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 $a$. 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, $\psi$, captures the property we informally described: $\forall x. [\varphi_{\text{odd}}(x) \rightarrow \exists y. x < y \land [\varphi_{>4}(y)]$. Here $\varphi_{\text{odd}}$ and $\varphi_{>4}$ are unary integer linear arithmetic predicates, and $\psi$ describes all sequences of integers such that if a position $x$ contains an odd element ($[\varphi_{\text{odd}}(x)]$), 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 $\varphi$ 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. For example, the set of equivalence classes of the S-M2L-STR formula $\psi$ is:

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

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 $\psi$ if and only if $l$ is a model of $\psi$. Using this argument, every S-M2L-STR formula $\psi$ can be compiled into an equi-satisfiable M2L-STR formula over the alphabet $M(\psi)$. Unfortunately, computing the set of formulas $M(\psi)$ 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].
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 $\varphi$, one can construct a finite automaton that accepts the same set of sequences accepted by $\varphi$ [9]. In this reduction, if the M2L-STR formula $\varphi$ has $k$ free variables and operates over a finite alphabet $\Sigma$, 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 $\varphi$, one can construct a symbolic finite automaton (s-FA) [16] that accepts the same set of sequences accepted by $\varphi$. 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 $\varphi$ over an (infinite) alphabet $\sigma$ and with $k$ free variables, we construct a symbolic automaton $A_\varphi$ over the alphabet $\sigma \times \{0, 1\}^k$. We do so by showing that if the alphabet theory used by $\varphi$ 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 $\varphi$ 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], 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 alphabet and whose terminals are predicates of another alphabet.

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 when (h mod 26) :: map_caesar t | (* Remove odd numbers *) |
let rec filter_even (x: int list) : int list =
match x with
[| [] -> [] |
| h :: t when (h mod 26) = 0 -> filter_even t |
(* Composition of the two functions *) |
let map_filt (x: int list) : int list =
filter_even (map_caesar x)
(* Composition of the functions *) |
let filt_map_filt (x: int list) : int list =
map_filt (filter_even 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_even removes all the odd elements from a list, while map_filt and filt_map_filt are different functional compositions of map_caesar and filter_even.

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_caesar, 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 automation-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 -> 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

$$\text{list_of_even } l \equiv l \in \mathcal{L} \forall p. [\forall x. x \mod 2 = 0](p)$$
For S which macros are used to define commonly occurring predicates.

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 A is a tuple (U, Ψ, [ ], ⊥, ∧, ∨, ∃, ¬) where U is a non-empty recursively enumerable set called the universe of A. Ψ is a recursively enumerable set of predicates closed under the Boolean connectives, ∨, ∧ : U × Ψ → Ψ, ¬ : Ψ → Ψ, and ∃, ⊥ ∈ Ψ. The denotation function [ ] : Ψ → 2U is defined as is such that, [ ] = θ, [⊥] = U, for all ϕ ∈ Ψ, [ϕ] = {ψ ∈ [ ] | ϕ}, and [¬ϕ] = U \ [ϕ].

For ϕ ∈ Ψ, we write Sat(ϕ) when ϕ is satisfiable. The algebra A is decidable if 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. B1 = (({θ}, {0, 1}, {0} → 0, 1 → {θ}), {0, 1}, ∨, ∧, ¬) is the simplest possible effective Boolean algebra. The connectives implement the standard truth tables. STM 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, STMτ = (U, Ψ, [ ], ⊥, ∧, ∨, ∃, ¬, τ) 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 : τ, τ is x = x, ⊥ is x ≠ x, and the Boolean operations are the corresponding connectives in STM formulas. The interpretation function [ ] is defined using the operations of satisfiability checking and model generation provided by an SMT solver.

General notations. For a sequence s = (e1, ..., en) and for i ∈ [1, n], let s(i) = ei, and s · e = (e1, ..., en, e). The empty sequence is {}. For a non-empty set S of positive integers we define ε(S) to be any element of S and ε(∅) ≠ 0. In practice, we only use ε(S) when |S| = 1. For a term or formula ϕ we let FV(ϕ) 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 A. We call our variant generic BDDs and we let the leaves belong to ΨA where A is a Boolean algebra instead of an arbitrary algebra. Generic BDDs differ from BDDs where leaves can only belong to ΨB. The terminals are elements that denote subsets of U_A. Since the number of bits in the domain of generic BDDs is unrestricted they denote sets of functions from N to {0}∪{1} that we view as relations over U_A × N.

Formally, a generic BDD algebra over a given Boolean leaf alphabet A is a tuple BDD(A) = (U_A × N, Ψ, [ ], ⊥, ⊤, ∨, ∧, ∃, ¬) where Ψ is the set of natural numbers. The set of predicates Ψ is defined as the set {ψ(k) | k ≥ 0}, where ψ(k) is the least set that satisfies the following conditions:

1. For all ψ ∈ Ψ_A, map ψ to a unique element leaf(ψ) such that, for every ϕ ∈ Ψ_A, if [ψ]A = [ϕ]A then leaf(ϕ) = leaf(ψ).
2. If ψ ∈ Ψ_A, then leaf(ψ) ∈ Ψ(0).
3. If k > 0, 0 ≤ i, j < k, ψ ∈ Ψ(i), ϕ ∈ Ψ(j), and ψ ≠ ϕ, then node(ψ, ϕ) ∈ Ψ(k).

A predicate leaf(ϕ) is called a leaf or a terminal. We discuss in Section 4 how leaf(ϕ) can be implemented using a predicate trie. Accordingly, a predicate node(κ, ipv) is a non-leaf 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 ≥ 0, k ≥ 1, we define Bit(k(n)) = (n ÷ 2n−1) mod 2 = 1, where ÷ 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(ψ) = [ψ]A × N. The denotation of a non-terminal node(κ, ipv) is defined as

leaf(ψ) = leaf(¬ψ), leaf(ψ) ∧ leaf(ϕ) = leaf(ψ ∨ ϕ)
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 ψ[k] to distinguish ψ from ϕ
        if ψ[k] ∈ ϕ
          t.t₁ = new Tree(k + 1, ψ)
          if ψ[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 ψ[k] ∈ ϕ
              t.t₁ = new Tree(k + 1, ϕ)
              return ϕ
            else
              t.t₀ = new Tree(k + 1, ψ)
              t.t₁ = new Tree(k + 1, ϕ)
              return ϕ
            else
              t.t₀ = new Tree(k + 1, ψ)
              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)
            else
              t.t₁ = new Tree(k + 1, ϕ)
            return v
          else // proceed recursively on internal nodes
            if ψ[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 $Ψ_A$ so that new terminals of predicates in $Ψ_BDD(Δ_A)$ 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 $A$ is atomic. $A$ is atomic if it is possible to create predicates in $Ψ_A$ that denote singleton sets. An atomic effective Boolean algebra $A$ has the additional component

$$\text{Atom} : Ψ_A \rightarrow Ψ_A$$

such that $[\text{Atom}(ϕ)] \subseteq [ϕ]$, and if $\text{Sat}(ϕ)$ then $|[\text{Atom}(ϕ)]| = 1$. For example, $\text{SMT}_ι$ is atomic but $\text{BDD(B1)}$ is not atomic.

Atoms allow us to efficiently check satisfiability of formulas of the form $α ∧ ψ$ by treating atoms $α$ as concrete values. For example, given a predicate $λx.ψ(x)$ and an atom $α$ with denotation $[α]$, we can simply evaluate the formula $ψ(α)$ instead of performing a general satisfiability check. In the following, we write $α ∈ ψ$ for $α ∈ [ψ]$ 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 $v = (v_i)_{i=1}^n$ where each element $v_i$ is an atom and a finite set $S ⊆ Ψ_A$ of pairwise inequivalent predicates, for every two predicates $ψ, ϕ ∈ S$ there is some $v_i$ such that $v_i ∈ ψ ⇔ v_i ∉ ϕ$. To efficiently maintain this invariant we order the predicates in $S$ with respect to the following order.\(^6\)

$$ψ \prec_ω ϕ \iff ∃ i \in [1; |v|](v_i ∈ ϕ \land \forall j < i(v_j ∈ ϕ \Rightarrow v_j ∉ ϕ))$$

Intuitively, $ψ \prec_ω ϕ$, or $ϕ \prec_ω ϕ$ when $v$ is clear, means that if $v_i$ is the first element of the sequence $v$ that distinguishes $ψ$ from $ϕ$ then $v_i$ occurs in $ϕ$. It follows in particular that $⊥ \prec_ω ϕ$ and $ϕ \prec_ω ⊤$ for all $ϕ$ other than $⊥$ and $⊤$. Using the order $\prec_ω$, we can perform a binary search for efficiently inserting new predicates into $S$, and if need be, extend $v$ in the process. A key property of our algorithm is that, when the sequence $v$ does not suffice to distinguish a new predicate from the ones already in $S$, appending a new atom at the end of $v$ does not affect the relative ordering among the predicates already in $S$.

**Proposition 1.** If $ψ \prec_ω ϕ$ and $α ∉ ψ$ then $ψ \prec_ω α \lor ϕ$.

We design a predicate trie data structure for representing $S$ in $\prec_ω$-order and illustrate its operations in Figure 2. A concrete predicate trie $\text{PredicateTrie}(v, ψ)$ contains two attributes: the sequence $v$ of atoms, and a binary tree $t$ of depth $|v|$. The leaves of $t$ are the elements of $S$ and the internal nodes of $t$ have two subtrees, called the 1-subtree and the 0-subtree, respectively. The 1-subtree $t_1$ and the 0-subtree $t_0$ are such that, for all leaves $ψ$ in $t_0$ and $ψ$ in $t_1$, we have that $ψ_0 \prec ψ_1$. The linear order $\prec_ω$ determined by $v$ 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 $m ≤ |v|$ and $b_i = 1$ if $v(i) ∈ ψ_1$, and $b_i = 0$ if $v(i) ∉ ψ_1$. For example, a branch $(1, 0, 0)$ to $ψ$ represents the valuation $v_1 ∈ ψ$, $v_2 ∉ ψ$, and $v_3 ∉ ψ$. The algorithm for searching a predicate $ϕ$ in a trie is given in Figure 2 as the method $\text{SEARCH}$.

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

---

\(^6\) Formula $ϕ \backslash ψ$ stands for $ϕ ∧ ¬ψ$. 
leaves that are ⊥ and ⊤, respectively. Trivially α ⊏ ⊥ and α ∈ ⊤. This ensures that valid predicates will always be mapped to ⊤ and unsatisfiable predicates to ⊥.

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 v (k ≤ |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 v(k) distinguishes them and the leaf is expanded to a subtree incorporating both ψ and ϕ. In the case k > |v| (lines 36–48), the sequence 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 ϕ 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 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 ϕ(k) ∈ ϕ 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 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 ∈ [2] 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 U₄ is the interval [0; K) for some finite K and that each predicate ϕ is a number in U₄ = [0; 2ₖ) representing the set of all b ∈ U₄ such that bit b of ϕ is one, i.e., ϕ ≡ 2ᵣ mod 2 = 1. An atom 2ᵣ represents the set {b}.

When constructing a trie from a sequence {ϕᵢ}ⁿᵢ=1 of predicates, we assume that all predicates have been chosen uniformly at random from U₄, so all n! permutations are equally probable. The test Sat(ϕ ⊨ ψ) is the test ϕ ◊ ψ. The function Atom(ϕ) is defined as 0 if ϕ = 0, else 2ᵣ where b is the least bit such that 2ᵣ ∈ ϕ.

Consider a trie that has been constructed from {ϕᵢ}ⁿᵢ=1. The trie orders the predicates according to ⊑. One can systematically map the trie into a ⊑-ordered BST, say BST₁, such that for any subtree t with root r, for all nodes x in the left subtree of t we have x ⊏ r and for all nodes x in the right subtree of t we have r ⊏ x. BST₁ 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 {ϕᵢ}ⁿᵢ=1 of the original input sequence {ϕᵢ}ⁿᵢ=1 such that inserting the elements ϕ₁, ϕ₂, . . . with respect to ⊑ order produces the same BST₁. The expected height of a random BST constructed from a random equiprobable permutation of n values is O(log n), cf. [37].

Example 2. Consider A as in Section 4.1 with K = 128. One can think of U₄ 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:

\[ \{d\} = \{48; 57\} \]
\[ \{w\} = \{48; 57\} \cup \{65; 90\} \cup \{97; 122\} \]
\[ \{s\} = \{9; 13\} \]

Moreover, let \D = ¬\d, \W = ¬\w, and \S = ¬\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 ∈ [1; |v|] have labels a such that v(i) = 2ᵃ. The root has label 0 because Atom(⊤) = 2⁰.

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 Α, or M2L-STR(Α), is defined by the grammar

ϕ := ¬ϕ | ϕ ∧ ϕ | ∃x(ϕ) | ∃X(ϕ) | x < y | X(x) | [α](x)

where α ∈ Α. 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., ψ ⇒ ϕ is defined as ¬ψ ∨ ϕ. Table 1 shows a set of complex formulas that can be derived from the basic syntax.

\footnote{Observe that the standard order ⪯ is the special case ⪯(2⁰, 2¹, ..., 2ᴷ⁻¹).}
The operator that differentiates \(S\text{-}M2L\text{-}STR\) from \(M2L\text{-}STR\) is the unary predicate \([\alpha](x)\). In \(M2L\text{-}STR\) predicates are not really used or even needed. Instead, if there is a finite signature \(\Sigma\) 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 \(S\text{-}M2L\text{-}STR\) the predicate \(\alpha\) can be an arbitrary predicate of the Boolean algebra \(A\).

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

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

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

**Example 4.** For example, the character \(r = 6\) satisfies \(\psi(r)\) above but also even\((r)\). Moreover, since for every two predicates \(\psi_1, \psi_2 \in \Psi_A\) the predicate \(\psi_1 \land_A \psi_2\) is in \(\Psi_A\), it follows that if \(\psi_1\) and \(\psi_2\) are unary predicates in \(S\text{-}M2L\text{-}STR\) then so is \(\psi_1 \land_A \psi_2\).

**Semantics** We formally define the semantics of \(S\text{-}M2L\text{-}STR\). Intuitively, \(w\) are elements of \(U_A\), i.e., the Kleene closure or the set of all finite sequences over \(U_A\). First-order variables in the formulas refer to positions of individual letters of words (elements of \(U_A\)) and second-order variables refer to finite sets of positions. Predicates \(\alpha\) 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 \(\phi\) be a \(S\text{-}M2L\text{-}STR\) formula such that \(FV(\phi) \subseteq \{x_j\}_{j \geq 1} \cup \{X_j\}_{j \geq 1}\) and, for any \(j\), at most one of \(x_j\) or \(X_j\) is free in \(\phi\), otherwise \(\phi\) is ill-formed.

Let \(J = \{j \mid x_j \in FV(\phi)\}\) be the set of all free variable indices in \(\phi\). Let \(w \in U_A\) and let \(\theta\) be a finite sequence of subsets of positions \(w\) (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 \(\phi\) then \(\theta(j)\) must be a singleton set. We define the semantics using judgements of the form \(w, \theta \vDash \phi\):

- \(w, \theta \vDash \neg \phi\) \iff \(w, \theta \not\vDash \phi\)
- \(w, \theta \vDash \phi_1 \land \phi_2\) \iff \(w, \theta \not\vDash \phi_1\) and \(w, \theta \not\vDash \phi_2\)
- \(w, \theta \vDash \exists x\phi(x)\) \iff there exists \(i \in \{1; |w|\}\) such that \(w, \theta \vDash \phi(x_\theta(i+1))\)
- \(w, \theta \vDash \phi(X(\theta))(X)\) \iff there exists \(i \subseteq \{1; |w|\}\) such that \(w, \theta \not\vDash \phi(X(\theta_\theta(i+1)))\)

A formula is closed if it has no free variables and it is open otherwise. Let \(\phi\) be a closed formula. We write \(w \vDash \phi\) for \(w, () \vDash \phi\). The language of \(\phi\) is the subset of \(U_A^*\),

\[\mathcal{L}(\phi) \equiv \{w \in U_A^* \mid w \vDash \phi\}.
\]

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

- \(OddEven(X, Y) \equiv X \cup Y = \emptyset \land \forall x \forall y ((y = x + 1 \land X(y)) \implies X) \land \forall x \forall y ((x = y + 1 \land X(x)) \implies Y))\)
- \(Odd(x) \equiv \exists Y (OddEven(X, Y) \land X(X))\)
- \(Even(x) \equiv \exists Y (OddEven(X, Y) \land Y(Y))\)

Consider \(A\) to be integer linear arithmetic with even as the predicate \(\lambda x.\mod x 2 = 0\) and odd as the predicate \(\lambda x.\mod x 2 = 1\). Let \(\phi\) be the following closed \(S\text{-}M2L\text{-}STR\) formula

\[\forall x ((\text{Odd}(x) \implies \text{Even}(x)) \land (\text{Even}(x) \implies \text{Odd}(x))).\]

The language of \(\phi\), \(\mathcal{L}(\phi)\), consist of all finite sequences \(w\) of integers such that for all \(i, 1 \leq i \leq |w|, w(i)\) is odd if \(i\) is even and \(w(i)\) is even if \(i\) is odd.

### 5.1 S-M2L-STR vs M2L-STR

The classic decision procedure for \(M2L\text{-}STR\) 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(⟨A⟩) 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 \( \land \) and let \( \neg \) denote \( \neg \). The formula \( [\alpha](x_i) \land [\beta](x_i) \) is equivalent to \( [\alpha \land \beta](x_i) \) because, using \( \theta(i) = 1 \),

\[
\begin{align*}
\forall \, x, \theta : \mathcal{A} \land [\beta](x_i) & \iff w(\theta(i)) \in \mathcal{A} \land [\beta]_A \\
& \iff w(\theta(i)) \in [\alpha \land \beta]_A \\
& \iff w(\theta(i)) \in [\neg \alpha](x_i).
\end{align*}
\]

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

\[
\begin{align*}
\forall \, x, \theta : \mathcal{A} \land \neg \alpha(x_i) & \iff w(\theta(i)) \notin [\alpha](x_i) \\
& \iff w(\theta(i)) \notin \mathcal{A} \land [\alpha]_A \\
& \iff w(\theta(i)) \notin [\neg \alpha]_A \iff w(\theta(i)) \notin [\neg \alpha](x_i).
\end{align*}
\]

Such transformations may be beneficial when the ability to push the negation into the alphabet theory can imply further simplifications. For example, if \( \alpha \) has the form \( \neg \beta \) then \( \neg \alpha \) 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 \( \Sigma \) 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 \( c = (a, (b_j)_{j \in I}) \in \Gamma \) in position \( i \) of a word \( w \) encodes the property: \( i \in X_j \iff b_j = 1 \), meaning that position \( i \) is an element of the set \( X_j \). Thus, every string \( w = (c_i)_{i \in I} \) accepted by \( M \), where \( c_i = (a_i, (b_{i,j})_{j \in I}) \) has the following property: if for each variable \( X_j \), we let \( I_j = \{i \in [1..n] \mid b_{i,j} = 1\} \), then

\[
(a_i)_{i \in I} \land (b_j)_{j \in I} = \varphi(X_1, \ldots, X_k).
\]

Our decision procedure uses the same idea and transforms a M2L-STR(⟨A⟩) formula into a symbolic finite automaton, which is a finite automaton in which edge labels are replaced by predicates. Given a M2L-STR(⟨A⟩) our algorithm constructs a symbolic finite automaton over the alphabet \( U_A \times \mathbb{N} \). 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 \( U_A \times \mathbb{N} \) we consider two approaches. First we show how we can use BDD(⟨A⟩). We then discuss an alternative way to construct a product algebra \( A \otimes \text{BDD(⟨B1⟩)} \) 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 \( (A, Q, q^0, F, \Delta) \) where \( A \) 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 \( \Delta \subseteq Q \times \Psi_A \times Q \) is a finite set of transitions. A transition \( (p, \varphi, q) \in \Delta \) is also denoted by \( p \xrightarrow{\varphi} M q \) (or \( p \xrightarrow{\varphi} q \)).

A word \( w = (a_i)_{i \in I} \in U_A \) is accepted from state \( p \in M \), denoted \( w \in \mathcal{L}_p(M) \), if either \( w = \epsilon \) and \( p \in F \), or there exist \( p_{i-1} \xrightarrow{\varphi} M p_i \), \( a_i \in \mathcal{F} \), for \( 1 \leq i \leq k \), such that \( p_0 = p \), and \( p_k \in F \). The language of \( M \) is \( \mathcal{L}(M) = \bigcup_{p \in \mathcal{F}} \mathcal{L}_p(M) \).

s-FAs are closed under Boolean operations [42] and we use \( \lor \), \( \land \), and \( \neg \) 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(⟨A⟩) to s-FAs over \( \mathcal{BDD}(A) = (U_A \times \mathbb{N}, \Psi, [], \lor, \land, \top, \bot, \neg) \). Let

\[
\beta_i \equiv \text{node}(i, \top, \bot) \quad P_\alpha \equiv \text{leaf}(\alpha).
\]

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

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 \( U_A \times \mathbb{N} \) such that for every character \((a, n) \) either \( a \) satisfies \( \alpha \) 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 \( \mathcal{BDD}(A) \) and we use it for correctly modelling existential quantifier in our translation. The transformation behaves as follows: for \( k \geq 1 \), 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,
we define the restriction for predicates in $\Psi_{\text{nBD}}(A)$,

\[
\text{leaf}(\alpha)|0 \overset{\text{def}}{=} \alpha, \quad \text{leaf}(\alpha)|k \overset{\text{def}}{=} \text{leaf}(\alpha) \quad (k > 0),
\]

\[
\text{node}(i, \varphi, \psi)|k \overset{\text{def}}{=} \begin{cases} 
\text{node}(i, \varphi, \psi), & \text{if } i < k; \\
\varphi \lor \psi, & \text{if } i = k; \\
\varphi \lor \psi|k, & \text{otherwise}.
\end{cases}
\]

Observe that, for $k = 0$ the restriction operation produces a predicate in $\Psi_A$. For $0$ and $k \geq 1$, the restriction operation is lifted to all $M \in \Psi_M(\text{nBD}(A))$

\[
M|0 \overset{\text{def}}{=} (A, Q_M, q_0^M, F_M, \{(p, \varphi|0, q) \mid (p, \varphi, q) \in \Delta_M\}), \quad M|k \overset{\text{def}}{=} (\text{BDD}(A), Q_M, q_0^M, F_M, \{(p, \varphi, q) \mid (p, \varphi, q) \in \Delta_M\}).
\]

We can now define the translation of the formulas as follows, where the atomic formulas use the s-FA from Figure 3. We only consider normalized formulas, where a formula $\varphi$ is normalized if:

1) all variables in $\varphi$ belong to $\{x_i \mid i \geq 1 \cup \{X_1\}_{i \geq 1}\};$
2) there is no $i$ and subformula $\psi$ of $\varphi$ such that $x_i, X_i \in \text{FV}(\psi);$ 3) for all subformulas $\exists \psi, \psi$ we have $i = \max\{j \mid x_j \in \text{FV}(\psi)\}.\]

\[
sFA(x_i = x_j) \overset{\text{def}}{=} \text{Equal}_{i,j},
\]

\[
sFA(\varphi_1 \land \varphi_2) \overset{\text{def}}{=} sFA(\varphi_1) \otimes sFA(\varphi_2),
\]

\[
sFA(X_i(x)) \overset{\text{def}}{=} \text{Member}_{i},
\]

\[
sFA(\neg \varphi) \overset{\text{def}}{=} sFA(\varphi)^{\otimes} \otimes \exists x_i \in \text{FV}(\varphi) \text{ Single}_{i},
\]

\[
sFA(\exists x_i \varphi) \overset{\text{def}}{=} \text{bd},
\]

\[
sFA(\exists x_i \psi) \overset{\text{def}}{=} \text{Less}_{i,j},
\]

\[
sFA(x_i < x_j) \overset{\text{def}}{=} \text{Less}_{i,j},
\]

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

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

\[
sFA(\neg \exists x_i (\neg (X_i(x) \land X_j(x)) \lor (\neg X_i(x) \land \neg X_j(x)))).
\]

\[
\text{Example 6.} \quad \text{Figure 4 compares } sFA(\neg \exists x_i (x_1)) \text{ to } sFA(\exists x_i (x_1))^0 \text{ to illustrate why the additional singleton restriction Single}_{i} \text{ is needed: } sFA(\exists x_i (x_1))^0 \text{ treats } x_1 \text{ as it was second-order.}
\]

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

\[
\begin{align*}
\text{Normalization is a simple syntactic variable renaming procedure.}
\end{align*}
\]

s-FA. We first define the language of normalized open formulas $\varphi$. To this end, given a judgement $w, \theta \models \varphi$ we let $w^{(\theta)}$ denote the following subset of $((U_A \times \mathbb{N})^\ast, \theta = X = (X_j)_{j=1}^k$.

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

The intuition is that $w^{(\theta)}$ is a lifting of the word $w$ from the alphabet $U_A$ to the alphabet $U_A \times \mathbb{N}$ such that, for each position $i$ of $w$, letter $a_i = w(i)$ in position $i$ 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

\[
\mathcal{L}_{\text{open}}(\varphi) \overset{\text{def}}{=} \bigcup\{w^{(\theta)} \mid w, \theta \models \varphi\}.
\]

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 $\text{m2L-str}(A)$ formulas.

\textbf{Lemma 1.} \textit{For normalized } $\varphi$, $\mathcal{L}_{\text{open}}(\varphi) = \mathcal{L}(sFA(\varphi))$.

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

\textbf{Theorem 1.} \textit{For closed } $\varphi$, $\mathcal{L}(\varphi) = \mathcal{L}(sFA(\varphi)) \setminus \{0\}$.

\textbf{Nonempty set semantics} \textit{The s-FA 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-FA would be different, e.g., in Figure 3(q) 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-FA involving first-order variables would be affected similarly.\}

The overall translation is affected so that $\text{Single}_{i}$ is replaced by $\text{Nonempty}_{i}$, 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 $\text{Nonempty}_{i}$ 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].}

\textbf{6.3} \textit{From s-m2L-str to s-FA using a Product Algebra} \textit{Given two effective Boolean algebras $A$ and $B$, a Cartesian product for $(A, B)$ is any effective Boolean algebra $C$, with }$U_C = U_A \times U_B$, \textit{that is associated with effective predicate transformers $D$ (decompose) and $C$ (compose) such that for all }$\psi \in \Psi_C$ the sum of products decomposition $D(\psi)$ is a finite subset of $\Psi_A \times \Psi_B$ and conversely, for every finite $F \subseteq \Psi_A \times \Psi_B$ we have $C(F) \in \Psi_C$ where

\[
[\psi]_C = \bigcup\{[\alpha]_A \times [\beta]_B \mid (\alpha, \beta) \in D(\psi)\} = [IC(D(\psi))].
\]

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

An abstract (and highly impractical) definition of a Cartesian product for $(A, B)$, denoted $A \times B$, can be given as follows: $U_{A \times B} = U_A \times U_B$, $\Psi_{A \times B}$ contains all finite nonempty subsets of $\Psi_A \times \Psi_B$ representing sums of products or DNFs
\[ \varphi \land \psi \overset{\text{def}}{=} \text{MERGE}(\varphi, \psi, \top A) \]

\[
\text{MERGE}(\varphi : \Psi_{AB}, \psi : \Psi_{AB}, \pi : \Psi_A) : \Psi_{AB}
\]

1. match \( \varphi \)
2. case \( [\alpha, \varphi_1, \varphi_2] \)
3. let \( \psi_1 = \text{MERGE}(\varphi_1, \psi, \pi \land A \alpha) \)
4. let \( \psi_2 = \text{MERGE}(\varphi_2, \psi, \pi \land A \neg A \alpha) \)
5. return \( [\alpha, \psi_1, \psi_2] \)
6. case \( [\beta] \)
7. match \( \psi \)
8. case \( [\alpha, \psi_1, \psi_2] \)
9. let \( \pi_1 = \pi \land A \alpha \)
10. let \( \pi_2 = \pi \land A \neg A \alpha \)
11. if not \( \text{Sat}_A(\pi_1) \)
12. return \( \text{MERGE}(\varphi, \psi_1, \pi) \)
13. else if not \( \text{Sat}_A(\pi_2) \)
14. return \( \text{MERGE}(\varphi, \psi_2, \pi) \)
15. else
16. let \( \phi_1 = \text{MERGE}(\varphi, \psi, \pi_1) \)
17. let \( \phi_2 = \text{MERGE}(\varphi, \psi, \pi_2) \)
18. return \( [\alpha, \phi_1, \phi_2] \)
19. case \( [\gamma] \)
20. if \( \text{Sat}_B(\beta \land B \gamma) \)
21. return \( [\beta \land B \gamma] \)
22. else
23. return \( [\bot B] \)

Figure 5. Conjunction \( \varphi \land \psi \) in \( A \boxtimes B \).

(Disjunctive normal forms), and for \( \psi \in \Psi_{AB} \), \( [\psi]_{AB} \) is the definition of \( A \land B \). The definition of \( A \land B \) follows the well-formedness (used later) that the first component is always satisfiable: \( A \land B \) is the union of the two sets, \( A \land B \) applies De Morgan's laws and computes the DNF, and \( A \land B \) is defined in terms of \( A \land B \) and \( A \land B \). The following proposition is immediate.

**Proposition 2.** If \( A \) and \( B \) are effective Boolean algebras then so is \( A \times B \).

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

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

- a terminal \([\beta]\) with \( \beta \in \Psi_B \), or
- a nonterminal \([\alpha, \varphi_1, \varphi_2]\) where \( \alpha \in \Psi_A \) and \( \varphi_1, \varphi_2 \) are ITEs.

In other words, an ITE for \( (A, B) \) is a Shannon expansion whose terminal predicates belong to \( \Psi_B \) and whose nonterminal predicates belong to \( \Psi_A \). We have \( T_{AB} \overset{\text{def}}{=} \Psi_T \) and \( \bot_{AB} \overset{\text{def}}{=} \bot_B \).

Let \( \psi \in \Psi_{AB} \). We define the path condition to a node in \( \psi \) as a predicate in \( \Psi_A \) as follows: the path condition to the root of \( \psi \) is \( T_A \), and if the path condition to a node \([\alpha, \varphi_1, \varphi_2]\) in \( \pi \) is \( \pi \), then the path condition to \( \varphi_1 \) is \( \pi \land A \alpha \) and the path condition to \( \varphi_2 \) is \( \pi \land A \neg A \alpha \).

The pair \( (\pi, \beta) \) where \( \pi \) is the path condition to a terminal \([\beta]\) of \( \psi \) is called a branch of \( \psi \). Define \( D(\psi) \) as the set of all branches of \( \psi \). We say that \( \psi \) is reduced if for all \( (\pi, \beta) \in D(\psi) \):

- \( \text{Sat}_A(\pi) \),
- either \( \beta \in \{\bot_B, \top B\} \) or both \( \text{Sat}_B(\beta) \) and \( \text{Sat}_A(\neg A \alpha) \).

For example, the ITE \( [T_A, [\bot_B], [\top B]] \) is not reduced because, in the branch \( [\neg A T_A, \top B] \), the path condition \( \neg A T_A \) is unsatisfiable. Observe that both \( [\top B] \) and \( [\bot B] \) are trivially reduced.

The two key operations over ITEs, much like for BDDs, are negation and conjunction. Negation \( \neg = \neg_{AB} \) is defined as follows.

\[ \neg[\beta] \overset{\text{def}}{=} [\neg_B \beta], \quad [\alpha, t, f] \overset{\text{def}}{=} [\alpha, \neg t, \neg f] \]

Trivially, negated ITEs are also reduced, provided that \( \neg_B \bot_B \) is \( \bot_B \) and \( \neg_B \top_B \) is \( \top_B \). Conjunction is defined in Figure 5 and uses satisfiability checks in \( A \) and \( B \) to maintain the property that the output formula is reduced if the inputs are reduced.

The denotation function \( [\psi] \) for \( \psi \in \Psi_{AB} \) is defined in the obvious way, as well as the composition operator, so that \([\psi] = [C(D(\psi))]\).

**6.4 BDD(\(A\)) vs \(A \boxtimes BDD(B1)\).**

The main structural difference between \( BDD(A) \) and \( A \boxtimes BDD(B1) \) is the following: in \( BDD(A) \) the leaves are predicates from \( \Psi_A \) while the internal nodes are bit-branches, but in \( A \boxtimes BDD(B1) \) the internal nodes are predicates from \( \Psi_A \) and the leaves are BDDs. Essentially, the representations are turned upside-down (relatively to each other).10

For the effective Boolean algebra \( C = A \boxtimes BDD(B1) \) we define the restriction operation \( \langle \_ \rangle \) as follows. Let \( \psi \in \Psi_C \) and \( k \geq 1 \); \( \psi \) is either a leaf \([\beta]\) or a node \([\alpha, t, f] \).

\[ [\beta] \langle k \rangle \overset{\text{def}}{=} [\beta \langle k \rangle], \quad [\alpha, t, f] \langle k \rangle \overset{\text{def}}{=} [\alpha \langle k \rangle, t, f \langle k \rangle] \]

So in \( C \) we have that \( \psi \langle k \rangle \) restricts the positions of all the terminal predicates in \( \psi \) to \( k \). As the special case, for \( k = 0 \), \( \psi \langle 0 \rangle \) is defined as a predicate in \( A \):

\[ [\beta] \langle 0 \rangle \overset{\text{def}}{=} \begin{cases} T_A, & \text{if } \beta \langle 0 \rangle = 1; \\ \bot_A, & \text{otherwise}. \end{cases} \]

\[ [\alpha, t, f] \langle 0 \rangle \overset{\text{def}}{=} (\alpha \land_A t \langle 0 \rangle) \lor_A (\neg A \alpha \land_A f \langle 0 \rangle) , \]

where \( \beta \langle 0 \rangle \) is always either 0 or 1 (recall the definition of \( B1 \) from Section 3.1).

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

\[ \beta \overset{\text{def}}{=} [\text{node}(i, 1, 0)], \quad P_\alpha \overset{\text{def}}{=} [\alpha, T_C, \bot_C] \]

In the translation from \( M2L-\text{STR}(\langle_\_ \rangle) \) to s-FA over \( A \boxtimes BDD(B1) \), if we replace the predicates \( \beta \) and \( P_\alpha \) in Figure 3 with the ones defined above, the algorithm \( \text{FA}(-) \) remains identical to the case of \( BDD(A) \). Lemma 1 and Theorem 1 carry over to \( C \).

**Example 7.** Let \( \varphi \) be the predicate \( \beta_1 \land \neg \beta_2 \land P_\alpha \) for some \( \alpha \in \Psi_A \). As a predicate of \( A \boxtimes BDD(B1) \), \( \varphi \) has the following ITE representation where leaves are BDDs:

\[ [\alpha, [\text{node}(8, \text{leaf}(0)), \text{node}(5, \text{leaf}(1), \text{leaf}(0))]], [\text{leaf}(0)] \]

As a predicate of \( BDD(A) \), \( \varphi \) has the following representation:

\[ \text{node}(8, \text{leaf}(\bot_A), \text{node}(5, \text{leaf}(\alpha), \text{leaf}(\bot_A))) \]

where the terminals (of this ADD) are predicates of \( A \).

---

10Formally, one may also consider \( BDD(B1) \) itself, but this would void the benefits of structural sharing in \( BDD(A) \).
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 $\text{BDD}(A)$ is unbounded which makes the Boolean algebra atomic 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 $\text{BDD}(A)$ and the algebra $A \otimes \text{BDD}(B1)$ implement the interface of the abstract product algebra $A \times B$ (where $B$ is $\text{BDD}(B1)$) and the restriction operation. The latter is used for “forgetting” bits for $\exists$-elimination. The implementation of the s-FA translation $\text{sFA}(\cdot)$ 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 $A \times B$ 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 nondeterministic s-FAs as it is done for NFAs in [1].

8. Experiments
We evaluate our decision procedure using both the representations of $U(A \times \mathbb{N})$ 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 $\text{BDD}(B1)$ containing unary predicates with large sets of satisfiable Boolean combinations. 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 $\text{BDD}(A)$ presented in Section 6.2 and product to refer to the algebra $A \otimes \text{BDD}(B1)$ presented in Section 6.3. All the experiments were run on an 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 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 $\text{BDD}(B1)$ is $\mathbb{N}$ and is therefore infinite.
trans, formulas with singletons

Notice that for some plots the y-axis denotes seconds while for others it denotes milliseconds. The last two plots show the difference in comparison against Mona on the Figure 6.

Table 2. M2L-STR benchmark formulas.

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>$\exists x_1, \ldots, x_k. x_1 &lt; x_2 \land \ldots \land x_{k-1} &lt; x_k$</td>
<td>$k \in {2, \ldots, 40}$</td>
</tr>
<tr>
<td>t2</td>
<td>$\exists x_1, \ldots, x_k. x_1 &lt; x_2 \land \ldots \land x_{k-1} &lt; x_k \land a(x_1) \land \ldots \land a(x_k)$</td>
<td>$k \in {2, \ldots, 40}$</td>
</tr>
<tr>
<td>t3</td>
<td>$\exists y. c(y) \lor \exists x_1, \ldots, x_k. x_1 &lt; x_2 \land \ldots \land x_{k-1} &lt; x_k \land a(x_1) \land \ldots \land a(x_k)$</td>
<td>$k \in {2, \ldots, 40}$</td>
</tr>
<tr>
<td>t4</td>
<td>$\exists x_1, \ldots, x_k. ((x_1 &lt; x_2 \land a(x_1)) \lor c(x_1)) \lor \ldots \lor ((x_{k-1} &lt; x_k \land a(x_{k-1})) \lor c(x_{k-1}))$</td>
<td>$k \in {2, \ldots, 12}$</td>
</tr>
</tbody>
</table>

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.
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.

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

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

Intuitively, the formulas in the class f1 describe strings in which the symbol at the $i$-th position, for $1 \leq i \leq k$, satisfies the predicate $\beta_i$, while the formulas in the class f2 describe strings in which for every $1 \leq i \leq k$ there exists some position in the string for which the corresponding symbol satisfies the predicate $\beta_i$. 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 formulas 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.

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 $ax \mod b = c$ 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 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 c-algebras [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 ≺ that led to the predicate trie data structure.

References
