

The predicate empty_list simply states that the list l contains no positions—i.e., the list is empty. Notice that, while the core logic of S-M2L-STR is quite wordy, one can easily imagine a language in which macros are used to define commonly occurring predicates. In this paper, we formalize the logic S-M2L-STR and provide a decision procedure for it. As the decision procedure of S-M2L-STR produces symbolic automata, S-M2L-STR can be directly used as the specification logic for languages like Fast [17] and Bek [26], in which properties have to be expressed as symbolic automata.

3. Effective Boolean Algebras and Generic BDDs

We recall the notion of an *effective Boolean algebra* that is used in place of a *concrete* alphabet and introduce a particular algebra based on BDDs that is used in our main algorithm.

3.1 Effective Boolean Algebras

We use effective Boolean algebras in place of concrete alphabets. An *effective Boolean algebra* $\mathcal{A}$ is a tuple $(U, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where U is a non-empty recursively enumerable set called the *universe* of $\mathcal{A}$. Ψ is a recursively enumerable set of *predicates* closed under the Boolean connectives, $\vee, \wedge : \Psi \times \Psi \rightarrow \Psi$, $\neg : \Psi \rightarrow \Psi$, and $\bot, \top \in \Psi$. The *denotation function* $[\![\_]\!] : \Psi \rightarrow 2^U$ is r.e. and is such that, $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = U$, for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = U \setminus [\![\varphi]\!]$.[3] For $\varphi \in \Psi$, we write $\mathbf{Sat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$ and say that φ is *satisfiable*. The algebra $\mathcal{A}$ is *decidable* if $\mathbf{Sat}$ is decidable.

In practice, an (effective) Boolean algebra is implemented as an API with corresponding methods implementing the operations. We use the following Boolean algebras. $\mathbf{B1} \stackrel{\text{def}}{=} (\{\emptyset\}, \{\mathbf{0}, \mathbf{1}\}, \{\mathbf{0} \mapsto \emptyset, \mathbf{1} \mapsto \{\emptyset\}\}, \mathbf{0}, \mathbf{1}, \vee, \wedge, \neg)$ is the simplest possible effective Boolean algebra. The connectives implement the standard truth tables. SMT_τ is a Boolean algebra representing a restricted use of an SMT solver such as Z3 [19, 20] on predicates over elements of type τ. Formally, $SMT_\tau = (U, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$, where U is the set of all elements of type τ, Ψ is the set of all quantifier free formulas containing a single uninterpreted constant $x : \tau$, $\top$ is $x = x$, $\bot$ is $x \neq x$, and the Boolean operations are the corresponding connectives in SMT formulas. The interpretation function $[\![\varphi]\!]$ is defined using the operations of satisfiability checking and model generation provided by an SMT solver.

General notations. For a sequence $s = (e_1, \ldots, e_n)$ and for $i \in [1; n]$ we let $s(i) \stackrel{\text{def}}{=} e_i$ and $s \cdot e \stackrel{\text{def}}{=} (e_1, \ldots, e_n, e)$. The empty sequence is $()$. For a non-empty set S of positive integers we define $\varepsilon(S)$ to be any element of S and $\varepsilon(\emptyset) \stackrel{\text{def}}{=} 0$. In practice, we only use $\varepsilon(S)$ when $|S| = 1$. For a term or formula φ we let $FV(\varphi)$ denote the set of all free variables in φ.

3.2 Generic BDDs

In our algorithm we use a variant of *Algebraic Decision Diagrams* [6] or *ADDs*, also known as *Multi-Terminal BDDs* [11]. ADDs are binary decision diagrams in which the terminals are elements of an algebra $\mathcal{A}$. We call our variant *generic BDDs* and we let the leaves belong to $\Psi_\mathcal{A}$ where $\mathcal{A}$ is a *Boolean* algebra instead of an arbitrary algebra. Generic BDDs differ from BDDs where leaves can only belong to $\Psi_{\mathbf{B1}}$. The terminals are elements that denote subsets of $U_\mathcal{A}$. Since the number of bits in the domain of generic BDDs is unrestricted[4] they denote sets of functions from $\mathbb{N}$ to $\{[\![\alpha]\!] \mid \alpha \in \Psi_\mathcal{A}\}$ that we view as relations over $U_\mathcal{A} \times \mathbb{N}$.

Formally, a generic BDD algebra over a given Boolean leaf algebra $\mathcal{A}$ is a tuple $\mathbf{BDD}\langle\mathcal{A}\rangle = (U_\mathcal{A} \times \mathbb{N}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathbb{N}$ is the set of natural numbers. The set of predicates Ψ is defined as the set $\bigcup_{k \geq 0} \Psi^{(k)}$, where $\Psi^{(k)}$ for $k \geq 0$, is the least set that satisfies the following conditions.

- For all $\psi \in \Psi_\mathcal{A}$, map ψ to a unique element *leaf*(ψ) such that, for every $\psi \in \Psi_\mathcal{A}$, if $[\![\psi]\!]_\mathcal{A} = [\![\phi]\!]_\mathcal{A}$ then *leaf*$(\psi) =$ *leaf*(ϕ).
- If $\psi \in \Psi_\mathcal{A}$, then *leaf*$(\psi) \in \Psi^{(0)}$.
- If $k \geq 1$, $0 \leq i, j < k$, $\psi \in \Psi^{(i)}$, $\varphi \in \Psi^{(j)}$, and $\psi \neq \varphi$, then *node*$(k, \psi, \varphi) \in \Psi^{(k)}$.

A predicate *leaf*(ψ) is called a *leaf* or a *terminal*. We discuss in Section 4 how *leaf*(ψ) can be implemented using a *predicate trie*. Accordingly, a predicate *node*(k, ψ, φ) is a *nonleaf* or a *nonterminal*, and k is its *position*. The position of a leaf is 0.

We now define the denotation of a predicate in Ψ. For $n \geq 0$, $k \geq 1$, we define $Bit(k, n) \Leftrightarrow (n \div 2^{k-1}) \bmod 2 = 1$, where $\div$ denotes integer division. In other words, $Bit(k, n)$ holds iff the k'th bit of the binary representation of n is 1, e.g., $Bit(2, 6)$ and $Bit(3, 6)$ are true (as a convention we start counting from 1). The denotation of a terminal *leaf*(ψ) is $[\![leaf(\psi)]\!] \stackrel{\text{def}}{=} [\![\psi]\!]_\mathcal{A} \times \mathbb{N}$. The denotation of a non-terminal *node*(k, ψ_1, ψ_0) is defined as

$$\begin{array}{rcl} [\![node(k, \psi_1, \psi_0)]\!] & \stackrel{\text{def}}{=} & \{(a, n) \in [\![\psi_1]\!] \mid Bit(k, n)\} \\ & \cup & \{(a, n) \in [\![\psi_0]\!] \mid \neg Bit(k, n)\}. \end{array}$$

Observe that the positions of ψ_1 and ψ_0 are strictly smaller than k. Let $\bot \stackrel{\text{def}}{=} leaf(\bot_\mathcal{A})$ and $\top \stackrel{\text{def}}{=} leaf(\top_\mathcal{A})$. Clearly, $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = U_{\mathbf{BDD}\langle\mathcal{A}\rangle}$. The Boolean operations of $\mathbf{BDD}\langle\mathcal{A}\rangle$ can be implemented as shown in [11, Section 4.3] as an extension of [8, Section 4.3] and the binary operators satisfy [11, Theorem 4.1]. Let & be $\wedge_\mathcal{A}$ and let $-$ be $\neg_\mathcal{A}$. For leaves, complement and conjunction are defined as follows:

$$\neg leaf(\psi) \stackrel{\text{def}}{=} leaf(-\psi), \quad leaf(\psi) \wedge leaf(\varphi) \stackrel{\text{def}}{=} leaf(\psi \ \& \ \varphi)$$

EXAMPLE 1. [Classic BDDs] $\mathbf{BDD}\langle\mathbf{B1}\rangle$ corresponds to the classic case of BDDs [8], i.e., ADDs with terminals $\mathbf{1}$ or $\mathbf{0}$ but with an unbounded (open-ended) number of Boolean variables. So each $\psi \in \Psi_{\mathbf{BDD}\langle\mathbf{B1}\rangle}$ represents a subset of $\mathbb{N}$ rather than a subset of $\{0, 1\}^k$ for some finite k.[5] We assume, without loss of generality, that for any set S, $S \times \{()\} = \{()\} \times S = S$. So $U_{\mathbf{BDD}\langle\mathbf{B1}\rangle} = \mathbb{N}$. ⊠

Since we are dealing with *predicates* in $\Psi_\mathcal{A}$ rather than concrete sets, a new terminal *leaf*(ϕ) (where ϕ is $\psi \ \& \ \varphi$ or $-\psi$ above) should be created only if, so far, there has been no terminal *leaf*(ϕ') such that $[\![\phi]\!]_\mathcal{A} = [\![\phi']\!]_\mathcal{A}$. Otherwise the already existing equivalent terminal should be used to keep the constructed ADD canonical. In general, the number of possible distinct terminals is unbounded and terminals themselves may denote infinite sets. To this end we introduce the new notion of *predicate tries* in Section 4.

[3] The underlying Boolean algebra of $\mathcal{A}$ corresponds to the *field of sets* $(U_\mathcal{A}, \{[\![\psi]\!] \mid \psi \in \Psi_\mathcal{A}\})$, where elements of $U_\mathcal{A}$ are sometimes called *points*, using the *Representation Theorem of Boolean Algebras* (cf. [10]).

[4] Typically, in BDDs, as well as ADDs, the number of bits is restricted by a fixed bound.

[5] Semantically, $\mathbf{BDD}\langle\mathbf{B1}\rangle$ is isomorphic to the *countable atomless Boolean algebra* also called the *countable Cantor algebra* (cf. [10]). In this case for example $\neg leaf(\mathbf{1}) = leaf(\mathbf{0})$ and $\neg leaf(\mathbf{0}) = leaf(\mathbf{1})$.

```
class Tree
    k : int    // depth of the node (distance from the root)
    ψ : Ψ      // representative predicate
    t0 : Tree = null
    t1 : Tree = null
    IsLeaf : bool
        return t0==null and t1==null

class PredicateTrie
    tree = new Tree(0, ⊥, new Tree(1, ⊥), new Tree(1, ⊤))
    v = (Atom(⊤))

    SEARCH(φ : Ψ) : Ψ
        return SEARCH(φ, tree)

    SEARCH(φ : Ψ, t : Tree) : Ψ
        let k = t.k
        let ψ = t.ψ
        if (t.IsLeaf)
            if (k ≤ |v|)  // use v[k] to distinguish ψ from φ
                if v[k] ∈ ψ
                    t.t1 = new Tree(k + 1, ψ)
                    if v[k] ∈ φ
                        return SEARCH(φ, t.t1)
                    else
                        t.t0 = new Tree(k + 1, φ)
                        return φ
                else
                    t.t0 = new Tree(k + 1, ψ)
                    if v[k] ∈ φ
                        t.t1 = new Tree(k + 1, φ)
                        return φ
                    else
                        return SEARCH(φ, t.t0)
            else // compute symmetric difference
                let Δ = φ ⇎ ψ
                if not Sat(Δ)
                    return ψ
                else // extend v to distinguish ψ from φ
                    let α = Atom(Δ)
                    v = v · α
                    if α ∈ φ
                        t.t0 = new Tree(k + 1, ψ)
                        t.t1 = new Tree(k + 1, φ)
                    else
                        t.t0 = new Tree(k + 1, φ)
                        t.t1 = new Tree(k + 1, ψ)
                    return φ
        else // proceed recursively on internal nodes
            if v[k] ∈ φ
                if t.t1==null
                    t.t1 = new Tree(k + 1, φ)
                    return φ
                else
                    return SEARCH(φ, t.t1)
            else
                if t.t0==null
                    t.t0 = new Tree(k + 1, φ)
                    return φ
                else
                    return SEARCH(φ, t.t0)
```

Figure 2. Predicate trie algorithm.

4. Predicate Trie

The goal of a predicate trie is to maintain a set S of pairwise inequivalent predicates from $\Psi_{\mathcal{A}}$ so that new terminals of predicates in $\Psi_{\mathbf{BDD}\langle\mathcal{A}\rangle}$ are created only for predicates that are inequivalent to all the predicates seen so far. In other words, all the elements of S are representatives of distinct equivalence classes of formulas. The naive algorithm for accomplishing this task, when receiving a new predicate φ, checks the equivalence of φ against all the predicates already in S and only adds φ to S if no predicate in S is equivalent to φ. For this algorithm, adding a new predicate has *linear* complexity in the size of S. We describe a different algorithm for which, under some assumptions, adding a new predicate to S has logarithmic complexity in the size of S.

A sufficient condition for the following algorithm to work is that the algebra $\mathcal{A}$ is *atomic*. $\mathcal{A}$ is *atomic* if it is possible to create predicates in $\Psi_{\mathcal{A}}$ that denote *singleton* sets. An atomic effective Boolean algebra $\mathcal{A}$ has the additional component

$$\mathit{Atom} : \Psi_{\mathcal{A}} \to \Psi_{A}$$

such that $[\![\mathit{Atom}(\varphi)]\!] \subseteq [\![\varphi]\!]$, and if $\mathbf{Sat}(\varphi)$ then $|[\![\mathit{Atom}(\varphi)]\!]| = 1$. For example, SMT_τ is atomic but $\mathbf{BDD}\langle\mathbf{B1}\rangle$ is not atomic.

Atoms allow us to efficiently check satisfiability of formulas of the form $\alpha \wedge \psi$ by treating atoms α as concrete values. For example, given a predicate $\lambda x.\psi(x)$ and an atom α with denotation $\{a\}$, we can simply evaluate the formula $\psi(a)$ instead of performing a general satisfiability check. In the following, we write $\alpha \in \psi$ for $a \in [\![\psi]\!]$ to emphasize the special case of α being an atom.

The main idea behind our algorithm is to maintain the following invariant. Given a *fixed* sequence $\boldsymbol{v} = (v_i)_{i=1}^n$ where each element v_i is an atom and a finite set $S \subseteq \Psi_{\mathcal{A}}$ of pairwise inequivalent predicates, for every two predicates $\psi, \varphi \in S$ there is some v_i such that $v_i \in \psi \Leftrightarrow v_i \notin \varphi$. To efficiently maintain this invariant we order the predicates in S with respect to the following order.[6]

$$\psi \prec_{\boldsymbol{v}} \varphi \stackrel{\text{def}}{=} \exists i \in [1; |\boldsymbol{v}|](v_i \in \varphi \backslash \psi \wedge \forall j{<}i(v_j \in \varphi \Leftrightarrow v_j \in \psi))$$

Intuitively, $\psi \prec_{\boldsymbol{v}} \varphi$, or $\psi \prec \varphi$ when $\boldsymbol{v}$ is clear, means that if v_i is the first element of the sequence $\boldsymbol{v}$ that distinguishes ψ from φ then v_i occurs in φ. It follows in particular that $\bot \prec \varphi$ and $\varphi \prec \top$ for all φ other than $\bot$ and $\top$. Using the order $\prec$, we can perform a binary search for efficiently inserting new predicates into S, and if need be, extend $\boldsymbol{v}$ in the process.

A key property of our algorithm is that, when the sequence $\boldsymbol{v}$ does not suffice to distinguish a new predicate from the ones already in S, appending a new atom at the end of $\boldsymbol{v}$ does not affect the relative ordering among the predicates already in S.

PROPOSITION 1. *If $\psi \prec_{\boldsymbol{v}} \varphi$ and $\alpha \notin \boldsymbol{v}$ then $\psi \prec_{\boldsymbol{v}\cdot\alpha} \varphi$*

We design a *predicate trie* data structure for representing S in $\prec$-order and illustrate its operations in Figure 2. A concrete predicate trie PredicateTrie$(t, \boldsymbol{v})$ contains two attributes: the sequence $\boldsymbol{v}$ of atoms, and a binary tree t of depth $|\boldsymbol{v}|$. The leaves of t are the elements of S and the internal nodes of t have two subtrees, called the $\mathbf{1}$-subtree and the $\mathbf{0}$-subtree, respectively The $\mathbf{1}$-subtree t_1 and the $\mathbf{0}$-subtree t_0 are such that, for all leaves ψ_0 in t_0 and ψ_1 in t_1, we have that $\psi_0 \prec \psi_1$. The linear order $\prec$ determined by $\boldsymbol{v}$ is just lexicographic order on branches to leaves ψ in t as represented by binary valuation sequences $(b_1, b_2, \ldots, b_m)$ where $m \leq |\boldsymbol{v}|$ and $b_i = \mathbf{1}$ if $\boldsymbol{v}(i) \in \psi$, and $b_i = \mathbf{0}$ if $\boldsymbol{v}(i) \notin \psi$. For example, a branch $(\mathbf{1}, \mathbf{0}, \mathbf{0})$ to ψ represents the valuation $v_1 \in \psi, v_2 \notin \psi$, and $v_3 \notin \psi$. The algorithm for searching for a predicate φ in a trie is given in Figure 2 as the method SEARCH.

The trie is initialized as follows. The initial sequence $\boldsymbol{v}$ has length one and contains an atom (α). The binary tree has two

[6] Formula $\varphi\backslash\psi$ stands for $\varphi \wedge \neg\psi$.

leaves that are $\bot$ and $\top$, respectively. Trivially $\alpha \notin \bot$ and $\alpha \in \top$. This ensures that valid predicates will always be mapped to $\top$ and unsatisfiable predicates to $\bot$.

When a predicate φ is searched in the trie, φ is searched in the tree of the trie. In the case of a leaf (lines 20–48), there are two possibilities depending on whether the depth k of the leaf is within the limits of the sequence $\boldsymbol{v}$ ($k \leq |\boldsymbol{v}|$) or not. In the former case (lines 21–34) the predicate ψ of the leaf and the predicate φ are compared by checking whether the symbol in the atom $\boldsymbol{v}(k)$ distinguishes them and the leaf is expanded to a subtree incorporating both ψ and φ. In the case $k > |\boldsymbol{v}|$ (lines 36–48), the sequence $\boldsymbol{v}$ was not enough to distinguish ψ from φ.

The algorithm then proceeds to evaluate the symmetric difference Δ of ψ and φ, which is the most costly operation of the algorithm. The two possible outcomes are that either, ψ and φ are equivalent, in which case ψ is returned (line 38) as the chosen representative, or they are not equivalent, in which case a representative α is chosen from Δ to distinguish the predicates, the leaf is expanded to a subtree, the sequence $\boldsymbol{v}$ is extended accordingly with α, and φ is returned.

In the case of a nonleaf (lines 50–61) the predicate is searched recursively in the subtrees depending on whether $\boldsymbol{v}(k) \in \varphi$ or not. If the corresponding subtree is **null** (lines 52 and 58), it means that there is yet no predicate with the corresponding valuation sequence, the tree is updated to have a new leaf whose predicate is φ, and φ is returned.

Using values (or atoms) in $\boldsymbol{v}$ to distinguish predicates exploits the assumption that evaluating a predicate with respect to a concrete value (or atom) is cheaper than a general satisfiability check. The algorithm works obviously also for effective Boolean algebras that directly support checking that $a \in [\![\varphi]\!]$ and use concrete values instead of atoms.

4.1 Complexity

In the following we argue that, under certain assumptions, the expected depth of the trie constructed by the search algorithm is logarithmic in the number of inequivalent predicates (leaves in the trie). Notice that distinct duplicate but equivalent predicates do not count. We show that the problem of constructing a trie with n leaves is similar to the problem of constructing a binary search tree or BST with approximately the same height and size.

In this analysis we assume that $U_{\mathcal{A}}$ is the interval $[0; K)$ for some finite K and that each predicate φ is a number in $\Psi_{\mathcal{A}} = [0; 2^K)$ representing the set of all $b \in U_{\mathcal{A}}$ such that bit b of φ is one, i.e., $\varphi \div 2^b \bmod 2 = 1$. An atom 2^b represents the set $\{b\}$. When constructing a trie from a sequence $(\varphi_i)_{i=1}^n$ of predicates, we assume that all predicates have been chosen uniformly at random from $\Psi_{\mathcal{A}}$, so all $n!$ permutations are equally probable. The test $\mathbf{Sat}(\varphi \not\Leftrightarrow \psi)$ is the test $\varphi \neq \psi$. The function *Atom*(φ) is defined as 0 if $\varphi = 0$, else 2^b where b is the least bit such that $2^b \in \varphi$.

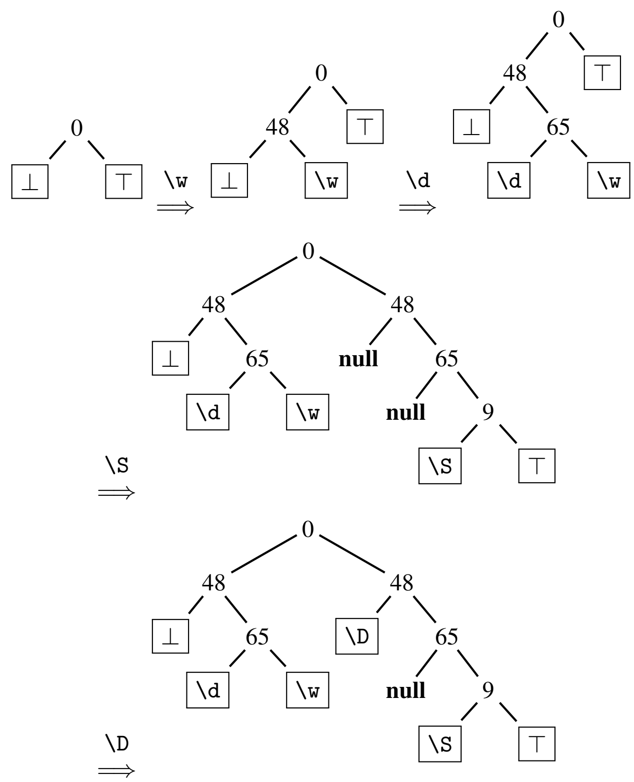
Consider a trie that has been constructed from $(\varphi_i)_{i=1}^n$. The trie orders the predicates according to $\prec_{\boldsymbol{v}}$.[7] One can systematically map the trie into a $\prec$-ordered BST, say BST$_\prec$, such that for any subtree t with root r, for all nodes x in the left subtree of t we have $x \prec r$ and for all nodes x in the right subtree of t we have $r \prec x$. BST$_\prec$ is constructed, roughly, by shifting, for each subtree t of the trie, the largest (rightmost) leaf from the left subtree of t into its root. This transformation induces a reordering $(\varphi'_i)_{i=1}^n$ of the original input sequence $(\varphi_i)_{i=1}^n$ such that inserting the elements $\varphi'_1, \varphi'_2, \ldots$ with respect to $\prec$-order produces the same BST$_\prec$. The expected height of a random BST constructed from a random equiprobable permutation of n values is $O(\log n)$, cf. [37].

[7] Observe that the standard order $<$ is the special case $\prec_{(2^0, 2^1, \ldots, 2^{K-1})}$.

EXAMPLE 2. Consider $\mathcal{A}$ as in Section 4.1 with $K = 128$. One can think of $U_{\mathcal{A}}$ as the set of all ASCII characters and a set of characters is represented by a 128-bit number. In this case let the regex character classes `\d` of *digits*, `\w` of *word letters*, and `\s` of *white-space* characters, be predicates that denote the following sets:

$$\begin{array}{rcl} [\![\texttt{\char`\\d}]\!] & = & [48; 57] \\ [\![\texttt{\char`\\w}]\!] & = & [48; 57] \cup [65; 90] \cup \{95\} \cup [97; 122] \\ [\![\texttt{\char`\\s}]\!] & = & [9; 13] \end{array}$$

Moreover, let `\D` $= \neg$`\d`, `\W` $= \neg$`\w`, and `\S` $= \neg$`\s`. Consider the input sequence (`\w`, `\d`, `\S`, `\D`) of predicates. The trie evolves as follows where the leaves are shown as boxed predicates, and internal nodes at depth $i-1$ for $i \in [1; |\boldsymbol{v}|]$ have labels a such that $\boldsymbol{v}(i) = 2^a$. The root has label 0 because *Atom*$(\top) = 2^0$.



In the end $\boldsymbol{v} = (2^0, 2^{48}, 2^{65}, 2^9)$ and `\d` $\prec$ `\w` $\prec$ `\D` $\prec$ `\S`. $\boxtimes$

5. Symbolic Weak Monadic Second Order Logic of One Successor

We define *symbolic weak monadic second-order logic of one successor on finite sequences (*S-M2L-STR*)* together with its abstract syntax and semantics. The syntax of symbolic weak monadic second-order logic of one successor (S-M2L-STR) formulas operating on words over an effective Boolean algebra $\mathcal{A}$, or M2L-STR$\langle\mathcal{A}\rangle$, is defined by the grammar

$$\varphi \quad := \quad \neg\varphi \mid \varphi \wedge \varphi \mid \exists x(\varphi) \mid \exists X(\varphi) \mid x < y \mid X(x) \mid [\alpha](x)$$

where $\alpha \in \Psi_{\mathcal{A}}$. *First-order* variables are denoted by lower case letters x, y, z and range over positions (i.e., positive integers), and *second-order* variables are denoted by upper case letters X, Y, Z and range over finite sets of positions. We write X for a variable that is either first-order or second-order. Universal quantification as well as other logical connectives are defined as usual, e.g., $\psi \Rightarrow \varphi$ is defined as $\neg\psi \vee \varphi$. Table 1 shows a set of complex formulas that can be derived from the basic syntax.

Derived formulas			Description
$x = y$	$\stackrel{\text{def}}{=}$	$\neg(x < y) \wedge \neg(y < x)$	x and y are the same positions
$\lvert X\rvert = 1$	$\stackrel{\text{def}}{=}$	$\exists x(X(x) \wedge \neg\exists y(X(y) \wedge x \neq y))$	X is a singleton set
$\mathit{succ}(x, y)$	$\stackrel{\text{def}}{=}$	$x < y \wedge \neg\exists z(x < z \wedge z < y)$	y is the successor of x
$\mathit{first}(x, X)$	$\stackrel{\text{def}}{=}$	$X(x) \wedge \neg\exists y(X(y) \wedge y < x)$	x is the first position in the set X
$\mathit{last}(x, X)$	$\stackrel{\text{def}}{=}$	$X(x) \wedge \neg\exists y(X(y) \wedge x < y)$	x is the last position in the set X
$\mathit{range}(x, y, X)$	$\stackrel{\text{def}}{=}$	$\forall z((x \leq z \wedge z \leq y) \Leftrightarrow X(z))$	the set X is the range $[x; y]$
$\mathit{range}(X)$	$\stackrel{\text{def}}{=}$	$\exists x\exists y(\mathit{range}(x, y, X))$	the set X is a range
$[\alpha](X)$	$\stackrel{\text{def}}{=}$	$\forall x(X(x) \Rightarrow [\alpha](x))$	the labels of all positions in X satisfy α
$\mathit{range}(\alpha, X)$	$\stackrel{\text{def}}{=}$	$\mathit{range}(X) \wedge [\alpha](X)$	the set X is an α-range
$X \subseteq Y$	$\stackrel{\text{def}}{=}$	$\forall x(X(x) \Rightarrow Y(x))$	X is a subset of Y
$X \subset Y$	$\stackrel{\text{def}}{=}$	$X \subseteq Y \wedge \neg(Y \subseteq X)$	X is a strict subset of Y
$\mathit{maxrange}(\alpha, X)$	$\stackrel{\text{def}}{=}$	$\mathit{range}(\alpha, X) \wedge \forall Y(X \subset Y \Rightarrow \neg\mathit{range}(\alpha, Y))$	X is a maximal α-range

Table 1. S-M2L-STR formulas derived from the basic syntax.

The operator that differentiates S-M2L-STR from M2L-STR is the unary predicate $[\alpha](x)$. In M2L-STR predicates are not really used or even needed. Instead, if there is a finite signature Σ of size at most 2^k with the intent of being used as labels, then the corresponding formula will encode each label using k free variables each corresponding to one bit of information about the label when represented as a natural number in binary format. A concrete example is the (extended) ASCII alphabet, where 8 bits are needed to represent one character, as illustrated in [29, Section 6.6]. In contrast, in S-M2L-STR the predicate α can be an arbitrary predicate of the Boolean algebra $\mathcal{A}$.

EXAMPLE 3. Given the predicate $\psi(r) = r > 0$ over the theory of linear integer arithmetic, the formula

$$\exists x_1.\exists x_2.[\psi(r)](x_1) \wedge [\psi(r)](x_2) \wedge x_1 < x_2$$

is true for all the strings $a_1 \ldots a_n \in \mathbb{N}^*$ for which there exists two positions $i, j \in [1; n]$ such that the symbols a_i and a_j are both numbers greater than 0 and i appears before j (i.e. $i < j$). ⊠

Moreover, while in M2L-STR any position x in the string over Σ satisfies exactly one predicate in Σ (i.e. each character is a fixed element of Σ, when encoded as a binary number), in our case, since no encoding is needed, each position can satisfy any subset of the unary predicates appearing in the formula.

EXAMPLE 4. For example the character $r = 6$ satisfies $\psi(r)$ above but also $\mathit{even}(r)$. Moreover, since for every two predicates $\psi_1, \psi_2 \in \Psi_{\mathcal{A}}$ the predicate $\psi_1 \wedge_{\mathcal{A}} \psi_2$ is in $\Psi_{\mathcal{A}}$, it follows that if ψ_1 and ψ_2 are unary predicates in S-M2L-STR then so is $\psi_1 \wedge_{\mathcal{A}} \psi_2$. ⊠

Semantics We formally define the semantics of M2L-STR$\langle\mathcal{A}\rangle$. Intuitively, *words* are elements of $U_{\mathcal{A}}^*$, i.e., the Kleene closure or the set of all *finite sequences* over $U_{\mathcal{A}}$. First-order variables in the formulas refer to *positions* of individual letters of words (elements of $U_{\mathcal{A}}$) and second-order variables refer to *finite sets of positions*. Predicates α in formulas $[\alpha](x)$ refer to the *letter* in position x.

Let $\{x_j\}_{j\geq 1}$ and $\{X_j\}_{j\geq 1}$ be fixed enumerations of first-order and second-order variables. Let φ be a M2L-STR$\langle\mathcal{A}\rangle$ formula such that $FV(\varphi) \subseteq \{x_j\}_{j\geq 1} \cup \{X_j\}_{j\geq 1}$ and, for any j, at most one of x_j or X_j is free in φ, otherwise φ is ill-formed.

Let $J = \{j \mid \mathrm{x}_j \in FV(\varphi)\}$ be the set of all free variable indices in φ. Let $w \in U_{\mathcal{A}}^*$ and let θ be a finite sequence of subsets of positions of w (subsets of $[1; |w|]$) such that, for all $j \in J$, $\theta(j)$ is defined, i.e., $j \leq |\theta|$. Moreover, if x_j is a free first-order variable in φ then $\theta(j)$ must be a singleton set. We define the semantics using judgements of the form $w, \theta \vDash \varphi$:

$w, \theta \vDash \neg\varphi$	$\Leftrightarrow$	$w, \theta \nvDash \varphi$
$w, \theta \vDash \varphi_1 \wedge \varphi_2$	$\Leftrightarrow$	$w, \theta \vDash \varphi_1$ and $w, \theta \vDash \varphi_2$
$w, \theta \vDash \exists x(\varphi(x))$	$\Leftrightarrow$	there exists $i \in [1; \lvert w\rvert]$ such that $w, \theta \cdot \{i\} \vDash \varphi(x_{\lvert\theta\rvert+1})$
$w, \theta \vDash \exists X(\varphi(X))$	$\Leftrightarrow$	there exists $I \subseteq [1; \lvert w\rvert]$ such that $w, \theta \cdot I \vDash \varphi(X_{\lvert\theta\rvert+1})$
$w, \theta \vDash x_i = x_j$	$\Leftrightarrow$	$\theta(i) = \theta(j)$
$w, \theta \vDash x_i < x_j$	$\Leftrightarrow$	$\varepsilon(\theta(i)) < \varepsilon(\theta(j))$
$w, \theta \vDash X_j(x_i)$	$\Leftrightarrow$	$\theta(i) \subseteq \theta(j)$
$w, \theta \vDash [\alpha](x_i)$	$\Leftrightarrow$	$w(\varepsilon(\theta(i))) \in [\![\alpha]\!]_{\mathcal{A}}$

A formula is *closed* if it has no free variables and it is *open* otherwise. Let φ be a closed formula. We write $w \vDash \varphi$ for $w, () \vDash \varphi$. The *language of* φ is the following subset of $U_{\mathcal{A}}^*$,

$$\mathscr{L}(\varphi) \stackrel{\text{def}}{=} \{w \in U_{\mathcal{A}}^* \mid w \vDash \varphi\}.$$

EXAMPLE 5. Define the following formulas, where the subformulas over positions and sets are defined as usual, e.g., $y = x + 1$ stands for $x < y \wedge \neg\exists z(x < z \wedge z < y)$.

$$\begin{array}{lcl}
\mathit{OddEven}(X, Y) & \stackrel{\text{def}}{=} & X \cup Y = \mathit{all} \wedge X \cap Y = \emptyset \wedge \\
& & \exists x(\mathit{first}(x) \wedge X(x)) \wedge \\
& & \forall x \forall y((y = x + 1 \wedge Y(y)) \Rightarrow X(x)) \wedge \\
& & \forall x \forall y((x = y + 1 \wedge X(x)) \Rightarrow Y(y)) \\
\mathit{Odd}(x) & \stackrel{\text{def}}{=} & \exists X\, \exists Y(\mathit{OddEven}(X, Y) \wedge X(x)) \\
\mathit{Even}(x) & \stackrel{\text{def}}{=} & \exists X\, \exists Y(\mathit{OddEven}(X, Y) \wedge Y(x))
\end{array}$$

Consider $\mathcal{A}$ to be integer linear arithmetic with `even` as the predicate $\lambda x.x \bmod 2 = 0$ and `odd` as the predicate $\lambda x.x \bmod 2 = 1$. Let φ be the following closed M2L-STR$\langle\mathcal{A}\rangle$ formula

$$\forall x((\mathit{Odd}(x) \Rightarrow [\mathtt{even}](x)) \wedge (\mathit{Even}(x) \Rightarrow [\mathtt{odd}](x))).$$

The language of φ, $\mathscr{L}(\varphi)$, consist of all finite sequences w of integers such that for all i, $1 \leq i \leq |w|$, $w(i)$ is odd iff i is even and $w(i)$ is even iff i is odd. ⊠

5.1 S-M2L-STR vs M2L-STR

The classic decision procedure for M2L-STR modifies formulas by replacing first-order variables with second-order variables and by adding additional constraints. This can be done using two approaches. The first approach represents first-order variables as second-order variables that always contain singleton sets—i.e., the element of the first-order variable. The second approach represents the value of each first-order variables using the minimal element of the non-empty set of a corresponding second order variable. The second approach has been taken in Mona, with the motivation that it provides slightly better performance for the automata translation [29, Section 3.2].

When dealing with formulas in M2L-STR$\langle\mathcal{A}\rangle$ there are new concerns and possibilities for optimizations that do not arise for M2L-STR. Therefore, both of the above approaches should be revisited and evaluated before committing to either one. It is also important to maintain the distinction between first-order and second-order variables as illustrated by the following two cases.

Let & denote $\wedge_{\mathcal{A}}$ and let $-$ denote $\neg_{\mathcal{A}}$. The formula $[\alpha](x_i) \wedge [\beta](x_i)$ is equivalent to $[\alpha \mathbin{\&} \beta](x_i)$ because, using $|\theta(i)| = 1$,

$$w, \theta \vDash [\alpha](x_i) \wedge [\beta](x_i) \iff w(\varepsilon(\theta(i))) \in [\![\alpha]\!]_{\mathcal{A}} \cap [\![\beta]\!]_{\mathcal{A}}$$
$$\iff w(\varepsilon(\theta(i))) \in [\![\alpha \mathbin{\&} \beta]\!]_{\mathcal{A}} \iff w, \theta \vDash [\alpha \mathbin{\&} \beta](x_i).$$

The formula $\neg[\alpha](x_i)$ is equivalent to $[-\alpha](x_i)$. This is because, using $|\theta(i)| = 1$,

$$w, \theta \vDash \neg[\alpha](x_i) \iff w, \theta \nvDash [\alpha](x_i) \iff$$
$$w(\varepsilon(\theta(i))) \notin [\![\alpha]\!]_{\mathcal{A}} \iff w(\varepsilon(\theta(i))) \in (U_{\mathcal{A}} \setminus [\![\alpha]\!]_{\mathcal{A}}) \iff$$
$$w(\varepsilon(\theta(i))) \in [\![-\alpha]\!]_{\mathcal{A}} \iff w, \theta \vDash [-\alpha](x_i).$$

Such transformations may be beneficial when the ability to push the negation into the alphabet theory can imply further simplifications. For example, if α has the form $-\beta$ then $\neg[\alpha](x_i)$ simplifies to $[\beta](x_i)$. A closer study of such transformations and implied simplifications is future work.

6. From S-M2L-STR to Symbolic Finite Automata

The classic decision procedure for M2L-STR relies on the fact that any formula $\varphi(X_1, \ldots, X_k)$ with k free variables and over a finite alphabet Σ can be compiled into a deterministic finite automaton M that accepts strings over the alphabet $\Gamma = \Sigma \times \{0,1\}^k$. Any occurrence of a character $c = (a, (b_j)_{j=1}^k) \in \Gamma$ in position i of a word w encodes the property: $i \in X_j \Leftrightarrow b_j = 1$, meaning that position i is an element of the set X_j. Thus, every string $w = (c_i)_{i=1}^n$ accepted by M, where $c_i = (a_i, (b_{i,j})_{j=1}^k)$, has the following property: if for each variable X_j we let $I_j = \{i \in [1..n] \mid b_{i,j} = 1\}$, then

$$(a_i)_{i=1}^n, (I_j)_{j=1}^k \vDash \varphi(X_1, \ldots, X_k).$$

Our decision procedure uses the same idea and transforms a M2L-STR$\langle\mathcal{A}\rangle$ formula into a symbolic finite automaton, which is a finite automaton in which edge labels are replaced by predicates. Given a M2L-STR$\langle\mathcal{A}\rangle$ our algorithm constructs a symbolic finite automaton over the alphabet $U_{\mathcal{A}} \times \mathbb{N}$. By using $\mathbb{N}$ instead of $\{0,1\}^k$ we do not need to know in advance the exact number of variables appearing in the formula. Technically this simplifies some of the constructs as we do not need to change the alphabet when k changes.

To represent the composite alphabet $U_{\mathcal{A}} \times \mathbb{N}$ we consider two approaches. First we show how we can use **BDD**$\langle\mathcal{A}\rangle$. We then discuss an alternative way to construct a *product* algebra $\mathcal{A} \boxtimes$ **BDD**$\langle$**B1**$\rangle$ that is also based on a variation of Shannon expansions.

6.1 Symbolic Finite Automata

A *symbolic finite automaton* (*s-FA*) M is a tuple $(\mathcal{A}, Q, q^0, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, called the *alphabet algebra*, Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of *transitions*. A transition $(p, \varphi, q) \in \Delta$ is also denoted by $p \xrightarrow{\varphi}_M q$ (or $p \xrightarrow{\varphi} q$).

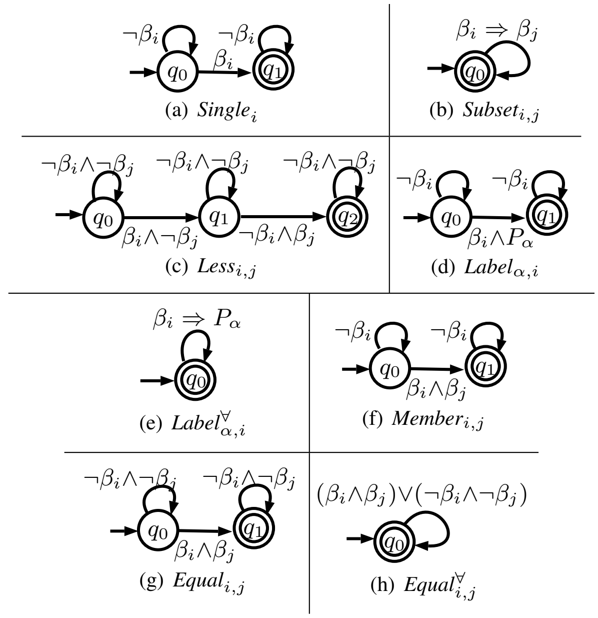


Figure 3. Basic s-FAs used as building blocks when translating M2L-STR$\langle\mathcal{A}\rangle$ formulas into s-FAs. The corresponding formula behind the s-FA is: a) $|X_i| = 1$; b) $X_i \subseteq X_j$; c) $x_i < x_j$; d) $[\alpha](x_i)$; e) $\forall x(X_i(x) \Rightarrow [\alpha](x))$; f) $X_j(x_i)$; g) $x_i = x_j$; h) $X_i = X_j$.

A word $w = (a_i)_{i=1}^k \in U_{\mathcal{A}}^*$, is *accepted from state* p *of* M, denoted $w \in \mathscr{L}_p(M)$, if either $w = \epsilon$ and $p \in F$, or there exist $p_{i-1} \xrightarrow{\varphi_i}_M p_i$, $a_i \in [\![\varphi_i]\!]$, for $1 \le i \le k$, such that $p_0 = p$, and $p_k \in F$. The *language of* M is $\mathscr{L}(M) \stackrel{\text{def}}{=} \mathscr{L}_{q^0}(M)$.

s-FAs are closed under Boolean operations [42] and we use $\oplus, \otimes$, and $^{\complement}$ to denote their union, intersection, and complement operations, respectively. We will use these Boolean operations over s-FAs when translating S-M2L-STR formulas to s-FAs.

6.2 From S-M2L-STR to s-FA using the Generic BDD Algebra

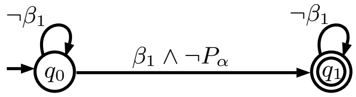
We define a translation ***sFA***$(\cdot)$ from M2L-STR$\langle\mathcal{A}\rangle$ to s-FAs over **BDD**$\langle\mathcal{A}\rangle = (U_{\mathcal{A}} \times \mathbb{N}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$. Let

$$\beta_i \stackrel{\text{def}}{=} \mathit{node}(i, \top, \bot), \quad P_\alpha \stackrel{\text{def}}{=} \mathit{leaf}(\alpha).$$

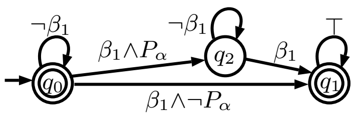
P_α means that for all $(a, n) \in [\![P_\alpha]\!]$, a satisfies α, and β_i means that for all $(a, n) \in [\![\beta_i]\!]$, bit i of n is 1. Recall our convention that the lowest bit is 1. Observe that $\neg\beta_i = \mathit{node}(i, \bot, \top)$.

The s-FAs used for constructing the core basic formulas are shown in Figure 3. For example, the s-FA depicted in Figure 3(e) describes all the words over the alphabet $U_{\mathcal{A}} \times \mathbb{N}$ such that for every character (a, n) either a satisfies α and the i-th bit of n is 1, or the i-th bit of n is 0.

We define a transformation to restrict the number of relevant bits in the domain of an s-FA M over **BDD**$\langle\mathcal{A}\rangle$ and we use it for correctly modelling existential quantifier in our translation. The transformation behaves as follows: for $k \ge 1$, it restricts the number of relevant bits of $\mathbb{N}$ of all guards to at most $k - 1$, and for $k = 0$, it projects away the BDD component completely. First,



(a) $\boldsymbol{sFA}(\neg[\alpha](x_1))$



(b) $\boldsymbol{sFA}([\alpha](x_1))^{\complement}$

Figure 4. Sample s-FAs.

we define the restriction for predicates in $\Psi_{\mathbf{BDD}\langle\mathcal{A}\rangle}$:[8]

$$\mathit{leaf}(\alpha){\restriction}0 \stackrel{\text{def}}{=} \alpha,\ \mathit{leaf}(\alpha){\restriction}k \stackrel{\text{def}}{=} \mathit{leaf}(\alpha) \quad (\text{if } k > 0),$$

$$\mathit{node}(i,\varphi,\psi){\restriction}k \stackrel{\text{def}}{=} \begin{cases} \mathit{node}(i,\varphi,\psi), & \text{if } i < k; \\ \varphi \vee \psi, & \text{if } i = k; \\ \varphi{\restriction}k \vee \psi{\restriction}k, & \text{otherwise.} \end{cases}$$

Observe that, for $k = 0$ the restriction operation produces a predicate in $\Psi_{\mathcal{A}}$. For 0 and $k \geq 1$, the restriction operation is lifted to all $M \in \Psi_{\mathcal{M}\langle\mathbf{BDD}\langle\mathcal{A}\rangle\rangle}$.

$$M{\restriction}0 \stackrel{\text{def}}{=} (\mathcal{A}, Q_M, q_M^0, F_M, \{(p, \varphi{\restriction}0, q) \mid (p,\varphi,q) \in \Delta_M\}),$$
$$M{\restriction}k \stackrel{\text{def}}{=} (\mathbf{BDD}\langle\mathcal{A}\rangle, Q_M, q_M^0, F_M, \{(p, \varphi{\restriction}k, q) \mid (p,\varphi,q) \in \Delta_M\}).$$

We can now define the translation of the formulas as follows, where the atomic formulas use the s-FAs from Figure 3. We only consider normalized formulas, where a formula φ is *normalized* if: 1) all variables in φ belong to $\{x_i\}_{i\geq 1} \cup \{X_i\}_{i\geq 1}$; 2) there is no i and subformula ψ of φ such that $x_i, X_i \in FV(\psi)$; 3) for all subformulas $\exists \mathrm{X}_i\psi$ of φ we have $i = \max\{j \mid \mathrm{X}_j \in FV(\psi)\}$.[9]

$$\begin{aligned}
\boldsymbol{sFA}(x_i = x_j) &\stackrel{\text{def}}{=} \mathit{Equal}_{i,j} \\
\boldsymbol{sFA}(\varphi_1 \wedge \varphi_2) &\stackrel{\text{def}}{=} \boldsymbol{sFA}(\varphi_1) \otimes \boldsymbol{sFA}(\varphi_2) \\
\boldsymbol{sFA}(X_j(x_i)) &\stackrel{\text{def}}{=} \mathit{Member}_{i,j} \\
\boldsymbol{sFA}(\neg\varphi) &\stackrel{\text{def}}{=} \boldsymbol{sFA}(\varphi)^{\complement} \otimes \textstyle\bigotimes_{x_i \in FV(\varphi)} \mathit{Single}_i \\
\boldsymbol{sFA}([\alpha](x_i)) &\stackrel{\text{def}}{=} \mathit{Label}_{\alpha,i} \\
\boldsymbol{sFA}(\exists \mathrm{X}_i\varphi) &\stackrel{\text{def}}{=} \boldsymbol{sFA}(\varphi){\restriction}i \\
\boldsymbol{sFA}(x_i < x_j) &\stackrel{\text{def}}{=} \mathit{Less}_{i,j}
\end{aligned}$$

In the $\exists$-case, $i = \max\{j \mid \mathrm{X}_j \in FV(\varphi)\}$, so all transition guards in $\boldsymbol{sFA}(\varphi)$ have position $\leq i$. Therefore, $\boldsymbol{sFA}(\varphi){\restriction}i$ eliminates constraints only on i.

Only four of the eight basic SFAs in Figure 3 are needed by the core transformation rules of $\boldsymbol{sFA}(\cdot)$. The implementation uses the other basic SFAs as shortcuts in translation rules of common cases such as $\boldsymbol{sFA}(X_i = X_j) \stackrel{\text{def}}{=} \mathit{Equal}^{\forall}_{i,j}$ instead of the equivalent

$$\boldsymbol{sFA}(\neg\exists x \neg((X_i(x) \wedge X_j(x)) \vee (\neg X_i(x) \wedge \neg X_j(x)))).$$

EXAMPLE 6. Figure 4 compares $\boldsymbol{sFA}(\neg[\alpha](x_1))$ to $\boldsymbol{sFA}([\alpha](x_1))^{\complement}$ to illustrate why the additional singleton restriction Single_1 is needed: $\boldsymbol{sFA}([\alpha](x_1))^{\complement}$ treats x_1 as if it was second-order. ⊠

We need the following additional definitions to state the correctness theorem of the translation and to precisely relate the language of a formula to the language accepted by the corresponding s-FA. We first define the language of normalized open formulas φ. To this end, given a judgement $w, \theta \vDash \varphi$ we let $w^{(\theta)}$ denote the following subset of $(U_{\mathcal{A}} \times \mathbb{N})^*$. Let $w = \bar{a} = (a_i)_{i=1}^n$ and $\theta = \bar{X} = (X_j)_{j=1}^k$.

$$\bar{a}^{(\bar{X})} \stackrel{\text{def}}{=} \{((a_1, m_1), \ldots, (a_n, m_n)) \mid \text{for all } i \in [1..n], j \in [1..k] : \mathit{Bit}(j, m_i) \Leftrightarrow i \in X_j\}$$

The intuition is that $w^{(\theta)}$ is a lifting of the word w from the alphabet $U_{\mathcal{A}}$ to the alphabet $U_{\mathcal{A}} \times \mathbb{N}$ such that, for each position i of w, letter $a_i = w(i)$ in potion i is lifted to (a_i, m_i) where m_i is a number that encodes which variables X_j include i and which don't. If the j'th bit of m_i is 1 (resp. 0) then this means that $i \in X_j$ (resp. $i \notin X_j$). All bits above k are irrelevant and can have any value. For example, if there is a single variable X_1 ($k = 1$) then what matters is only the first bit of m_i, then, if m_i is odd then $i \in X_1$ else $i \notin X_1$. Let

$$\mathscr{L}_{\text{open}}(\varphi) \stackrel{\text{def}}{=} \bigcup\{w^{(\theta)} \mid w, \theta \vDash \varphi\}.$$

The following lemma relates the language of an open formula to the language of the corresponding s-FA and it is proved by case analysis and induction over the structure of M2L-STR$\langle\mathcal{A}\rangle$ formulas.

LEMMA 1. *For normalized φ, $\mathscr{L}_{open}(\varphi) = \mathscr{L}(\boldsymbol{sFA}(\varphi))$.*

We can now state our main theorem, which follows from Lemma 1 and the property that the guards of all transitions in $\boldsymbol{sFA}(\varphi)$ have the form $\mathit{leaf}(\alpha)$ when φ is closed.

THEOREM 1. *For closed φ, $\mathscr{L}(\varphi) = \mathscr{L}(\boldsymbol{sFA}(\varphi){\restriction}0)$.*

Nonempty set semantics The s-FAs that are shown in Figure 3 are based on the *singleton-set* semantics of first-order variables. An alternative is to consider first-order variables as minimal elements of nonempty sets. With such *minimum-of-nonempty-set* semantics some of the s-FAs would be different, e.g., in Figure 3(g) the predicate on the q_1-loop would be $\top$, because (with respect to the assumption that we only care about the minimal elements) $x_i = x_j$ would be interpreted as $\min(X_i) = \min(X_j)$, so the positions occurring in either X_i or X_j after state q_1 do not matter. Other s-FAs involving first-order variables would be affected similarly.

The overall translation is affected so that Single_i is replaced by $\mathit{Nonempty}_i$, the latter is a modification of the former from Figure 3(a) with the label of the q_1-loop replaced by $\top$. Observe that $\mathit{Nonempty}_i$ checks precisely that the *minimum* position exists in X_i and does not care about any other positions (thus the label $\top$ after state q_1). Mona uses this latter approach [29].

6.3 From S-M2L-STR to S-FA using a Product Algebra

Given two effective Boolean algebras $\mathcal{A}$ and $\mathcal{B}$, a *Cartesian product for* $(\mathcal{A}, \mathcal{B})$ is any effective Boolean algebra $\mathcal{C}$, with $U_{\mathcal{C}} = U_{\mathcal{A}} \times U_{\mathcal{B}}$, that is associated with effective predicate transformers $\mathbf{D}$ (decompose) and $\mathbf{C}$ (compose) such that for all $\psi \in \Psi_{\mathcal{C}}$ the *sum of products* decomposition $\mathbf{D}(\psi)$ is a finite subset of $\Psi_{\mathcal{A}} \times \Psi_{\mathcal{B}}$ and conversely, for every finite $F \subseteq \Psi_{\mathcal{A}} \times \Psi_{\mathcal{B}}$ we have $\mathbf{C}(F) \in \Psi_{\mathcal{C}}$ where

$$[\![\psi]\!]_{\mathcal{C}} = \bigcup\{[\![\alpha]\!]_{\mathcal{A}} \times [\![\beta]\!]_{\mathcal{B}} \mid (\alpha, \beta) \in \mathbf{D}(\psi)\} = [\![\mathbf{C}(\mathbf{D}(\psi))]\!]_{\mathcal{C}}$$

For example, it is easy to see from the definitions that we can effectively transform any predicate in $\psi \in \Psi_{\mathbf{BDD}\langle\mathcal{A}\rangle}$ into $\mathbf{D}(\psi) = \{(\alpha_i, \beta_i)\}_{i=1}^k$ where all $\alpha_i \in \Psi_{\mathcal{A}}$ and $\beta_i \in \Psi_{\mathbf{BDD}\langle\mathbf{B1}\rangle}$. Thus, $\mathbf{BDD}\langle\mathcal{A}\rangle$ is (a particular implementation of) a Cartesian product for $(\mathcal{A}, \mathbf{BDD}\langle\mathbf{B1}\rangle)$.

An abstract (and highly impractical) definition of a Cartesian product for $(\mathcal{A}, \mathcal{B})$, denoted $\mathcal{A}{\times}\mathcal{B}$, can be given as follows: $U_{\mathcal{A}\times\mathcal{B}} = U_{\mathcal{A}} \times U_{\mathcal{B}}$, $\Psi_{\mathcal{A}\times\mathcal{B}}$ contains all finite nonempty subsets of $\Psi_{\mathcal{A}} \times \Psi_{\mathcal{B}}$ representing *sums of products* or *DNFs*

[8] Observe that both "$i = k$" and "otherwise" are needed, the latter ($i > k$) is the recursive case of restriction whereas the former stops the restriction.

[9] Normalization is a simple syntactic variable renaming procedure.

$$\varphi \wedge \psi \stackrel{\text{def}}{=} \textsc{Merge}(\varphi, \psi, \top_{\mathcal{A}})$$

$\textsc{Merge}(\varphi : \Psi_{\mathcal{A}\boxtimes\mathcal{B}}, \psi : \Psi_{\mathcal{A}\boxtimes\mathcal{B}}, \pi : \Psi_{\mathcal{A}}) : \Psi_{\mathcal{A}\boxtimes\mathcal{B}}$

```
match φ
    case ⌊α, φ1, φ2⌋
        let ψ1 = Merge(φ1, ψ, π ∧_A α)
        let ψ2 = Merge(φ2, ψ, π ∧_A ¬_A α)
        return ⌊α, ψ1, ψ2⌋
    case ⌊β⌋
        match ψ
            case ⌊α, ψ1, ψ2⌋
                let π1 = π ∧_A α
                let π2 = π ∧_A ¬_A α
                if not Sat_A(π1)
                    return Merge(φ, ψ2, π)
                else if not Sat_A(π2)
                    return Merge(φ, ψ1, π)
                else
                    let ϕ1 = Merge(φ, ψ1, π1)
                    let ϕ2 = Merge(φ, ψ2, π2)
                    return ⌊α, ϕ1, ϕ2⌋
            case ⌊γ⌋
                if Sat_B(β ∧_B γ)
                    return ⌊β ∧_B γ⌋
                else
                    return ⌊⊥_B⌋
```

Figure 5. Conjuction $\varphi \wedge \psi$ in $\mathcal{A} \boxtimes \mathcal{B}$.

(disjunctive normal forms), and for $\psi \in \Psi_{\mathcal{A}\times\mathcal{B}}$, $[\![\psi]\!]_{\mathcal{A}\times\mathcal{B}} = \bigcup_{(\alpha,\beta)\in\psi} [\![\alpha]\!]_{\mathcal{A}} \times [\![\beta]\!]_{\mathcal{B}}$. The definition of $\perp_{AxB}$ follows the well-formedness convention (used later) that the first component is always satisfiable: $\perp_{\mathcal{A}\times\mathcal{B}}$ is $\{(\top_{\mathcal{A}}, \perp_{\mathcal{B}})\}$; $\top_{\mathcal{A}\times\mathcal{B}}$ is $\{(\top_{\mathcal{A}}, \top_{\mathcal{B}})\}$; $\vee_{\mathcal{A}\times\mathcal{B}}$ is the union of the two sets, $\neg_{\mathcal{A}\times\mathcal{B}}$ applies De Morgans laws and computes the DNF, and $\wedge_{\mathcal{A}\times\mathcal{B}}$ is defined in terms of $\vee_{\mathcal{A}\times\mathcal{B}}$ and $\neg_{\mathcal{A}\times\mathcal{B}}$. The following proposition is immediate.

PROPOSITION 2. *If $\mathcal{A}$ and $\mathcal{B}$ are effective Boolean algebras then so is $\mathcal{A} \times \mathcal{B}$.*

In the following, we consider another implementation of a Cartesian product algebra for $(\mathcal{A}, \mathcal{B})$, denoted by $\mathcal{A} \boxtimes \mathcal{B}$, that does not rely on $\mathcal{B}$ being $\mathbf{BDD}\langle\mathbf{B1}\rangle$. In Section 6.4 we discuss the key differences between these two implementations, as Cartesian product algebras for $(\mathcal{A}, \mathbf{BDD}\langle\mathbf{B1}\rangle)$.

***Algebra* $\mathcal{A} \boxtimes \mathcal{B}$.** For an efficient implementation of predicates in $\Psi_{\mathcal{A}\boxtimes\mathcal{B}}$ we use *If-Then-Else* expressions or *ITEs*. An *ITE for* $(\mathcal{A}, \mathcal{B})$ is either

- a *terminal* $\lfloor\beta\rfloor$ with $\beta \in \Psi_{\mathcal{B}}$, or
- a *nonterminal* $\lfloor\alpha, \varphi_1, \varphi_2\rfloor$ where $\alpha \in \Psi_{\mathcal{A}}$ and φ_1, φ_2 are ITEs.

In other words, an ITE for $(\mathcal{A}, \mathcal{B})$ is a Shannon expansion whose terminal predicates belong to $\Psi_{\mathcal{B}}$ and whose nonterminal predicates belong to $\Psi_{\mathcal{A}}$. We have $\top_{\mathcal{A}\boxtimes\mathcal{B}} \stackrel{\text{def}}{=} \lfloor\top_{\mathcal{B}}\rfloor$ and $\perp_{\mathcal{A}\boxtimes\mathcal{B}} \stackrel{\text{def}}{=} \lfloor\perp_{\mathcal{B}}\rfloor$.

Let $\psi \in \Psi_{\mathcal{A}\boxtimes\mathcal{B}}$. We define the *path condition to a node in* ψ as a predicate in $\Psi_{\mathcal{A}}$ as follows: the path condition to the root of ψ is $\top_{\mathcal{A}}$, and if the path condition to a node $\lfloor\alpha, \varphi_1, \varphi_2\rfloor$ in ψ is π, then the path condition to φ_1 is $\pi \wedge_{\mathcal{A}} \alpha$ and the path condition to φ_2 is $\pi \wedge_{\mathcal{A}} \neg_{\mathcal{A}}\alpha$.

The pair (π, β) where π is the path condition to a terminal $\lfloor\beta\rfloor$ of ψ is called a *branch* of ψ. Define $\mathbf{D}(\psi)$ as the set of all branches of ψ. We say that ψ is *reduced* if for all $(\pi, \beta) \in \mathbf{D}(\psi)$:

- $\mathbf{Sat}_{\mathcal{A}}(\pi)$, and
- either $\beta \in \{\perp_{\mathcal{B}}, \top_{\mathcal{B}}\}$ or both $\mathbf{Sat}_{\mathcal{B}}(\beta)$ and $\mathbf{Sat}_{\mathcal{B}}(\neg_{\mathcal{B}}\beta)$.

For example, the ITE $\lfloor\top_{\mathcal{A}}, \lfloor\perp_{\mathcal{B}}\rfloor, \lfloor\top_{\mathcal{B}}\rfloor\rfloor$ is not reduced because, in the branch $(\neg_{\mathcal{A}}\top_{\mathcal{A}}, \top_{\mathcal{B}})$, the path condition $\neg_{\mathcal{A}}\top_{\mathcal{A}}$ is unsatisfiable. Observe that both $\lfloor\top_{\mathcal{B}}\rfloor$ and $\lfloor\perp_{\mathcal{B}}\rfloor$ are trivially reduced.

The two key operations over ITEs, much like for BDDs, are negation and conjunction. Negation $\neg = \neg_{\mathcal{A}\boxtimes\mathcal{B}}$ is defined as follows.

$$\neg\lfloor\beta\rfloor \stackrel{\text{def}}{=} \lfloor\neg_{\mathcal{B}}\beta\rfloor, \quad \neg\lfloor\alpha, t, f\rfloor \stackrel{\text{def}}{=} \lfloor\alpha, \neg t, \neg f\rfloor$$

Trivially, negated ITEs are also reduced, provided that $\neg_{\mathcal{B}}\perp_{\mathcal{B}}$ is $\top_{\mathcal{B}}$ and $\neg_{\mathcal{B}}\top_{\mathcal{B}}$ is $\perp_{\mathcal{B}}$. Conjunction is defined in Figure 5 and uses satisfiability checks in $\mathcal{A}$ and $\mathcal{B}$ to maintain the property that the output formula is reduced if the inputs are reduced.

The denotation function $[\![\psi]\!]$ for $\psi \in \Psi_{\mathcal{A}\times\mathcal{B}}$ is defined in the obvious way, as well as the composition operator, so that $[\![\psi]\!] = [\![\mathbf{C}(\mathbf{D}(\psi))]\!]$.

6.4 $\mathbf{BDD}\langle\mathcal{A}\rangle$ vs $\mathcal{A} \boxtimes \mathbf{BDD}\langle\mathbf{B1}\rangle$

The main structural difference between $\mathbf{BDD}\langle\mathcal{A}\rangle$ and $\mathcal{A}\boxtimes\mathbf{BDD}\langle\mathbf{B1}\rangle$ is the following: in $\mathbf{BDD}\langle\mathcal{A}\rangle$ the leaves are predicates from $\Psi_{\mathcal{A}}$ while the internal nodes are bit-branches, but in $\mathcal{A} \boxtimes \mathbf{BDD}\langle\mathbf{B1}\rangle$ the internal nodes are predicates from $\Psi_{\mathcal{A}}$ and the leaves are BDDs. Essentially, the representations are turned upside-down (relatively to each other).[10]

For the effective Boolean algebra $\mathcal{C} = \mathcal{A} \boxtimes \mathbf{BDD}\langle\mathbf{B1}\rangle$ we define the restriction operation $\upharpoonright$ as follows, let $\psi \in \Psi_{\mathcal{C}}$ and $k \geq 1$; ψ is either a leaf $\lfloor\beta\rfloor$ or a node $\lfloor\alpha, t, f\rfloor$,

$$\lfloor\beta\rfloor{\upharpoonright}k \stackrel{\text{def}}{=} \lfloor\beta{\upharpoonright}k\rfloor, \quad \lfloor\alpha, t, f\rfloor{\upharpoonright}k \stackrel{\text{def}}{=} \lfloor\alpha, t{\upharpoonright}k, f{\upharpoonright}k\rfloor$$

So in $\mathcal{C}$ we have that $\psi{\upharpoonright}k$ restricts the positions of all the *terminal* predicates in ψ to $< k$ for $k \geq 1$. As the special case, for $k = 0$, $\psi{\upharpoonright}k$ is defined as a predicate in $\mathcal{A}$:

$$\begin{aligned} \lfloor\beta\rfloor{\upharpoonright}0 &\stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{A}}, & \text{if } \beta{\upharpoonright}0 = \mathbf{1}; \\ \perp_{\mathcal{A}}, & \text{otherwise.} \end{cases} \\ \lfloor\alpha, t, f\rfloor{\upharpoonright}0 &\stackrel{\text{def}}{=} (\alpha \wedge_{\mathcal{A}} t{\upharpoonright}0) \vee_{\mathcal{A}} (\neg_{\mathcal{A}}\alpha \wedge_{\mathcal{A}} f{\upharpoonright}0), \end{aligned}$$

where $\beta{\upharpoonright}0$ is always either $\mathbf{0}$ or $\mathbf{1}$ (recall the definition of $\mathbf{B1}$ from Section 3.1).

While in $\mathbf{BDD}\langle\mathcal{A}\rangle$ the predicate trie algorithm is used to keep the representation *canonical*, in $\mathcal{C}$ the leaves are canonical by virtue of $\mathbf{BDD}\langle\mathbf{B1}\rangle$ but the predicates are in general not canonical. Instead, the main effort in constructing the predicates in $\mathcal{C}$ goes into keeping them *reduced*. Efficiency of the *restriction* operation $\upharpoonright$ is critical in both cases. For $\mathcal{C}$, let

$$\beta_i \stackrel{\text{def}}{=} \lfloor \mathit{node}(i, \mathbf{1}, \mathbf{0})\rfloor, \quad P_\alpha \stackrel{\text{def}}{=} \lfloor\alpha, \top_{\mathcal{C}}, \perp_{\mathcal{C}}\rfloor.$$

In the translation from M2L-STR$\langle\mathcal{A}\rangle$ to s-FA over $\mathcal{A} \boxtimes \mathbf{BDD}\langle\mathbf{B1}\rangle$, if we replace the predicates β_i and P_α in Figure 3 with the ones defined above, the algorithm $\boldsymbol{sFA}(\cdot)$ remains identical to the case of $\mathbf{BDD}\langle\mathcal{A}\rangle$. Lemma 1 and Theorem 1 carry over to $\mathcal{C}$.

EXAMPLE 7. Let φ be the predicate $\beta_5 \wedge \neg\beta_8 \wedge P_\alpha$ for some $\alpha \in \Psi_{\mathcal{A}}$. As a predicate of $\mathcal{A} \boxtimes \mathbf{BDD}\langle\mathbf{B1}\rangle$, φ has the following ITE representation where leaves are BDDs:

$$\lfloor\alpha, \lfloor \mathit{node}(8, \mathit{leaf}(\mathbf{0}), \mathit{node}(5, \mathit{leaf}(\mathbf{1}), \mathit{leaf}(\mathbf{0})))\rfloor, \lfloor \mathit{leaf}(\mathbf{0})\rfloor\rfloor$$

As a predicate of $\mathbf{BDD}\langle\mathcal{A}\rangle$, φ has the following representation:

$$\mathit{node}(8, \mathit{leaf}(\perp_{\mathcal{A}}), \mathit{node}(5, \mathit{leaf}(\alpha), \mathit{leaf}(\perp_{\mathcal{A}})))$$

where the terminals (of this ADD) are predicates of $\mathcal{A}$. $\boxtimes$

[10] Formally, one may also consider $\mathbf{BDD}\langle\mathbf{B1}\rangle \boxtimes \mathcal{A}$, but this would devoid the benefits of structural sharing in $\mathbf{BDD}\langle\mathcal{A}\rangle$.

7. Implementation

Our S-M2L-STR solver is implemented in C# and it uses the Microsoft Automata library [5] for building the necessary s-FAs. The solver provides an interface for specifying custom Boolean algebras and it can be easily integrated with externally specified alphabet theories. The implementation is already integrated with the SMT solver Z3 [19] and can therefore handle all the complex theories Z3 supports.

Generic BDDs We implemented our own version of binary and algebraic decision diagrams for the generic BDD algebra. One of the core differences to existing implementations is that the number of bits in $\mathbf{BDD}\langle\mathcal{A}\rangle$ is unbounded which makes the Boolean algebra *atomless* and, while unboundedness simplifies some aspects of the implementation, it also limits the application of some of the optimizations available when the number of bits is bounded (the classical case) and the algebra is *atomic*. Another core difference is the use of the predicate trie in the leaf algebra. The problem of having to check for equivalence of terminals does not occur in algebraic decision diagrams where the terminals are concrete elements, and not predicates.

Product algebras Both the algebra $\mathbf{BDD}\langle\mathcal{A}\rangle$ and the algebra $\mathcal{A}\boxtimes\mathbf{BDD}\langle\mathbf{B1}\rangle$ implement the interface of the abstract product algebra $\mathcal{A}\times\mathcal{B}$ (where $\mathcal{B}$ is $\mathbf{BDD}\langle\mathbf{B1}\rangle$) and the restriction operation. The latter is used for "forgetting" bits for $\exists$-elimination. The implementation of the s-FA translation $\boldsymbol{sFA}(\cdot)$ is solely based on that interface, thus hiding the internal differences between the two algebras. This separation of concerns was crucial for fast experimentation and for dismissing some of our earlier attempts, such as a naive implementation of $\mathcal{A}\times\mathcal{B}$ outlined in Section 6.3, and, we believe, will also be useful for future experiments.

s-FA operations The s-FA operations admit a whole range of optimizations that we have not discussed. Classically, the favoured approach for implementing automata translations fuses predicates and transitions into a single system of multi-terminal BDDs or MTBDDs where the terminals are the states of the automaton. Mona, for example, uses this approach [29, 30]. This means that the underlying automaton must always be deterministic. One advantage is that, for a single start state, there may be multiple target states, which admits predicate sharing (from given source states); one disadvantage is that if the same predicate occurs for different target states, its structure is not shared. In contrast to Mona, for example, we cannot share predicates from the same state, but we do share predicates on different transitions, because the target state is not an integrated part of the predicate.

Minimizations plays a key role in our implementation and we determinize and minimize all the s-FAs using the algorithm presented in [16]. Our translation does not require the s-FAs to be deterministic and can potentially work with nondeterministic s-FAs. Determinization is only needed for complementation and, as part of our ongoing work, we are investigating ways to apply minimization directly to nondeterminisic s-FAs as it is done for NFAs in [1].

8. Experiments

We evaluate our decision procedure using both the representations of $U_{\mathcal{A}}\times\mathbb{N}$ we described in Section 6. We perform the following three experiments.

1. We compare our solver against the M2L-STR solver Mona [25] using more than 10,000 M2L-STR formulas taken from papers on M2L-STR [16, 22] and program verification [21]. This experiment shows that, even though our algorithm is general and handles complex alphabets, it has comparable expressiveness to existing M2L-STR solvers when considering formulas over finite alphabets (Section 8.1).
2. We check satisfiability of S-M2L-STR formulas over $\mathbf{BDD}\langle\mathbf{B1}\rangle$ containing unary predicates with large sets of satisfiable Boolean combinations.[11] This experiment shows how the symbolic algorithms for S-M2L-STR are in certain cases preferable to using a reduction to M2L-STR that builds the set of all maximal satisfiable Boolean combinations of the unary predicates appearing in the formula (Section 8.2).
3. We check satisfiability of randomly generated S-M2L-STR formulas over the theory of linear integer arithmetic. This experiment shows how, in the presence of complex alphabet theories, even if the set of maximal satisfiable Boolean combinations of the unary predicates appearing in the formula is relatively small, our decision procedure can be faster the reduction to M2L-STR that builds the set of all maximal satisfiable Boolean combinations of the unary predicates appearing in the formula (Section 8.3).

In the following we use generic-BDD to refer to the algebra $\mathbf{BDD}\langle\mathcal{A}\rangle$ presented in Section 6.2 and product to refer to the algebra $\mathcal{A}\boxtimes\mathbf{BDD}\langle\mathbf{B1}\rangle$ presented in Section 6.3. All the experiments were run on iMac 5K, 4GHz Intel Core i7, with 32 GB of memory. Since the code is written in C# the experiments were run on a Windows Virtual Machine with 22 GB of memory.

8.1 M2L-STR Benchmarks

The goal of this experiment is to show whether how our prototype solver, which can handle complex theories and the logic S-M2L-STR, has comparable performances to existing M2L-STR solvers in the case of finite alphabets. In particular, do the proposed algebras introduce overhead when dealing with small finite alphabets?

We compare our solver against the M2L-STR solver Mona [25]. While new M2L-STR solvers have recently been investigated [22], Mona, which was introduced in the late 90s, remains competitive on many benchmarks, is still maintained, and is adopted in many verification applications [33].

We consider three classes of M2L-STR formulas from the literature.

- 126 M2L-STR formulas over small finite alphabets used in Section 6.5 of [16] to measure how different minimization algorithms affect the classic decision procedure for M2L-STR over a finite alphabet.
- 48 M2L-STR formulas generated in [22] to evaluate the effectiveness of antichain-based algorithm in solving M2L-STR formulas.
- 10,048 LTL formulas used to evaluate antichain-based algorithms for Buchi automata in [21]. While the semantics of LTL is typically defined over infinite strings, recently there has also been interest in the interpretation of LTL over finite traces [18] as this variant can be used in applications such as program monitoring. We use LTL-F to refer to the interpretation of LTL over finite traces. LTL-F has been proven to be equivalent to first-order logic over strings and in this experiment we use the encoding proposed in [18] for our experiments.

In total we evaluate our algorithm on approximately 10,200 formulas which we are further described in Table 2. Given the large number of experiments, we set the timeout at 5 seconds. The results are shown in Fig. 7.

Results Mona is overall faster than our solver on most instances, but in most cases the performance difference is relatively small. The S-M2L-STR algorithms are faster than Mona on the first three classes of formulas described in Table 2 and on some of the random

[11] Notice that the alphabet of $\mathbf{BDD}\langle\mathbf{B1}\rangle$ is $\mathbb{N}$ and is therefore infinite.

	Name	Formula	Parameters
[16]	t1	$\exists x_1,\dots,x_k.\ x_1 < x_2 \wedge \dots \wedge x_{k-1} < x_k$	$k \in \{2,\dots,40\}$
	t2	$\exists x_1,\dots,x_k.\ x_1 < x_2 \wedge \dots \wedge x_{k-1} < x_k \wedge a(x_1) \wedge \dots \wedge a(x_k)$	$k \in \{2,\dots,40\}$
	t3	$\exists y.c(y) \vee \exists x_1,\dots,x_k.\ x_1 < x_2 \wedge \dots \wedge x_{k-1} < x_k \wedge a(x_1) \wedge \dots \wedge a(x_k)$	$k \in \{2,\dots,40\}$
	t4	$\exists x_1,\dots,x_k.((x_1 < x_2 \wedge a(x_1)) \vee c(x_1)) \vee \dots \vee ((x_{k-1} < x_k \wedge a(x_{k-1})) \vee c(x_{k-1}))$	$k \in \{2,\dots,12\}$
[22]	horn_sub	$\exists Y \forall X_1 \exists X_2 \dots \forall X_k \exists X_{k+1},\dots,X_{k+9}.\bigwedge_{1\le i\le k+9}(X_i \subseteq Y \wedge X_i \subset X_{i+1}) \rightarrow X_{i+1} \subseteq Y$	$k \in \{1,\dots,6\}$
	horn_trans	$\forall Y \exists X_1,\dots,X_k.\neg \bigwedge_{1\le i,j,l\le k} X_i \subseteq Y \wedge (X_i \subseteq X_j \wedge X_j \subseteq X_l) \rightarrow X_i \subseteq X_l$	$k \in \{3,\dots,20\}$
	set_obvious	$\exists X_1,\dots,X_k \forall Y.\bigwedge_{1\le i\le k}(Y \subset Y) \rightarrow X_i \not\subseteq X$	$k \in \{1,\dots,12\}$
	set_singletons	$\exists X_1,\dots,X_k \forall x,y.\bigwedge_{1\le i\le k}(x \in X_i \wedge y \in X_i) \rightarrow x = y$	$k \in \{1,\dots,7\}$
	set_closed	$\exists X_1,\dots,X_k \forall x \exists y,z.\neg \bigwedge_{1\le i\le k}(x \in X_i \wedge x \le y \wedge y \le z \wedge z \in X_i) \rightarrow y \in X_i$	$k \in \{1,\dots,5\}$
LTL-F [21]	counter	binary counter of length k [38]	$k \in \{2,\dots,16\}$
	counter-l	binary counter of length k, version 2 [38]	$k \in \{2,\dots,16\}$
	lift	linear encoding of a lift system with k floors [24]	$k \in \{2,\dots,9\}$
	lift-b	logarithmic encoding of a lift system with k floors [24]	$k \in \{2,\dots,9\}$
	szymanski	liveness properties of increasing size for the Szymanski mutual exclusion protocol [39]	$k \in \{1,\dots,4\}$
	random	randomly generated LTL formulas of size varying between 10 and 100 [15]	10,000 total

Table 2. M2L-STR benchmark formulas.

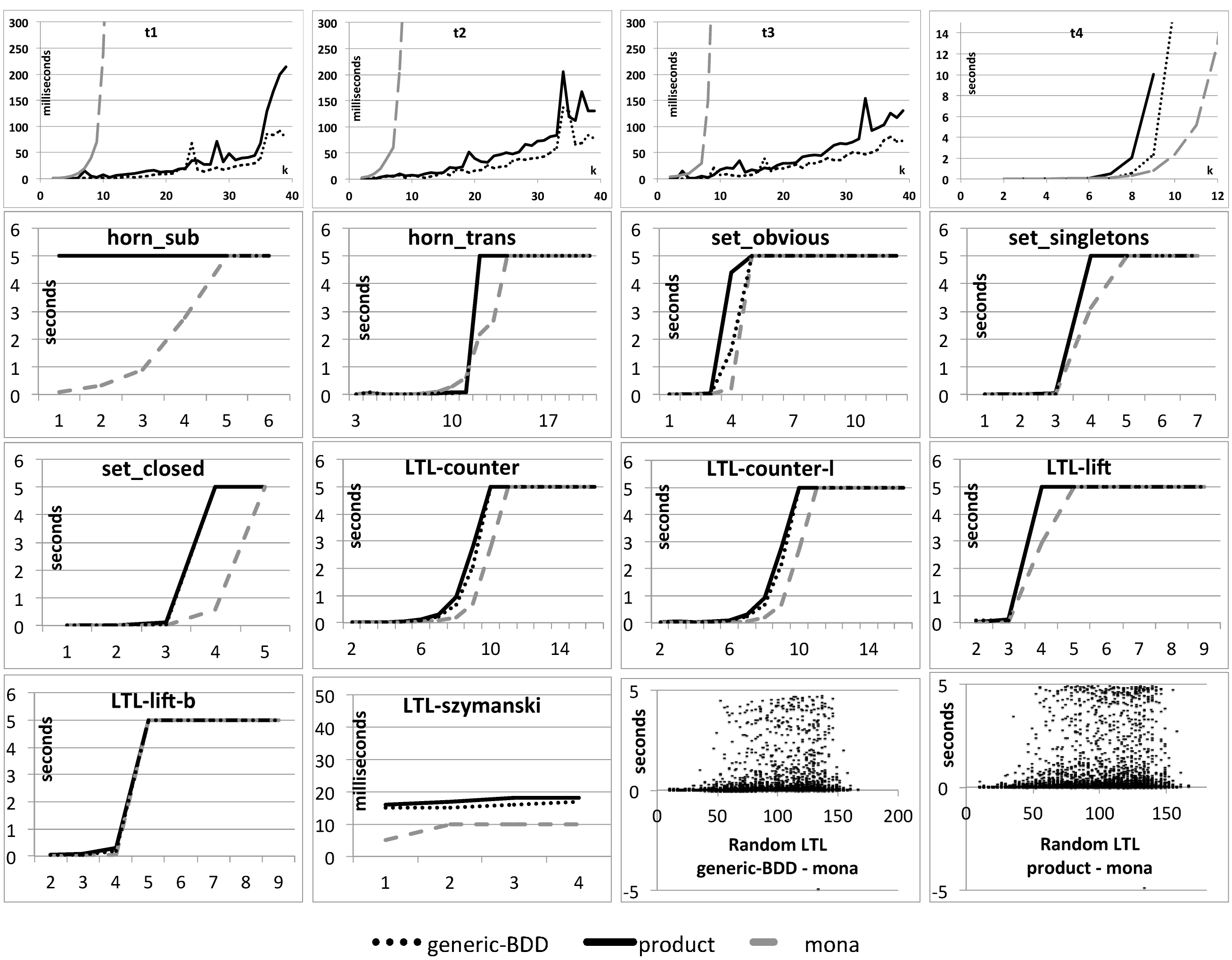


Figure 6. Comparison against Mona on the M2L-STR formulas from Table 2. The legend for the first 14 plots is at the bottom of the figure. Notice that for some plots the y-axis denotes seconds while for others it denotes milliseconds. The last two plots show the difference in runtime between our solver and Mona in the case of random LTL formulas. A point above 0 means that Mona was faster than our solver by the indicated number of seconds. In these two plots the x-axis denotes the size of the LTL formula. In the case of horn_trans, formulas with $k \geq 11$ caused our prototype parser to throw stack overflow exceptions due to the deep nesting of the formulas.

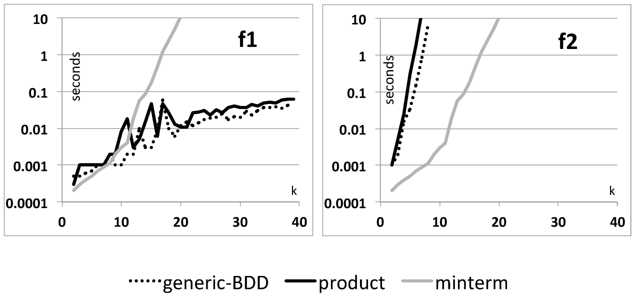


Figure 7. S-M2L-STR to S-FA running times for f1, f2, and minterm generation. The y-axes are in log-scale. The missing values are due to the algorithms running out of memory or timing out at 60s.

LTL formulas. In particular, the solver based on the product algebra (Section 6.3) is faster than Mona on 0.5% of all the instances while the one based on generic BDDs (Section 6.2) is faster than Mona on 4% of all the instances. The generic BDD algebra is generally faster than the product algebra and we observe this trend in approximately 90% of the instances. This experiment shows that our decision procedure for S-M2L-STR adds generality and support for different theories without sacrificing too much performance in the case of finite alphabets. Despite the many tuned optimizations that are implemented in Mona and not in our tool, our solver can handle most practical M2L-STR formulas appearing in verification applications.

8.2 Formulas with Large Sets of Minterms

In our introduction we explained how every S-M2L-STR formula can be transformed into an equisatisfiable M2L-STR one. However, to perform this reduction, one needs compute the set of all satisfiable Boolean combinations of all the unary predicates (minterms) appearing in the original S-M2L-STR formula. In the worst case, the set of minterms can have exponential size in the number of given predicates and this reduction can be impractical. In this experiment, we evaluate our decision procedure against the running time of computing the set of minterms.

We consider the following two classes of S-M2L-STR formulas over **BDD⟨B1⟩**, where the predicate $\beta_i(x)$ is true iff the i-th bit of a bit-vector x is 1, and we let k vary between 2 and 40.

f1 $\exists x_1, \ldots, x_k.\mathit{first}(x_1) \bigwedge_{i<k} \mathit{succ}(x_i, x_{i+1}) \bigwedge_{i \le k} \beta_i(x_i)$.

f2 $\exists x_1, \ldots, x_k.\ \bigwedge_{i \le k} \beta_i(x_i)$.

Intuitively, the formulas in the class f1 describe strings in which the symbol at the i-th position, for $1 \le i \le k$, satisfies the predicate β_i, while the formulas in the class f2 describe strings in which for every $1 \le i \le k$ there exists some position in the string for which the corresponding symbol satisfies the predicate β_i. Although these two classes of formulas are similar and contain the same set of unary predicates, the equivalent automata are drastically different. While the s-FAs for the formulas in the class f1 have $k+1$ states, the automata for the formula in the class f2 have 2^k states. Regardless of the decision procedure, solving the formulas in f2 has to be at least as hard as computing the set of minterms.

The running times are shown in Figure 7. We report the running time of the minterm computation but DO NOT measure the performance of solving the M2L-STR formula over the resulting minterm alphabet; the resulting M2L-STR formulas are exponentially larger than the original ones and impractical for any solver.

Results As expected, the minterm computation shows an exponential behaviour. For the formulas in the class f1, the S-M2L-STR algorithms perform linearly are exponentially faster than the minterm computation. This is because, on such formulas, our decision procedure does not need to explore all the possible minterms. As expected, for the formulas in the class f2, the S-M2L-STR algorithms perform exponentially and are slower than the minterm computation. Indeed, the minterm computation is performed by the algorithm whenever, upon removing a quantifier, the obtained s-FA needs to be determinized. Hence, the performance overhead. Similarly to what we observed in Section 8.1, using the generic BDD algebra is faster than using the product algebra on approximately 70% of the instances.

Even though some of the existing M2L-STR solver [25] could be tailored to handle the theory of bit-vectors, this example shows how computing the set of all minterms is in general impractical. Our experiments also highlight how, on certain types of formulas, the S-M2L-STR solver does not necessarily explore the set of all minterms and achieves exponential speedup when compared to the minterm computation.

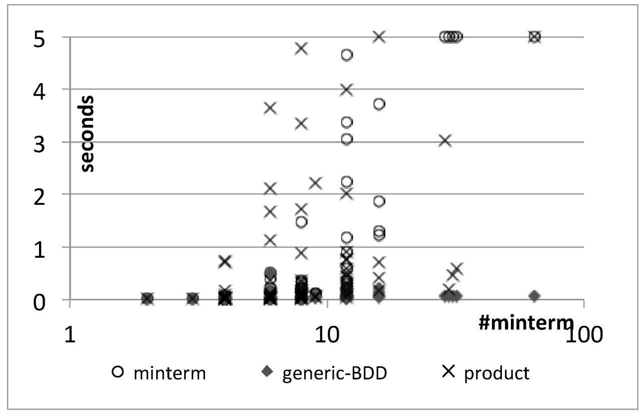


Figure 8. Minterm generation vs S-M2L-STR solving time for randomly generated formulas over the theory of linear integer arithmetic. The x-axis indicates the number of minterms in the formula. Timeout at 5 seconds.

8.3 Formulas Over the Theory of Linear Integer Arithmetic

The examples in Section 8.2 are somewhat pathological and should not appear too often in practice. In particular, it is rarely the case that all the Boolean combinations of the predicates appearing in the input formulas are indeed satisfiable and the set of minterms is impractically large. In this experiment we show how the number of minterms is not the only limiting factor in reducing S-M2L-STR formulas to M2L-STR ones. We show that, when considering formulas over complex alphabet theories for which it is expensive to check satisfiability, the minterm computation can be expensive even when the number of minterms is not prohibitively large.

We randomly generate 75 S-M2L-STR formulas over the unary theory of linear integer arithmetic satisfying the following requirements.

- Unary predicates are of the form $ax \mathit{ mod } b = c$ with $a, b, c \in \{-3, -2, -1, 0, 1, 2, 3\}$ and are satisfiable.
- Formula have size smaller than 15.
- Formulas contain at least 3 unary predicates.

We used the SMT solver Z3 [19] to represent the algebra of the alphabet theory. We compare the time taken by our S-M2L-STR solver against the time to compute the set of minterms. The results are shown in Figure 8.

Results The minterm computation is slower than the S-M2L-STR algorithm which uses the generic BDD algebra on approximately 75% of the instances, slower than the algorithm which uses the product algebra on 55% of the instances, and already times out (5 seconds) or runs out of memory for instances with fewer than 30 minterms. These results are due to the following two reasons: (1) the S-M2L-STR solver rarely needs to compute all the satisfiable Boolean combinations of the given predicates, (2) each minterm is a Boolean combination of all the unary predicates and therefore a potentially large formula.

In line with the former experiments, the generic BDD algebra is faster than the product algebra on 86% of the instances. In this experiment the difference is more noticeable and the solver based on the product algebra is on average 10 times slower than the one based on generic BDDs.

This experiment clearly illustrates the need for a specialized solver for S-M2L-STR. While practical for very simple theories, in the presence of more complex alphabet theories such as linear integer arithmetic, the minterm generation procedure can be prohibitive already for very small formulas.

9. Related Work

Logic M2L-STR. Every regular string language can be expressed as an M2L-STR formula and vice versa. This relation was first discovered by Büchi [9]. Since then, similar results have been proven for tree languages, infinite string languages, and transductions [14, 40]. M2L-STR has non-elementary complexity and the tightness of this bound was proven in [35]. Our symbolic extension, S-M2L-STR is strictly more expressive than M2L-STR and retains decidable satisfiability.

Solvers for M2L-STR. The first practical solver for M2L-STR was Mona [25], which used multi-terminal BDDs to efficiently encode the automata corresponding to M2L-STR. Recently, novel approaches to solving M2L-STR using anti-chain [22] and coalgebras [41] have been proposed. The benefits of these algorithms have only been demonstrated on restricted classes of formulas. However, applying the recent advances in M2L-STR solving to improve our decision procedures S-M2L-STR is a promising direction.

In general our solver has comparable performance to such tools when operating over finite domains, but it also directly supports arbitrary theories. This aspect allows S-M2L-STR to be easily integrated with existing SMT solvers as we did in our tool with Z3.

Decision diagrams. Binary decision diagrams [2, 3, 32] have been used as efficient and succinct data-structures for over 40 years. BDDs, or more precisely, Reduced Ordered BDDs, are a canonical data structure for Boolean functions that was introduced in [8] and have had an immense success in many areas of computer science, such as model checking of both hardware [13] and software [34]. Extended BDDs or Multi-Terminal BDDs or MTBDDs, also known as Algebraic Decision Diagrams or ADDs, allow multiple terminals with associated terminal algebras [6, 11, 12, 23]. Here we specialize ADDs to have effective *Boolean* algebras as their terminals and call this model *generic BDDs*. Our use of ITE expressions (or DAGs) for representing predicates in Cartesian products of Boolean algebras explores another form of Shannon expansions, that unlike generic BDDs, are not canonical but depend ultimately on the representation of predicates in the given algebras. ITE DAGs are standard data structures in SMT tools [20]. ITE DAGs have also been studied as an alternative to BDDs in [28]. We do not know of prior work that has studied Shannon expansions for representing Cartesian products of Boolean algebras.

Applications. Thanks to the advances in solving practical instances, M2L-STR has found application in many domains such as hardware verification [7], personalized education [4], and pointer analysis [27]. In particular, the last application could greatly benefit from the symbolic logic S-M2L-STR. When reasoning about linked data-structures such a lists, the content of the nodes is abstracted to enable decidable analysis (e.g., the node is nil or not nil). In previous approaches, these abstractions have been separated from the logic that reasons about the list structure (e.g., M2L-STR). S-M2L-STR separates the alphabet theory from the sequence predicates and provides an elegant way to jointly reason about the list structure and its content.

Symbolic automata. The concept of automata with predicates instead of concrete symbols was first mentioned in [43], then in [31], and further formalized in [16]. Symbolic automata provide an elegant framework for separating the structure of the alphabet from the structure of the automaton. This separation of concerns has proven to be beneficial in many applications and checking satisfiability of S-M2L-STR is yet another one. It is important to stress out that, thanks to symbolic automata, implementing the algorithms described in this paper only required us to design novel Boolean algebras rather than designing novel data-structures or efficient representations for the automata themselves as it was done in [25].

10. Conclusions

We introduced the monadic second-order logic of one successor on finite sequence (S-M2L-STR) for describing sets of sequences drawn from arbitrary domains. S-M2L-STR extends M2L-STR by allowing character predicates to range over a decidable background theory instead of a finite domain. We presented a decision procedure for S-M2L-STR that reduces a formula to a symbolic finite automaton operating over an alphabet consisting of pairs of symbols. The first element of the pair is a symbol in the original formula's alphabet, while the second element is a bit-vector. We then propose two implementations of the Boolean algebras that are necessary to efficiently manipulate predicates of the alphabet of pairs. Our preliminary implementation of the S-M2L-STR decision procedure is integrated with the SMT solver Z3 and can therefore support arbitrary SMT theories as alphabet theories. Despite this generality, our implementation has comparable performance with the state-of-the-art M2L-STR solver Mona.

Acknowledgements We thank Nikolaj Bjørner, who suggested the order $\prec$ that led to the predicate trie data structure.

References

[1] P. Abdulla, J. Deneux, L. Kaati, and M. Nilsson. Minimization of non-deterministic automata with large alphabets. In *Implementation and Application of Automata*, volume 3845 of *LNCS*, pages 31–42. Springer, 2006.

[2] S. B. Akers. On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics*, 7(4):487–498, December 1959.

[3] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.

[4] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of DFA constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1976–1982. AAAI Press, 2013.

[5] Automata. `https://github.com/AutomataDotNet/Automata`, 2015.

[6] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in Systems Design*, 10(2/3):171–206, 1997.

[7] D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *Formal Methods In System Design*, 13:255–288, 1998. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[9] J. Buchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik und Grundl. Math.*, 6:66–92, 1960.

[10] C. C. Chang and H. J. Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundation of Mathematics*. North Holland, third edition, 1990.

[11] E. Clarke, M. Fujita, P. McGeer, K. McMillan, and J. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *IWLS93: International Workshop on Logic Synthesis*, pages 6a:1–15, Lake Tahoe, CA, May 1993.

[12] E. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation, 1993. 30th Conference on*, pages 54–60, June 1993.

[13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[14] B. Courcelle. Monadic second-order definable graph transductions: a survey. *Theoretical Computer Science*, 126(1):53 – 75, 1994.

[15] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 249–260, London, UK, UK, 1999. Springer-Verlag.

[16] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *POPL'14*. ACM, 2014.

[17] L. D'antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. volume 38, pages 1:1–1:32, New York, NY, USA, Oct. 2015. ACM.

[18] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 854–860. AAAI Press, 2013.

[19] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.

[20] L. de Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Comm. ACM*, 54(9):69–77, 2011.

[21] M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. *TACAS 2008*, chapter Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[22] T. Fiedor, L. Holík, O. Lengál, and T. Vojnar. Nested antichains for WS1S. In *TACAS 2015*, pages 658–674, 2015.

[23] M. Fujita, P. McGeer, and J.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997.

[24] A. Harding. Symbolic strategy synthesis for games with LTL winning conditions. Technical report, 2005.

[25] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*. Springer, 1995.

[26] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *USENIX Security*, August 2011.

[27] J. L. Jensen, M. E. Joergensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *PLDI '97*, 1997.

[28] K. Karplus. Using if-then-else DAGs for multi-level logic minimization. In *Proceedings of the Decennial Caltech Conference on VLSI on Advanced Research in VLSI*, pages 101–117. MIT Press, 1989.

[29] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, January 2001.

[30] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[31] D. Kozen. Automata on guarded strings and applications. *Matématica Contemporânea*, 24:117–139, 2003.

[32] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[33] P. Madhusudan and X. Qiu. *Efficient Decision Procedures for Heaps Using STRAND*, pages 43–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[34] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[35] A. R. Meyer. Weak monadic second order theory of successor is not elementary-recursive. Technical report, Cambridge, MA, USA, 1973.

[36] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.

[37] B. Reed. The height of a random binary search tree. *Journal of the ACM*, 50(3):306–332, May 2003.

[38] K. Y. Rozier and M. Y. Vardi. *LTL Satisfiability Checking*, pages 149–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[39] R. Sebastiani, S. Tonetta, and M. Y. Vardi. Symbolic systems, explicit properties: on hybrid approaches for LTL symbolic model checking. *International Journal on Software Tools for Technology Transfer*, 13(4):319–335, 2011.

[40] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[41] D. Traytel. A coalgebraic decision procedure for WS1S. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, pages 487–503, 2015.

[42] M. Veanes. *Implementation and Application of Automata: 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*, chapter Applications of Symbolic Finite Automata, pages 16–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[43] B. W. Watson. Implementing and using finite automata toolkits. In *Extended finite state models of language*, pages 19–36, New York, NY, USA, 1999. Cambridge University Press.