



Reasoning About Data Trees Using CHCs

Marco Faella and Gennaro Parlato

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

August 8, 2022

Reasoning about Data Trees using CHCs^{*} ^{**}

Marco Faella¹[0000–0001–7617–5489] and Gennaro Parlato²[0000–0002–8697–2980]

¹ University of Naples Federico II, Italy
m.faella@unina.it
² University of Molise, Italy
gennaro.parlato@unimol.it



Abstract. Reasoning about data structures requires powerful logics supporting the combination of structural and data properties. We define a new logic called MSO-D (*Monadic Second-Order logic with Data*) as an extension of standard MSO on trees with predicates of the desired data logic. We also define a new class of *symbolic data tree automata* (SDTAs) to deal with data trees using a simple machine. MSO-D and SDTAs are both Turing-powerful, and their high expressiveness is necessary to deal with interesting data structures. We cope with undecidability by encoding SDTA executions as a system of CHCs (*Constrained Horn Clauses*), and solving the resulting system using off-the-shelf solvers. We also identify a fragment of MSO-D whose satisfiability can be effectively reduced to the emptiness problem for SDTAs. This fragment is very expressive since it allows us to characterize a variety of data trees from the literature, solving certain infinite-state games, etc. We implement this reduction in a prototype tool that combines an MSO decision procedure over trees (MONA) with a CHC engine (Z3), and use this tool to conduct several experiments, demonstrating the effectiveness of our approach across different problem domains.

1 Introduction

Reasoning about linear or tree-like data structures requires very expressive logics that allow combining structural and data properties. Logical characterizations of common data structures often impose restrictions on the structural part, which are intertwined with constraints on the data part. For example, in a *binary search tree* (BST) the data values are organized in the form of a binary tree, where the numerical value associated with each node is greater than or equal to all the values stored in its left sub-tree and smaller than all those in its right sub-tree. Logical characterisations of data structures may also require the calculation of measures concerning parts of the structure such as size or height. Think of *red-black trees* (RBT), a type of BST with additional constraints, such as “every path from a given node to any of its descendant leaves goes through the same number

^{*} This work was partially supported by INDAM-GNCS 2020-2021, and by AWS 2021 Amazon Research Awards.

^{**} The authors dedicate this paper to their advisor Margherita Napoli.

of black nodes”. Similarly, for AVL trees we need to impose that the heights of the sub-trees rooted in the children of any node differ by a maximum of one.

As a first contribution, we define a new logic called MSO-D (*Monadic Second-Order logic with Data*) as an extension of standard MSO on binary trees with data constraints. The MSO component of the logic allows us to express structural properties, while the data constraint component allows us to impose properties on the data associated with the nodes. Constraints on data are expressed by predicates from a desired data logic that is completely agnostic to the underlying tree structure. We connect the two components by means of uninterpreted functions that map each node of the tree to a data item. An example of an MSO-D formula that defines BSTs is:

$$\forall x. \forall y. \left((path_l(x, y) \rightarrow val(x) \geq val(y)) \wedge (path_r(x, y) \rightarrow val(x) < val(y)) \right), \quad (1)$$

where x and y are first-order variables ranging over the set of nodes, $path_l(x, y)$ (resp., $path_r(x, y)$) is an MSO formula expressing that “ y is in the left (resp., right) sub-tree of x ”, and val is an uninterpreted function that maps each node of the tree to an integer.

As a second contribution, we define a new class of *symbolic data tree automata* (SDTAs) to recognize languages of data trees using a simple machine. Such automata perform a bottom-up computation starting from the leaves of the data tree. The state of an SDTA is represented by the value of a set of *state variables*, whereas the data trees recognized by the automaton carry another set of *alphabet variables*. The transitions of an SDTA are expressed by joint constraints over state and alphabet variables. For example, BSTs attach to each node a single alphabet variable, say val , holding the numerical value of that node. An SDTA recognizing BSTs will use additional state variables to check that the data tree is indeed a BST. In this case, two state variables are sufficient to achieve this goal: one holding the minimum and one holding the maximum value stored in the sub-tree rooted in the current node. Similarly, SDTAs can be designed to recognize the classes of RBTs and AVL trees.

We have to deal with undecidable problems when reasoning about data trees using MSO-D or SDTAs, and this is unavoidable if we want to (a) handle trees with data from infinite domains, and (b) relate data from different nodes. These two features make SDTAs *Turing-powerful* since they can encode executions of two-counter machines. A similar argument holds for the satisfiability problem of MSO-D, since we can write a formula that allows us to relate data in consecutive nodes. By prohibiting the propagation of unbounded information between nodes, the decidability of relevant decision problems can be recovered (see [11,12] for an account on this). However, these features are both essential to deal with data structures such as BSTs, RBTs, AVL trees, Heaps, etc.

A way to cope with undecidability is to encode the executions of an SDTA as a system of CHCs (*Constrained Horn Clauses*) or, equivalently, as a CLP (*Constraint Logic Program*) [26], and solve the system using efficient off-the-shelf tools. Systems of CHCs correspond to a restricted class of first-order logic, and are a versatile formalism for representing and solving a variety of program verification or model checking problems, including those regarding sequential,

concurrent, and functional programs. Efficient algorithms have been proposed for solving systems of CHCs, often leveraging or generalizing techniques developed in the context of automatic program verification [21,22,2]. As a result, CHCs are often used as an intermediate representation in a variety of verification and synthesis tools [6,18,20,23,25,27,29,35]. Here, we follow a similar approach to solve the emptiness problem for SDTAS, and this offers several advantages. First, it provides a separation of concerns, allowing users of our framework to focus only on aspects related to the tree data structure at hand, while giving CHC solver developers a clean framework that can be instantiated using various model checking algorithms and specialized decision procedures. Furthermore, by expressing CHCs in the standard SMT-LIB language, one can take advantage of different CHC engines, whose performance keeps improving year-over-year, as witnessed by the *competition on constrained Horn clauses* CHC-COMP [19].

As a third contribution, we show several results linking the MSO-D satisfiability problem to the emptiness of SDTAS, and thus to the problem of solving a CHC system. A fundamental theorem for the class of regular (word or tree) languages states that a language is regular if and only if it is MSO-definable, i.e., definable by a closed formula (i.e., a sentence) of standard MSO [4,5,15,42,40,14]. Here, we show that if we allow MSO-D data predicates to talk only about the data of a single node, the satisfiability problem can be reduced to the emptiness of SDTAS. Furthermore, both decision problems are decidable in this case [11]. Moreover, we identify a larger undecidable syntactical fragment of MSO-D where the above reduction can still be performed. Namely, we give an effective reduction when the MSO-D formula is of the form $\exists \mathbf{x} \forall \mathbf{y} . \varphi(\mathbf{x}, \mathbf{y})$, where φ can contain additional quantifiers and each data constraint in φ is either unary, or accesses the data in a bounded neighborhood of the nodes referred to by \mathbf{x} and at most one of the variables of \mathbf{y} . We show that this fragment is very expressive as it allows us to characterize a variety of tree data structures from the literature, solve certain infinite-state games, and handle many other potential applications.

As a fourth and final contribution, we have implemented the reduction for the syntactic MSO-D fragment described above in a prototype tool that combines an MSO decision procedure over trees (MONA [28]) with a CHC engine (of the SMT solver Z3 [37,24]). Using this tool we have conducted several experiments to demonstrate the effectiveness and the practicality of our approach.

Organization of the paper. The rest of the paper is organized as follows. Sec. 2 defines data trees, while Sec. 3 introduces the MSO-D logic. Sec. 4 deals with the definition of SDTAS, and Sec. 5 shows that the emptiness problem for SDTAS is undecidable in general but can be solved by off-the-shelf CHC engines. Sec. 6 shows a reduction from the MSO-D satisfiability problem to the emptiness of SDTAS, for the MSO-D fragment $\exists \mathbf{x} \forall \mathbf{y} . \varphi(\mathbf{x}, \mathbf{y})$. Section 7 describes our prototype implementation and summarises our experimental evaluation. Related work and concluding remarks can be found in Sec. 8 and 9, respectively.

2 Data Trees

Here we formally define *data trees*. We deal with trees that are finite in size and labeled with data from possibly infinite domains. We consider only binary trees (i.e., trees of arity 2) to keep notation to a minimum. However, the methods and approaches presented in the paper apply to any class of trees of fixed arity.

We will use \mathbb{N} to denote the set of all natural numbers, \mathbb{Z} to stand for the set of integers, and \mathbb{B} to represent the set $\{0, 1\}$. For a number $n \in \mathbb{N}$, we write $[n]$ to denote the interval $\{1, \dots, n\}$.

Words. An *alphabet* is a finite set of *symbols*. A *word* w over an alphabet Σ is a finite (possibly empty) sequence $w = a_1 a_2 \dots a_n$ where $a_i \in \Sigma$ for $i \in [n]$. We denote with $|w|$ the *length* of the sequence of symbols forming w . The *empty word*, denoted by ϵ , is the word formed by no symbol. We denote the set of all words over Σ by Σ^* . A *language* L over Σ is any subset of Σ^* . A *prefix* (resp. *suffix*) of a word w is either ϵ , or any sequence $a_1 \dots a_j$ (resp., $a_j \dots a_n$), for some $j \in [n]$. Given two words $a = a_1 a_2 \dots a_n$ and $b = b_1 b_2 \dots b_m$, their *concatenation* denoted ab , is the word $a_1 a_2 \dots a_n b_1 b_2 \dots b_m$. Given a word $w \in \Sigma^*$ and a language $L \subseteq \Sigma^*$, we define $Ext(w, L)$ as the language of all words w' such that ww' is a word in L , i.e., $Ext(w, L) = \{w' \mid ww' \in L\}$.

Trees. A *binary tree* T , or simply a *tree*, is a finite and prefix-closed subset of $\{0, 1\}^*$. We call the elements of T *nodes*, and the node identified by ϵ the *root* of T . The *edge relation* is defined implicitly: for $d \in \{0, 1\}$, if v and vd are both nodes of T , then (v, vd) is an *edge* of T . Further, if d is 0 (resp., 1) we say that vd is the *left* (resp., *right*) *child* of v , and v is the *parent* of vd . A *leaf* is a node with no children, while an *internal node* is a node that is not a leaf. The *height* of T is $\max_{t \in T} |t|$. The *sub-tree* of T rooted at a given node $t \in T$ is $Ext(t, T)$. Further, $Ext(t, T)$ is a *left* (resp., *right*) *sub-tree* of t if $t = t'0$ (resp., $t = t'1$), for some $t' \in T$. The k -th *level* of a tree T consists of the sequence of all $t \in T$, with $|t| = k$, sorted in ascending lexicographic order. Further, the k -th *level* of T is filled *left to right* if it is a prefix of the k -th *level* of $\{0, 1\}^*$.

Data signatures. Data signatures are like structured data types (a.k.a. *records*) in programming languages: a *data signature* \mathcal{S} is a set of pairs $\{id_i : type_i\}_{i=1 \dots n}$. Common types of interest include bounded or unbounded integers (denoted by `int` and \mathbb{Z} , resp.), floating point rationals and real numbers (`float` and \mathbb{R}), the Boolean type \mathbb{B} and the bit vectors of length k . If a signature contains a single field whose type is a finite alphabet Σ , we call that signature an *enumeration*. An *evaluation* ν of a data signature \mathcal{S} is a map that associates each field name id in \mathcal{S} with a value of the corresponding type, denoted by $\nu.id$. We denote by $L(\mathcal{S})$ the set of all evaluations of \mathcal{S} , also called the *language* of \mathcal{S} .

Data Trees. A *data tree* with data signature \mathcal{S} , or an \mathcal{S} -*tree*, is a pair (T, λ) where T is a tree and λ is a labelling function that maps each node $t \in T$ into an evaluation of \mathcal{S} , i.e., $\lambda(t) \in L(\mathcal{S})$. Another way of looking at data trees is to think of them as a traditional tree data structure where the data $\lambda(t)$ associated with

each node t is structured. Thus, to simplify the notation when λ is clear from the context, we adopt a C-like notation to refer to the value of fields associated with tree nodes: if t is a tree node and id is a field of \mathcal{S} , we write $t.id$ as a shorthand for $\lambda(t).id$. If the data signature is an enumeration, we recover the traditional notion of Σ -labelled tree.

Many data structures from the literature can be seen as data trees. Below we give a high-level description of well-known data structures [9]. In addition to using them for motivating purposes, we will also use them as running examples.

Example 1 (Binary Search Trees). A BST is a binary tree where each node stores a key taken from a totally ordered set, with the property that the key stored in each internal node is greater than or equal to all the keys stored in the node's left subtree, and smaller than those in its right subtree. Thus, an appropriate signature for data trees representing BSTs is $\{val : \mathbb{Z}\}$. \square

Example 2 (Red-black Trees). An RBT is a binary tree where each internal node stores a numerical value, satisfying the binary search tree property. The leaves do not contain keys or data and they represent a NIL pointer. Each node has a color (red or black), and the following properties hold: (i) every leaf is black, (ii) if a node is red then both its children are black, and (iii) every path from a given node to a descendent leaf contains the same number of black nodes.

Note that while the color is a piece of information stored in the node, the *black height* can instead be computed on demand. Thus, the signature for data trees representing RBTs may be $\{val : \mathbb{Z}, is_black : \mathbb{B}\}$. \square

Example 3 (Max-Heap). A MAX-HEAP is a binary tree where each node stores a key taken from a totally ordered set, say \mathbb{Z} , and can be described as an \mathcal{S} -tree (T, λ) where \mathcal{S} is a data signature consisting of a single integer field, say $\{key : \mathbb{Z}\}$, that obeys the following two constraints: (i) (*shape property*) T is almost complete, i.e., all its levels are complete, except the last one, that is filled from left to right; and (ii) (*heap property*) the value stored in each node is greater than or equal to the values stored in the node's children.

3 Monadic Second-Order Logic with Data

In this section, we introduce our MSO-D logic to express properties of data trees. We define MSO-D by extending the standard Monadic Second-Order logic on (enumeration) trees (a.k.a. MSO) with constraints on the Data.

Data constraints are formulas in first-order logic (FOL) with equality (here we use standard FOL syntax and semantics [34]). However, since data trees may involve different data types, we will consider formulas with many-sorted signatures as opposed to the classical unsorted version. Specifically, we deal with formulas of a many-sorted first-order theory \mathcal{D} with sorts $data_1, \dots, data_n$. For each $data_i$, we allow a theory \mathcal{D}_{data_i} whose function symbols have type $data_i^n \rightarrow data_i$ and whose relation symbols have type $data_i^m \rightarrow \mathbb{B}$, for some n

and m . For example, each of these theories can be the theory of *arithmetic*, *reals*, *arrays*, etc. From now on, we may refer to \mathcal{D} as the *data theory* of MSO-D.

We also introduce a finite set of *connecting function symbols*, denoted \mathcal{F} , which we use to extend the MSO component of our logic with data. Let *nodes* be the sort of the MSO component. Then, each $f \in \mathcal{F}$ is an uninterpreted function symbol with type $nodes \rightarrow data_i$, for some $i \in [n]$. These functions allow us to model fields that we associate with each node in the tree. In particular, we say that $f \in \mathcal{F}$ models a field of an \mathcal{S} -tree if f is also the name of a field in \mathcal{S} . Otherwise, f may serve the purpose of endowing tree nodes with extra data fields without these being present in the labels of the data tree. This is a very useful feature for characterizing tree data structures, e.g., we can use $bh \in \mathcal{F}$ to logically characterize the *black height* of nodes in RBTs, even if that information is not part of the data signature of RBTs.

We are now ready to formally define the syntax of MSO-D over \mathcal{S} -trees with data theory \mathcal{D} and connecting functions \mathcal{F} . We fix countable sets of propositional variables (denoted by p), first-order node variables (denoted by x, y , etc.), and node-set variables (denoted by X, Y , etc.). We assume that \mathcal{D} includes relation symbols \mathcal{D}^{rel} . We also assume that the symbols in \mathcal{D} , the variable names, and the symbols in \mathcal{F} do not overlap. Since \mathcal{D} is imported in MSO-D unchanged we do not report its definition here. The remaining components of the syntax of MSO-D($\mathcal{D}, \mathcal{F}, \mathcal{S}$) are defined by the following grammar:

<p style="margin: 0;">Node terms: $t \stackrel{\text{def}}{=} x \mid t.\text{left} \mid t.\text{right}$</p> <p style="margin: 0;">Formulas: $\varphi \stackrel{\text{def}}{=} p \mid t_1 = t_2 \mid t \in X \mid \exists x. \varphi \mid \exists X. \varphi \mid \neg \varphi \mid \varphi \wedge \varphi$ $\mid r(f_1(t_1), \dots, f_k(t_k)) \quad r \in \mathcal{D}^{\text{rel}}, f_1, \dots, f_k \in \mathcal{F}$</p>
--

where r and f_1, \dots, f_k are well-typed, i.e., there is an index i such that the type of r is $data_i^k$, and for every $j \in [k]$, f_j has type $nodes \rightarrow data_i$. We denote the set of all variables occurring in φ by $Var(\varphi)$.

An *interpretation* of a formula φ is a pair (T^λ, \mathbb{I}) , where T^λ is an \mathcal{S} -tree (T, λ) , and \mathbb{I} interprets the remaining symbols of the logic. We interpret the \mathcal{D} -component of our theory as we would interpret \mathcal{D} in isolation. Assume that \mathbb{D} is the chosen interpretation of \mathcal{D} , with underlying universes D_i for sort $data_i$. Also, \mathbb{I} maps each function symbol f in \mathcal{F} with type $nodes \rightarrow data_i$ to a concrete function $\mathbb{I}(f) : T \rightarrow D_i$. Moreover, if f is also the name of a field in \mathcal{S} , then we require that $\mathbb{I}(f)$ coincides with the value of the field f in each node of the tree, i.e., $\mathbb{I}(f)(v) = \lambda(v).f$, for every $v \in T$. The satisfaction relation depends also on \mathbb{D} , but we omit it here because we consider it fixed.

We interpret the variables in $Var(\varphi)$ by mapping each of them into a subset of nodes of T with the following properties: (i) first-order variables are assigned singletons, and (ii) propositional variables are assigned either all nodes (encoding *true*) or no node at all (encoding *false*). For a set of nodes $S \subseteq T$ and a (first- or second-order) variable $\alpha \in Var(\varphi)$, we denote by $\mathbb{I}[S/\alpha]$ the function that maps α to S , and agrees with \mathbb{I} on all the other variables. Node terms are interpreted

as follows:

$$\mathbb{I}(t) = \begin{cases} \mathbb{I}(x) & \text{if } t = x \\ \mathbb{I}(s)0 & \text{if } t = s.\textit{left} \text{ and } \mathbb{I}(s)0 \in T \\ \mathbb{I}(s)1 & \text{if } t = s.\textit{right} \text{ and } \mathbb{I}(s)1 \in T \\ \mathbb{I}(s) & \text{otherwise.} \end{cases}$$

Notice that the *.left* and *.right* operators *stutter* on leaves, that is, for all leaves v it holds $v = v.\textit{left} = v.\textit{right}$.

The *satisfaction relation* $T^\lambda, \mathbb{I} \models \varphi$ is so defined.

$$\begin{array}{ll} T^\lambda, \mathbb{I} \models p & \text{iff } \mathbb{I}(p) = T \\ T^\lambda, \mathbb{I} \models t_1 = t_2 & \text{iff } \mathbb{I}(t_1) = \mathbb{I}(t_2) \\ T^\lambda, \mathbb{I} \models t \in X & \text{iff } \mathbb{I}(t) \subseteq \mathbb{I}(X) \\ T^\lambda, \mathbb{I} \models r(f_1(t_1), \dots, f_k(t_k)) & \text{iff } \mathbb{D}(r)(\mathbb{I}(f_1)(\mathbb{I}(t_1)), \dots, \mathbb{I}(f_k)(\mathbb{I}(t_k))) \\ T^\lambda, \mathbb{I} \models \neg\varphi & \text{iff } T^\lambda, \mathbb{I} \not\models \varphi \\ T^\lambda, \mathbb{I} \models \varphi_1 \wedge \varphi_2 & \text{iff } T^\lambda, \mathbb{I} \models \varphi_1 \text{ and } T^\lambda, \mathbb{I} \models \varphi_2 \\ T^\lambda, \mathbb{I} \models \exists x. \varphi & \text{iff there exists } v \in T \text{ such that } T^\lambda, \mathbb{I}[\{v\}/x] \models \varphi \\ T^\lambda, \mathbb{I} \models \exists X. \varphi & \text{iff there exists } S \subseteq T \text{ such that } T^\lambda, \mathbb{I}[S/X] \models \varphi \end{array}$$

We say that T^λ *satisfies* φ , denoted $T^\lambda \models \varphi$, if there is an interpretation \mathbb{I} such that $T^\lambda, \mathbb{I} \models \varphi$. We define the language of trees satisfying an MSO-D sentence in the usual way. An MSO-D *sentence* is a formula with no free variables. We define the set of all \mathcal{S} -trees T^λ satisfying an MSO-D($\mathcal{D}, \mathcal{F}, \mathcal{S}$) sentence φ by $\mathcal{L}(\varphi)$, i.e., the set of all \mathcal{S} -trees T^λ such that $T^\lambda \models \varphi$. A language of trees L is MSO-D($\mathcal{D}, \mathcal{F}, \mathcal{S}$) *definable* if there exists an MSO-D($\mathcal{D}, \mathcal{F}, \mathcal{S}$) sentence φ such that $L = \mathcal{L}(\varphi)$.

We recover standard MSO when \mathcal{S} is an enumeration and \mathcal{D} includes a unary relation r_a for each $a \in \Sigma$, whose interpretation is $\{a\}$.

Undecidability of the Satisfiability Problem. The *satisfiability problem* for a given MSO-D($\mathcal{D}, \mathcal{F}, \mathcal{S}$) sentence φ asks whether $\mathcal{L}(\varphi)$ is empty. Let \mathcal{D} be the theory of linear integer arithmetic. It is easy to model an execution of any given 2-counter machine using a unary data tree whose signature has two fields of type \mathbb{N} to model the counters, and an enumeration field to keep track of the current instruction. Each machine configuration is represented by a node, and we can impose constraints in our logic so that two consecutive nodes in the tree model a machine transition. Likewise, we can also express the property of a halting computation in our logic. Thus, the satisfiability of the MSO-D logic is undecidable, even though the underlying data logic \mathcal{D} is decidable. Of course, by choosing a finite domain for the interpretation of the underlying data logic, we regain decidability in that the problem matches that of the standard MSO.

Examples. We now show various examples to illustrate the expressiveness of MSO-D. We will use the usual predefined abbreviations to denote the remaining propositional connectives ($\varphi_1 \vee \varphi_2$ and $\varphi_1 \rightarrow \varphi_2$), the universal quantifier ($\forall\alpha. \varphi \stackrel{\text{def}}{=} \neg\exists\alpha. \neg\varphi$), $x \neq y \stackrel{\text{def}}{=} \neg(x = y)$, and the conditional expression

$(\varphi ? \varphi_1 : \varphi_2) \stackrel{\text{def}}{=} (\varphi \wedge \varphi_1) \vee (\neg\varphi \wedge \varphi_2)$. Finally, the following standard MSO predicates will come in handy: $child(x, y)$, $root(x)$, $leaf(x)$, and $path(x, y)$.

Example 4 (MSO-D characterization of BSTs). We define the characteristic property of BSTs on data trees with data signature $\{val : \mathbb{Z}\}$. We first introduce the auxiliary predicate $path_l(x, y)$ (resp., $path_r(x, y)$) with the meaning “ y is in the left (resp., right) sub-tree of x ”. Using these predicates, we define the MSO-D sentence (1) that says that all values in the left sub-tree of a node x contain values that are smaller than the value in x , and similarly for the right sub-tree.

To demonstrate the use of connecting functions to model auxiliary node fields, we give an alternative way to characterize BSTs. We introduce two auxiliary connecting functions: min and max . We impose constraints to ensure that for each node x in the tree $min(x)$ and $max(x)$ are the minimum and maximum values of the sub-tree rooted in x , respectively. It is straightforward to see that we can impose the BST property by relating the values in each node with min and max in their children as follows:

$$\psi_{\text{bst}} \stackrel{\text{def}}{=} \forall x. \left(\begin{array}{l} (x \neq x.\text{left} \ ? \ min(x) = min(x.\text{left}) \wedge max(x.\text{left}) \leq val(x) \\ \quad : \ min(x) = val(x) \end{array} \right) \wedge \\ \left(\begin{array}{l} (x \neq x.\text{right} \ ? \ max(x) = max(x.\text{right}) \wedge min(x.\text{right}) > val(x) \\ \quad : \ max(x) = val(x) \end{array} \right) \right). \quad \square$$

Example 5 (MSO-D characterization of RBTs). We can also express the defining properties of red-black trees as follows:

- (a) *Every leaf is black:* $\forall x. leaf(x) \rightarrow is_black(x)$.
- (b) *If a node is red, both its children are black:*

$$\forall x. (\neg is_black(x)) \rightarrow (is_black(x.\text{left}) \wedge is_black(x.\text{right})).$$

- (c) *Every path from a node to a leaf contains the same number of black nodes.*

We encode this property as the consistency of the black height data field bh :

$$\forall x. \forall y. \left(\begin{array}{l} \left((is_black(x) \wedge child(y, x)) \rightarrow (bh(y) = bh(x) - 1) \right) \\ \wedge \left((\neg is_black(x) \wedge child(y, x)) \rightarrow (bh(y) = bh(x)) \right) \\ \wedge \left((leaf(x) \wedge is_black(x)) \rightarrow bh(x) = 1 \right) \\ \wedge \left((leaf(x) \wedge \neg is_black(x)) \rightarrow bh(x) = 0 \right) \end{array} \right). \quad \square$$

Extended models. Since we are going to build automata corresponding to formulas with free variables, it is convenient to encode the variable interpretation in the tree itself, by expanding the data signature with an extra Boolean flag for each free variable. The flag corresponding to a free variable will be set to 1 in the node(s) that belong to the interpretation of that variable. In detail, for a given interpretation (T^λ, \mathbb{I}) , assume that $Var(\varphi) = \{\alpha_1, \dots, \alpha_n\}$, we can define an *extended tree* (T, λ^E) with data signature $\mathcal{S}^E = \mathcal{S} \cup \{a_1, \dots, a_n : \mathbb{B}\}$, where $\lambda^E(u)(a_i) = 1$ iff $u \in \mathbb{I}(\alpha_i)$. Conversely, from an extended tree we can extract the corresponding variable interpretation. For such an extended tree we can write $T^{\lambda^E} \models \varphi$ without mentioning \mathbb{I} .

4 Symbolic Data-Tree Automata

In this section, we define a new class of tree automata called *Symbolic Data-Tree Automata*. They generalize traditional bottom-up finite tree automata as they work with data trees. Furthermore, they are symbolic because the alphabet and set of states are defined using evaluations of data signatures, and its transition function is defined through constraints³ involving states and alphabet.

Definition 1 (SYMBOLIC DATA-TREE AUTOMATA). A symbolic data-tree automaton, or SDTA for short, \mathcal{A} is a tuple $(\mathcal{S}^\Sigma, \mathcal{S}^Q, \psi^F, \Psi^\Delta)$ where:

- \mathcal{S}^Σ is the alphabet data signature defining the tree alphabet $\Sigma = L(\mathcal{S}^\Sigma)$;
- \mathcal{S}^Q is the state data signature defining the set of states $Q = L(\mathcal{S}^Q)$;
- ψ^F is a unary constraint defining the set of final states $F \subseteq Q$, i.e., the set consisting of all elements $q \in Q$ such that $\psi^F(q)$ evaluates to true;
- Ψ^Δ is a tuple of four transition constraints:

$$\psi_{lr}(q_l, q_r, \sigma, q), \quad \psi_l(q_l, \sigma, q), \quad \psi_r(q_r, \sigma, q), \quad \psi_{leaf}(\sigma, q),$$

where q_l, q_r , and q are variables of type \mathcal{S}^Q , and σ is of type \mathcal{S}^Σ .

\mathcal{A} accepts \mathcal{S}^Σ -trees. A tree (T, λ) is accepted by \mathcal{A} if there is a total function $\pi : T \rightarrow Q$ such that for every node $t \in T$ the following holds:

- t has both children, and $\psi_{lr}(\pi(t_0), \pi(t_1), \lambda(t), \pi(t))$ holds;
- t has only the left child, and $\psi_l(\pi(t_0), \lambda(t), \pi(t))$ holds;
- t has only the right child, and $\psi_r(\pi(t_1), \lambda(t), \pi(t))$ holds;
- t is a leaf, and $\psi_{leaf}(\lambda(t), \pi(t))$ holds;
- $\psi^F(\pi(\epsilon))$ holds.

The language of \mathcal{A} , denoted $L(\mathcal{A})$, is the class of all \mathcal{S}^Σ -trees accepted by \mathcal{A} . \square

We recover standard tree automata when both data signatures \mathcal{S}^Σ and \mathcal{S}^Q are enumerations. In that case, we call \mathcal{A} an *enumeration tree automaton* and we denote it as (Σ, Q, F, Δ) , where $\Sigma = L(\mathcal{S}^\Sigma)$, $Q = L(\mathcal{S}^Q)$, and so on.

Example 6 (Symbolic Data-Tree Automaton for Max Heap). We define an SDTA \mathcal{A}_{bmh} where $L(\mathcal{A}_{bmh})$ is the set of all max heaps. The state data signature of \mathcal{A}_{bmh} is $\{h : \mathbb{N}, f : \mathbb{B}, val : \mathbb{N}\}$. We use the h field to store the height of the sub-tree rooted in the node, the f field to store whether the sub-tree rooted in the node is complete with the last level completely filled, and val stores the node's data value. The transition data constraints are as follows:

$$\begin{aligned} \psi_{leaf}(\sigma, q) &\stackrel{\text{def}}{=} q.h = 1 \wedge q.f \wedge q.val = \sigma.val \\ \psi_l(q_l, \sigma, q) &\stackrel{\text{def}}{=} q_l.h = 1 \wedge q.h = 2 \wedge \neg q.f \wedge \sigma.val \geq \sigma_l.val \wedge q.val = \sigma.val \\ \psi_{lr}(q_l, q_r, \sigma, q) &\stackrel{\text{def}}{=} (q_l.h - 1 \leq q_r.h \leq q_l.h) \wedge (q.f \leftrightarrow (q_l.f \wedge q_r.f \wedge q_l.h = q_r.h)) \\ &\quad \wedge (\neg q_l.f \rightarrow (q_r.h < q_l.h \wedge q_r.f)) \wedge (\neg q_r.f \rightarrow (q_r.h = q_l.h \wedge q_l.f)) \\ &\quad \wedge \sigma.val \geq q_l.val \wedge \sigma.val \geq q_r.val \wedge q.val = \sigma.val. \end{aligned}$$

³ We use the term *constraint* to denote a generic predicate $con(x_1, \dots, x_k)$ in which the type of variable x_i is some data signature \mathcal{S}_i . We deliberately leave the definition of the constraints unspecified, and specify them only when it is necessary to do so.

For each leaf, we set the height field h to 1, the field f to `true`, and copy the label of the node into the state field val . Note that all sub-trees of a complete tree are still complete trees. Thus, if a node has only the left child, this child must be a leaf, and we set the parent node’s fields accordingly. A node with only the right child leads to a violation of the shape property, thus $\psi_r(q_r, \sigma, q) \stackrel{\text{def}}{=} \text{false}$. Finally, we consider the case where the node has both children. Here ψ_{lr} constrains the fields of the state data signature to guarantee their invariants. Specifically, the first two lines enforce the shape property while the last line enforces the heap property and copies the value of the label into the state field val . To conclude, we define ψ^F as a tautology. However, if, for example, we wanted to accept only max heaps of height at least 100 we could have defined $\psi^F(q) \stackrel{\text{def}}{=} (q.h \geq 100)$. \square

5 Solving the Emptiness problem for SDTAs

The *emptiness problem* for SDTAs consists in determining whether the tree language recognized by a given SDTA \mathcal{A} is empty, i.e., whether $L(\mathcal{A})$ is empty. We first prove that the emptiness problem for SDTAs is undecidable, and then show that it can be reduced to the satisfiability of a system of constrained Horn clauses (CHCs), for which increasingly efficient off-the-shelf semi-procedures exist.

It is well known that the emptiness problem for tree automata is decidable [8]. However, as explained for MSO-D in Sec. 3, as soon as the state data signature involves an unbounded data domain (such as integers or reals) and basic arithmetics (e.g., increment and test for zero), the emptiness problem becomes undecidable. Thus, we have the following.

Theorem 1. *The emptiness problem for SDTAs is undecidable.*

We cope with this negative result by providing a reduction to the satisfiability of a system of CHCs, when the transition constraints of the automaton are defined through quantifier-free first-order logic formulas.

Constrained Horn Clauses. We fix a set R of uninterpreted fixed-arity relation symbols, which represent the unknowns in the system. A *Constrained Horn Clause*, or CHC for short, is a formula of the form $H \leftarrow C \wedge B_1 \wedge \dots \wedge B_n$ where:

- C is a constraint over some background theory that does not contain any application of predicates in R ;
- for every $i \in [n]$, B_i is an application $p(v_1, \dots, v_k)$ of a relation symbol $p \in R$ to first-order variables v_1, \dots, v_k ;
- H is the clause *head* and, similarly to B_i , is an application $p(v_1, \dots, v_k)$ of a relation symbol $p \in R$ to the first-order variables, or *false*;
- the first-order variables appearing in the signature of the predicates and constraints are all implicitly universally quantified.

A finite set \mathcal{H} of CHCs is a *system*, and it corresponds to the first-order formula obtained by putting all its CHCs in conjunction. We assume that the semantics of constraints is given a priori as a structure. A system \mathcal{H} with relation

symbols R is *satisfiable* if there is an interpretation to each predicate in R that makes all clauses in \mathcal{H} valid.

It is a well-known result from constraint logic programming that every system of CHCs \mathcal{H} has a unique minimal model that can be computed as the fixed-point of an operator derived by the clauses of \mathcal{H} [16,26]. This property, which allows us to use a fixed-point semantics for CHC systems, to justify the correctness of the reduction defined below (i.e., Theorem 2).

Reduction. We give a linear time reduction from the emptiness problem for SDTAs to the satisfiability of systems of CHCs. Let $\mathcal{A} = (\mathcal{S}^\Sigma, \mathcal{S}^Q, \psi^F, \Psi^\Delta)$ be an SDTA with $\Psi^\Delta = (\psi_{lr}, \psi_l, \psi_r, \psi_{leaf})$, q_l, q_r and q be structured variables of type \mathcal{S}^Q , σ be a structured variable of type \mathcal{S}^Σ , and $h(q)$ be an uninterpreted predicate. We map \mathcal{A} into the CHC system $\mathcal{H}_\mathcal{A}$ formed by the CHCs shown on the right.

$$\begin{array}{l} h(q) \leftarrow \psi_{lr}(q_l, q_r, \sigma, q) \wedge h(q_l) \wedge h(q_r) \\ h(q) \leftarrow \psi_l(q_l, \sigma, q) \wedge h(q_l) \\ h(q) \leftarrow \psi_r(q_r, \sigma, q) \wedge h(q_r) \\ h(q) \leftarrow \psi_{leaf}(\sigma, q) \\ false \leftarrow \psi^F(q) \wedge h(q) \end{array}$$

Theorem 2 (EMPTINESS). *Let \mathcal{A} be an SDTA. Then, $L(\mathcal{A})$ is empty if and only if $\mathcal{H}_\mathcal{A}$ is satisfiable.*

6 From Logic to Automata

In this section, we describe a reduction from the satisfiability problem of MSO-D to the emptiness problem of SDTAs, when the MSO-D formula φ is a sentence in the following form:

$$\varphi = \exists x_1, \dots, x_n. \forall y_1, \dots, y_m. \theta, \quad (2)$$

where each data constraint of the formula θ , say $r(f_1(t_1), \dots, f_k(t_k))$, satisfies one of the following:

- r is unary (i.e., $k = 1$), or
- r depends only on variables x_1, \dots, x_n and at most one of the variables y_1, \dots, y_m , i.e., $Var(t_1, \dots, t_k) \subseteq \{x_1, \dots, x_n, y_i\}$, for some $i \in [m]$.

Notice that θ may contain other quantifiers, but the additional quantified variables can occur only inside unary data constraints. Moreover, it is easy to see that this fragment is closed under positive Boolean combinations (i.e., conjunctions and disjunctions).

This fragment strictly includes the MSO logic with data defined in [11] for data words, which only allows unary data constraints. Below we show that the added expressivity can be used to define and verify properties of a variety of data structures, including those from Examples 4 and 5, and infinite-state games.

In our reduction, we first construct a standard finite-state tree automaton over a finite alphabet (Sec. 6.1), which we then convert to an SDTA (Sec. 6.2).

6.1 Building the enumerated tree automaton

The first step in our reduction from MSO-D to SDTAS is to convert the MSO-D formula φ of type (2) into a formula φ' in standard MSO by *abstracting away* all data constraints. We distinguish two types of data constraints. *Global constraints* refer only to the data of the existentially quantified variables x_i ; on a given data tree, once the interpretation of those variables is chosen, each global constraint is either *true* or *false*: it is a global property of the tree. *Local constraints*, instead, additionally refer to a variable, say z , that is not one of $\{x_1, \dots, x_n\}$; even if the interpretation of $\{x_1, \dots, x_n\}$ is fixed, the truth of such constraints depends on the interpretation of z . Accordingly, we replace each data constraint in θ , say $r(f_1(t_1), \dots, f_k(t_k))$, as follows:

Global constraints. If $\text{Var}(t_1, \dots, t_k) \subseteq \{x_1, \dots, x_n\}$, we replace all occurrences of the data constraint with a new propositional variable p . We denote by p_1, \dots, p_h all such propositional variables.

Local constraints. Otherwise, there is a unique variable $z \in \text{Var}(t_1, \dots, t_k)$ that is *not* one of $\{x_1, \dots, x_n\}$. We then introduce a new free second-order variable C , and replace each occurrence of the above data constraint with the clause $z \in C$. We denote by C_1, \dots, C_l all the second-order variables introduced in this process.

Besides the above substitutions, in the resulting MSO formula we leave variables x_1, \dots, x_n free, so that the models of the formula will carry the interpretation of those variables as extra bits in the node labels (recall the discussion on extended models in Sec. 3). We thus obtain the following MSO formula:

$$\varphi' \stackrel{\text{def}}{=} \forall y_1, \dots, y_m. \theta'. \quad (3)$$

Since φ' has no data constraints, we can take its data signature to be empty.

Example 7. Consider the formula ψ_{bst} from Example 4 that defines BSTs using auxiliary data *min* and *max*. Since it uses a single universal quantifier, it belongs to the syntactic fragment (2). For the sake of simplicity, consider a stronger formula ψ'_{bst} forcing internal nodes to have two children (a.k.a. a *full* BST):

$$\begin{aligned} \psi'_{\text{bst}} &\stackrel{\text{def}}{=} \psi_{\text{bst}} \wedge \forall y. \text{full_tree}(y), \quad \text{where} \\ \text{full_tree}(y) &\stackrel{\text{def}}{=} (\neg \text{leaf}(y) \rightarrow (y.\text{left} \neq y \wedge y.\text{right} \neq y)). \end{aligned}$$

Now, consider the following true property of full BSTs: the successor of an internal node is the left-most leaf in its right sub-tree. The following formula states the opposite of that property:

$$\begin{aligned} \psi_{\text{succ}} &\stackrel{\text{def}}{=} \exists x_1, x_2, x_3. \left((\text{val}(x_1) < \text{val}(x_2) < \text{val}(x_3)) \wedge \right. \\ &\quad \left. \neg \text{leaf}(x_1) \wedge \text{leaf}(x_3) \wedge \text{left_only_path}(x_1.\text{right}, x_3) \right). \end{aligned}$$

It is easy to see that $\psi'_{\text{bst}} \wedge \psi_{\text{succ}}$ is equivalent to a formula ψ in our fragment:

$$\begin{aligned} \exists x_1, x_2, x_3. \forall y. & \left((val(x_1) < val(x_2) < val(x_3)) \wedge \neg leaf(x_1) \wedge leaf(x_3) \right. \\ & \wedge left_only_path(x_1.right, x_3) \wedge full_tree(y) \\ & \wedge (y \neq y.left ? min(y) = min(y.left) \wedge max(y.left) < val(y) : min(y) = val(y)) \\ & \left. \wedge (y \neq y.right ? max(y) = max(y.right) \wedge min(y.right) > val(y) : max(y) = val(y)) \right). \end{aligned}$$

The conversion outlined above turns ψ into the following MSO formula:

$$\forall y. \left(p_1 \wedge \neg leaf(x_1) \wedge leaf(x_3) \wedge left_only_path(x_1.right, x_3) \wedge full_tree(y) \right. \\ \left. \wedge (y \neq y.left ? y \in C_1 : y \in C_2) \wedge (y \neq y.right ? y \in C_3 : y \in C_4) \right),$$

where proposition p_1 corresponds to the global constraint $val(x_1) < val(x_2) < val(x_3)$, the second-order variable C_1 corresponds to the local constraint $min(y) = min(y.left) \wedge max(y.left) < val(y)$, and variables $C_2 - C_4$ correspond to the other data constraints in ψ_{bst} . \square

We now apply the standard MSO construction to φ' , leading to a bottom-up finite-state tree automaton $A_{\varphi'}$ on the alphabet $\Sigma = \{0, 1\}^{n+h+l}$, accepting all finite trees that represent interpretations satisfying φ' . The alphabet is Σ because $n + h + l$ is the total number of free variables in φ' : n first-order variables x_i , h propositional variables p_i (corresponding to global constraints), and l second-order variables C_i (corresponding to local constraints). We recall the formal statement of this construction below, for more details see [41] and [8].

Theorem 3. *For all MSO formulas φ' on the empty data signature, with free first-order variables x_1, \dots, x_n , propositional variables p_1, \dots, p_h , and second-order variables C_1, \dots, C_l , there is a deterministic bottom-up tree automaton on the alphabet $\{0, 1\}^{n+h+l}$ whose language consists of all extended trees T such that $T \models \varphi'$.*

Simplifying assumptions. To simplify the presentation of the following constructions, we make two simplifying assumptions. First, we assume that all terms appearing in data constraints are variables, and not composite terms like $x.left.right$. Dropping this assumption is technically simple and omitted due to space constraints. Second, we assume that all connecting functions f appearing in data constraints correspond to fields in \mathcal{S} . Sentences that satisfy the second assumption have a unique interpretation \mathbb{I} , because they have no free variables and the connecting functions must be interpreted as the functions extracting the corresponding field from each node. We discuss how to remove this restriction in Sec. 6.3.

We now establish a relation between Σ -trees accepted by $A_{\varphi'}$, and data trees on the data signature \mathcal{S} defined by φ . Denote by $(a_1, \dots, a_n, b_1, \dots, b_h, c_1, \dots, c_l)$ the generic element of Σ . Given a Σ -tree (T, σ) and a variable x_i in φ' , we define $node(\sigma, x_i)$ to be the unique node $u \in T$ such that the a_i component of $\sigma(u)$ is 1. In words, the function $node$ picks the position in the tree where the Σ -tree activates the bit a_i .

Definition 2. Consider an MSO-D sentence φ of the form (2) on the data signature \mathcal{S} , and let \mathbb{I} be its unique interpretation. We say that a Σ -tree (T, σ) and an \mathcal{S} -tree (T, λ) are consistent iff for all nodes $u \in T$ the following hold:

1. For all $i \in [h]$, let $r_i^{\text{glb}}(f_i^1(\alpha_i^1), \dots, f_i^{j_i}(\alpha_i^{j_i}))$ be the global constraint from φ corresponding to the propositional variable p_i from φ' . Recall that under the simplifying assumptions each α_i^j is one of x_1, \dots, x_n , and each f_i^j is one of the names of the fields of \mathcal{S} . Then, $\sigma(u)(b_i) = 1$ iff the following holds

$$\mathbb{D}(r_i^{\text{glb}})(\mathbb{I}(f_i^1)(\text{node}(\sigma, \alpha_i^1)), \dots, \mathbb{I}(f_i^{j_i})(\text{node}(\sigma, \alpha_i^{j_i}))).$$

2. For all $i \in [l]$, let $r_i^{\text{loc}}(g_i^1(\beta_i^1), \dots, g_i^{k_i}(\beta_i^{k_i}), g_i(z_i))$ be the local constraint from φ corresponding to the second-order variable C_i from φ' . Recall that each β_i^j is one of x_1, \dots, x_n , and each g_i^j (as well as g_i) is one of the names of the fields of \mathcal{S} . Then, $\sigma(u)(c_i) = 1$ iff the following holds

$$\mathbb{D}(r_i^{\text{loc}})(\mathbb{I}(g_i^1)(\text{node}(\sigma, \beta_i^1)), \dots, \mathbb{I}(g_i^{k_i})(\text{node}(\sigma, \beta_i^{k_i})), \mathbb{I}(g_i)(\lambda(u))).$$

The following result states the fundamental relationship between φ and $A_{\varphi'}$.

Theorem 4. Let φ be an MSO-D sentence of the form (2) on the data signature \mathcal{S} , and let $A_{\varphi'}$ be the corresponding tree automaton described above. For all data trees (T, λ) with data signature \mathcal{S} , the following are equivalent:

1. it holds $T^\lambda, \mathbb{I} \models \varphi$, where \mathbb{I} is the unique interpretation of φ ;
2. there exists a tree $(T, \sigma) \in L(A_{\varphi'})$ s.t. (T, λ) and (T, σ) are consistent.

6.2 Building the symbolic data tree automaton

We now convert the tree automaton from the previous section into an SDTA that accepts all and only the data trees satisfying the original MSO-D formula φ .

Intuitively, the SDTA mimics the behavior of the tree automaton, and in doing so, it enforces the data constraints contained in φ . The information about which constraints should be true and which should be false at every node is encoded in the alphabet $\Sigma = \{0, 1\}^{n+h+l}$ of the tree automaton. In detail, if $(a_1, \dots, a_n, b_1, \dots, b_h, c_1, \dots, c_l)$ is a generic symbol from the alphabet, the b_i 's encode the truth value of the global constraints, and the c_i 's encode the truth value of the local constraints. However, the *data* on which to evaluate those constraints comes from different sources. The global constraints are evaluated only on the *guessed* data for the existentially quantified variables x_1, \dots, x_n , whereas the local constraints also access the data of the current node.

Finally, the a_i component of the alphabet encodes the actual position of each x_i in the current tree (i.e., a_i is 1 only in the node that is the interpretation of x_i). So, when $a_i = 1$ the symbolic automaton checks that the guessed data evaluation for x_i corresponds to the data in the current node.

Let $A_{\varphi'} = (\Sigma, Q, F, \Delta)$ be the tree automaton from Sec. 6.1, we now define the SDTA $\mathcal{A}_\varphi = (\mathcal{S}, \mathcal{S}^Q, \psi^F, \Psi^\Delta)$. First, notice that the alphabet data signature

\mathcal{S} coincides with that of the original MSO-D formula. We then set the state data signature \mathcal{S}^Q to $\{state : Q\} \cup \{id^i : type \mid (id : type) \in \mathcal{S}, i = 1 \dots n\}$, i.e., \mathcal{S}^Q contains an enumerated data field representing the state of the tree automaton $\mathcal{A}_{\varphi'}$, and n copies of each data field in \mathcal{S} . These copies are used to store the guessed data evaluations for the existentially quantified variables x_i from (2). For a symbolic state $q \in L(\mathcal{S}^Q)$ and $i \in [n]$, we denote by $q[x_i]$ the i -th projection of q on \mathcal{S} , i.e., the evaluation that assigns to each field id in \mathcal{S} the value $q.id^i$. The acceptance constraint $\psi^F(q)$ is simply defined as $q.state \in F$.

Regarding the transition constraints Ψ^Δ , we will focus only on the case of nodes with two children, since the other cases are similar. Let (s_l, s_r, a, s) be a transition in $\mathcal{A}_{\varphi'}$, where $a = (a_1, \dots, a_n, b_1, \dots, b_h, c_1, \dots, c_l) \in \Sigma$. We add the following implicant to the transition constraint ψ_{lr} :

$$\left\{ \begin{array}{l} q_l.state = s_l \wedge q_r.state = s_r \wedge q.state = s \end{array} \right. \quad (4a)$$

$$\wedge \bigwedge_{i \in [n]} (q[x_i] = q_l[x_i] \wedge q[x_i] = q_r[x_i]) \quad \wedge \quad \bigwedge_{\{i \mid a_i=1\}} (q[x_i] = \sigma) \quad (4b)$$

$$\wedge \bigwedge_{i \in [h]} \left[(b_i = 1) \leftrightarrow \mathbb{D}(r_i^{\text{glb}})(q[\alpha_i^1].f_i^1, \dots, q[\alpha_i^{j_i}].f_i^{j_i}) \right] \quad (4c)$$

$$\wedge \bigwedge_{i \in [l]} \left[(c_i = 1) \leftrightarrow \mathbb{D}(r_i^{\text{loc}})(q[\beta_i^1].g_i^1, \dots, q[\beta_i^{k_i}].g_i^{k_i}, \sigma.g_i) \right] \quad (4d)$$

$$\implies \psi_{lr}(q_l, q_r, \sigma, q).$$

The above conjuncts can be explained as follows: (4a) mimics the state change in the discrete transition, the first part of (4b) states that the n copies of the data fields held by the symbolic automaton are uniform over the whole tree, the second part of (4b) additionally states that the i -th copy of the data fields coincides with the data σ in the unique node where the discrete automaton prescribes $a_i = 1$, (4c) enforces the i -th global constraint r_i^{glb} in all nodes where the discrete automaton prescribes $b_i = 1$, and finally (4d) enforces the local constraints when the c_i component of the discrete alphabet is 1.

Theorem 5. *Let φ be an MSO-D sentence of the form (2) and let \mathcal{A}_φ be the corresponding SDTA described above. We have $\mathcal{L}(\varphi) = L(\mathcal{A}_\varphi)$.*

6.3 Supporting auxiliary data

So far, we have assumed that all connecting function symbols f appearing in the data constraints correspond to fields in \mathcal{S} . In other words, all data constraints refer to data fields that are present in the trees. However, our logic also supports connecting function symbols that do not correspond to fields in the data signature. In that case, the interpretation is free to assign any value to $f(u)$, for each node u in the data tree. Thus, the SDTA \mathcal{A}_φ must accept a data tree if there exists an interpretation for those functions that satisfies the formula. To achieve this effect, let $\{f_i\}_{i=1 \dots k}$ be the set of connecting function symbols occurring in φ and not corresponding to data fields in \mathcal{S} , where f_i has type $nodes \rightarrow data_i$.

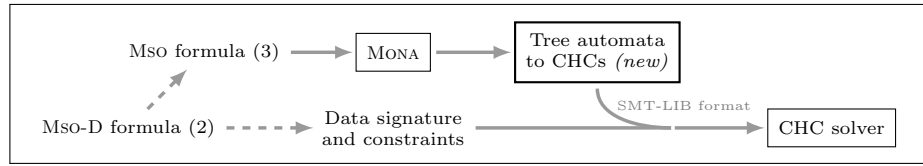


Fig. 1: Architecture of the prototype implementation. Dashed transformations are performed manually, but could be automated by an MSO-D parser.

Define the extended data signature

$$\mathcal{S}' = \mathcal{S} \cup \{f_i : data_i\}_{i=1\dots k}.$$

We enrich the state data signature of \mathcal{A}_φ as follows:

$$\mathcal{S}^Q = \{state : Q\} \cup \{name^i : type \mid (name : type) \in \mathcal{S}', i = 0 \dots n\}.$$

Compared to the original definition from Sec. 6.2, we store an extra copy of the data fields, identified by index 0, representing the data in the current node. Moreover, all copies include the auxiliary data fields. It is straightforward to adapt the constraint Ψ^Δ from Sec. 6.2 to support such auxiliary data fields.

7 Implementation and Experiments

We implemented a prototype toolchain supporting our framework as shown in Fig. 1. Instead of developing an MSO-D parser, we provide an MSO formula already in the form (3), and supply the data constraints and the data signature for the formula in a separate file, directly in the SMT-LIB format. Next, we convert the MSO formula into an equivalent tree automaton, and in turn into a system of CHCs (as described in Sec. 5). We used Z3 v4.8.10 (64bit for Windows 10) [36] as the CHC solver, and MONA v1.4 [28] as the MSO-to-automata translator. The only new piece of code required by this implementation is the converter from the MONA tree automaton format to CHCs in the SMT-LIB language, which is a simple one-to-one textual transformation. For the experiments, we used a dedicated machine with 16GB of physical memory and an AMD Ryzen 7 2700X clocked at 3.7Ghz, running Windows 10.

7.1 Proving Properties of Tree Data Structures

Consider the property of full BSTs described in Example 7, namely that the successor of an internal node is the left-most leaf in its right subtree. We submitted to our tool the conjunction $\psi'_{\text{bst}} \wedge \psi_{\text{succ}}$, which would be satisfied only by a full BST where the successor of an internal node is *not* the left-most leaf in its right subtree (property SUCCESSOR). Once the formula is converted into a system of CHCs, the SMT solver proves satisfiability of the system (and hence, unsatisfiability of the original formula) in less than a second.

Example	MSO-D property	Number of CHCs	Result	Time (Z3)
RBTs	BLACKHEIGHT	76	unsat	0.3"
Full BSTs	SUCCESSOR	945	unsat	0.2"
	STEPMOTHER(1.0)		sat	3'
	STEPMOTHER(1.5)		sat	7' 41"
Cinderella	STEPMOTHER(1.8)	23,387	sat	11' 56"
	STEPMOTHER(2.0)		unsat	1h 54'
	STEPMOTHER(3.0)		unsat	1h 16'

Table 1: MSO-D satisfiability experiments (Sec. 7).

For RBTs, we consider the property that there exists an internal node whose black height is less than half of its height (property BLACKHEIGHT). Our approach can prove that this property is unsatisfiable on RBTs in less than a second. Both experiments are summarized in Table 1.

7.2 Solving an Infinite-State Game

Our approach can be used to solve certain infinite-state games, such as the Cinderella-Stepmother game [3,1]. In this software synthesis benchmark, two players share n buckets, each holding up to c units of water. The buckets are positioned in a circle and are initially empty. The game is played in a discrete sequence of turns: when it is Cinderella’s turn, she can empty two adjacent buckets. When it is the Stepmother’s turn, she can pour water into any subset of buckets, for a total of 1 unit of water. If any of the buckets overflows, Stepmother wins. If the game continues forever with no overflows, Cinderella wins. It can be described as an infinite-state turn-based two-player game of infinite duration with a safety objective (for Cinderella). Notice how not only the game state-space is infinite, but so are the moves available to Stepmother at each step.

Given values for the parameters n and c , we build an MSO-D formula $\varphi_{n,c}$ that is satisfiable if and only if Stepmother wins the game with those parameters. The formula holds true on finite trees representing winning strategies of Stepmother. In other words, a tree that satisfies φ tracks all possible game plays where Stepmother pours water according to a specific deterministic plan and Cinderella takes all possible moves. Due to space constraints, further details on the encoding are deferred to an extended version of this paper.

In our experiments, we fixed the number of buckets n to 5 and checked the satisfiability of $\varphi_{5,c}$ for various values of the capacity c . In Table 1, we denote by STEPMOTHER(c) the formula $\varphi_{5,c}$. Bodlaender et al. [3], among their comprehensive analysis of this game, show that for $n = 5$, the minimum capacity for which Cinderella wins the game is $c = 2$ (see Table 1 in [3]). Their proof for this case is manual. Other cases were settled with the help of an SMT solver, using invariants based on non-trivial insights on the reasonable strategies of Stepmother. On the contrary, our encoding based on MSO-D employs only the rules of the game, with no further constraints on the players’ moves.

Our setup successfully solves the game for various values of the capacity. The time needed by the SMT solver is very uneven, ranging from three minutes to a maximum of almost two hours for $c = 2$. That is explained by the fact that $c = 2$ is the hardest case for Cinderella to win the game. Therefore, proving that property requires building a complex winning strategy for Cinderella. Such strategy is embedded in the proof of unsatisfiability, and extracting it would be an interesting exercise beyond the scope of the present paper. When the capacity moves away from the critical threshold in either direction, the solving time visibly decreases.

8 Related Work

Our work is related to many works in the literature in different ways. In addition to the works already mentioned in the introduction, here we focus on those that seem to be closest to the results presented in this paper.

Automata on infinite alphabets. Symbolic finite automata (SFAs) [44] and symbolic tree automata (STAs) [43] replace the traditional finite alphabet by a decidable theory of unary predicates over a possibly infinite domain. They predicate over data words and trees, but they do not support storing, comparing, or combining data from different positions in the model, as that leads quickly to undecidability. Symbolic register automata [10] extend SFAs by storing data values in a set of registers. They retain decidability of the emptiness problem by only allowing equality comparisons between registers and input data.

Recently, Shimoda et al. [39] introduced *symbolic automatic relations* (SARs) as a formalism to verify properties of recursive data structures. While both MSO-D and SARs rely on CHCs as a backend, they differ in motivation and purpose. SARs aim at encoding specific properties of interest in a way that reduces the verification effort of the underlying CHC solver, whereas MSO-D is intended to provide a high-level language that can be compiled into CHCs.

Decidable logics with data extensions. In [11], D’Antoni et al. design an extension of WS1S on finite data sequences where data can be examined with arbitrary predicates from a decidable theory, similarly to the capabilities of SFAs. They develop custom representations and algorithms to efficiently solve the satisfiability problem by reducing it to the emptiness of SFAs. Colcombet et al. [7] study a decidable fragment of MSO with data equality, called *rigidly guarded MSO* \sim , where data equality constraints of the type $val(x) = val(y)$ can only be checked on a single y -position for each x -position. Constraint LTL [13] is another decidable logic for infinite data words, where data in different positions can be compared for equality and for order. Segoufin [38] provides a wider, albeit slightly outdated, perspective on decidable data logics and automata.

Logics for automated reasoning about heap-manipulating programs. Similarly to MSO-D, STRAND [32] is a logic that combines MSO on tree-like graphs with the theory of integers. Although STRAND has a fragment that admits a decidable and efficient decision procedure, it is not sufficiently expressive to state properties

of classic data structures such as the balancedness of a tree. Also it does not allow solving the Cinderella-Stepmother game. DRYAD logic [33] is a quantifier-free logic supporting recursion on trees, that is deliberately undecidable but admits a sound, incomplete, and terminating validity procedure, based on *natural* proofs [30]. DRYAD recursive definitions could be expressed by our SDTAs that uses the theories of integers and integer (multi)sets; vice versa, proof techniques developed for DRYAD could be used to check the emptiness of (some) SDTAs.

Infinite-state games. Many infinite-state reachability games like the Cinderella game of Sec. 7.2 can be encoded in MSO-D, including all the reachability games used in the experiments performed by Farzan and Kincaid [17]. In that paper, the authors present a fully automated but incomplete approach for the (undecidable) class of *linear arithmetic games*. Our approach is incomparable to theirs: on the one hand, the approach proposed in Sec. 7.2 does not extend to all linear arithmetic games, because it assumes that one player has a bounded number of moves; on the other hand, we could easily handle games whose transition relations is not limited to linear arithmetic. Another related approach is presented by Beyene et al. [1], who reduce infinite-state games to CHCs extended with existential quantifiers. Such existential quantifiers are handled with the help of user-provided templates.

9 Conclusions and Future Directions

We presented MSO-D and SDTAs as extensions of MSO on trees and finite-state tree automata, respectively, for the purpose of reasoning about data trees. We have shown that these are versatile and powerful models for reasoning about relevant problems, outside the realm of classical automata theory. We believe that the key idea, namely separating the structural properties of interest from the data constraints, makes it easier to reason about challenging problems.

Several future directions are interesting. First, we may want to investigate theoretical questions about SDTAs, such as closure properties, and whether we can reduce classical automata decision problems to solving a system of CHCs. In addition, it will be interesting to identify more expressive MSO-D fragments that can be reduced to the emptiness of SDTAs.

Secondly, we believe that our results have applications to other areas in verification. We have conducted preliminary studies defining extensions of LTL with data (LTL-D) and, by using the framework developed in this paper and closure properties of SDTAs, obtained LTL-D model checking algorithms for (recursive) programs using scalar variables. Our approach is limited to finite runs only, so it will also be interesting to see how we can extend it to infinite trees and games.

Finally, (enumeration) trees can be used to encode executions of different classes of automata, such as concurrent pushdown automata or concurrent queue systems. It will be interesting to see if our approach can help lift the results of [31] to the corresponding class of concurrent programs.

References

1. Beyene, T., Chaudhuri, S., Popeea, C., Rybalchenko, A.: A constraint-based approach to solving games on infinite graphs. In: POPL'14, Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 221–233 (2014)
2. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday. LNCS, vol. 9300, pp. 24–51. Springer (2015)
3. Bodlaender, M.H.L., Hurkens, C.A., Kusters, V.J., Staals, F., Woeginger, G.J., Zantema, H.: Cinderella versus the wicked stepmother. In: IFIP International Conference on Theoretical Computer Science. pp. 57–71. Springer (2012)
4. Büchi, J.R.: Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly* **6**(1-6), 66–92 (1960)
5. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proc. 1960 International Congress for Logic, Methodology and Philosophy of Science. pp. 1–11. Stanford University Press (1962)
6. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. *J. Autom. Reason.* **64**(7), 1393–1418 (2020)
7. Colcombet, T., Ley, C., Puppis, G.: Logics with rigidly guarded data tests. *Log. Methods Comput. Sci.* **11**(3) (2015)
8. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: *Tree Automata Techniques and Applications* (2008), <https://hal.inria.fr/hal-03367725>
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, 3rd Edition. MIT Press (2009)
10. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A.: Symbolic register automata. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. LNCS, vol. 11561, pp. 3–21. Springer (2019)
11. D'Antoni, L., Veanes, M.: Monadic second-order logic on finite sequences. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. pp. 232–245. ACM (2017)
12. D'Antoni, L., Veanes, M.: Automata modulo theories. *Commun. ACM* **64**(5), 86–95 (2021)
13. Demri, S., D'Souza, D.: An automata-theoretic approach to constraint ltl. *Information and Computation* **205**(3), 380–415 (2007)
14. Doner, J.: Tree acceptors and some of their applications. *J. Comput. Syst. Sci.* **4**(5), 406–451 (1970)
15. Elgot, C.C.: Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society* **98**, 21–51 (1961)
16. van Emden, M.H., Kowalski, R.A.: The semantics of predicate logic as a programming language. *J. ACM* **23**(4), 733–742 (1976)
17. Farzan, A., Kincaid, Z.: Strategy synthesis for linear arithmetic games. In: *POPL. Proc. ACM Program. Lang.*, vol. 2, pp. 61:1–61:30 (2018)
18. Fedyukovich, G., Ahmad, M.B.S., Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs. In: *Proceedings of the 38th*

- ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 572–585. ACM (2017)
19. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021)
 20. Garoche, P., Kahsai, T., Thirioux, X.: Hierarchical state machines as modular horn clauses. In: Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016. EPTCS, vol. 219, pp. 15–28 (2016)
 21. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012. pp. 405–416. ACM (2012)
 22. Gurfinkel, A., Bjørner, N.: The science, art, and magic of constrained horn clauses. In: 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2019, Timisoara, Romania, September 4-7, 2019. pp. 6–10. IEEE (2019)
 23. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. LNCS, vol. 9206, pp. 343–361. Springer (2015)
 24. Hoder, K., Bjørner, N., de Moura, L.M.: μZ - an efficient engine for fixed points with constraints. In: Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. LNCS, vol. 6806, pp. 457–462. Springer (2011)
 25. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. LNCS, vol. 7436, pp. 247–251. Springer (2012)
 26. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. *J. Log. Program.* **19/20**, 503–581 (1994)
 27. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying java programs. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. LNCS, vol. 9779, pp. 352–358. Springer (2016)
 28. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, University of Aarhus (January 2001), notes Series NS-01-1. Available from <http://www.brics.dk/mona/>
 29. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 222–233. ACM (2011)
 30. Löding, C., Madhusudan, P., Peña, L.: Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.* **2**(POPL), 10:1–10:30 (2018)
 31. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 283–294. ACM (2011)
 32. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on

- Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011. pp. 611–622. ACM (2011)
33. Madhusudan, P., Qiu, X., Stefanescu, A.: Recursive proofs for inductive tree data-structures. In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. pp. 123–136. ACM (2012)
 34. Manna, Z., Zarba, C.G.: Combining decision procedures. In: Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers. LNCS, vol. 2757, pp. 381–422. Springer (2002)
 35. Matsushita, Y., Tsukada, T., Kobayashi, N.: Rusthorn: Chc-based verification for rust programs. In: Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. LNCS, vol. 12075, pp. 484–514. Springer (2020)
 36. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
 37. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. LNCS, vol. 4963, pp. 337–340. Springer (2008)
 38. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: Computer Science Logic. pp. 41–57. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
 39. Shimoda, T., Kobayashi, N., Sakayori, K., Sato, R.: Symbolic automatic relations and their applications to SMT and CHC solving. In: Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings. LNCS, vol. 12913, pp. 405–428. Springer (2021)
 40. Thatcher, J.W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Syst. Theory* **2**(1), 57–81 (1968)
 41. Thomas, W.: Automata on infinite objects. In: Van Leeuwen, J. (ed.) *Formal Models and Semantics*, pp. 133–191. Handbook of Theoretical Computer Science, Elsevier, Amsterdam (1990)
 42. Trakhtenbrot, B.A.: Finite automata and logic of monadic predicates. *Doklady Akademii Nauk SSSR* **149**, 326–329, In Russian (1961)
 43. Veanes, M., Bjørner, N.: Symbolic tree automata. *Information Processing Letters* **115**(3), 418–424 (2015)
 44. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: 2010 Third International Conference on Software Testing, Verification and Validation. pp. 498–507 (2010)