Economical Transformations of Structured Data (Extended Version)

Jyotirmoy V. Deshmukh, E. Allen Emerson, and Roopsha Samanta

Computer Engineering Research Centre and Dept. of Computer Sciences
The University of Texas at Austin, Austin TX 78712, USA
{deshmukh, emerson, roopsha}@cs.utexas.edu

Abstract. Reliability of large-scale hardware and software systems often depends on the correctness of the underlying structured data. Examples of structured data include heap-allocated linked data structures, files, and program states in software, and netlists and simulator states for modeling hardware. In this paper, we focus on automatically transforming structured data to make it satisfy certain properties of interest. In particular, we consider tree-like structured data (i.e. trees and lists), and finite state properties of such structures (e.g. acyclicity, sorted-ness). We address two separate problems: (1) given an input structure, transforming it in a minimal fashion to obtain a desired output structure, and (2) given an input property and an output property, synthesizing a (small) update program that transforms any input satisfying the input property to some desired output. Our approach uses automata-theoretic constructions to enable efficient reasoning. The practical utility of our approach is in its ability to scale to input structures of arbitrary sizes, as illustrated by our prototype tool.

1 Introduction

Large-scale hardware and software systems often manipulate various kinds of structured data. In hardware models, examples of structured data include net-lists and simulator states. In software systems, structured data manifests explicitly as heap-allocated linked data structures and files, or implicitly as program states, control-flow graphs and synchronization skeletons (for concurrent programs). High-level tasks in such systems often involve transforming structured data to satisfy certain properties of interest. For instance, certain net-lists can be modeled as directed graphs, and transformations for power and timing optimizations can be expressed as graph manipulation procedures [1, 2]. In software, for a rich class of programs [3, 4], the program state can be modeled as a tree. For such programs, static repair of the program can be reduced to static repair of program states, which in turn can be modeled as a tree transformation problem. In some ongoing programs like mail-servers and real-time life support systems, it is crucial to ensure data structure consistency for continued execution of the program [5, 6]. Here, the objective is to repair a given faulty data structure at run-time. Other examples include file recovery and repair, and compiler

optimizations. Typically, procedures for implementing such transformations are highly tailored to the application and are often developed manually.

In this paper, we present a uniform methodology to automatically generate transformations for a broad class of applications. In particular, we focus on structured data that can be modeled as trees (or lists). We target finite state properties for such structures, e.g., acyclicity, sorted-ness, reachability of specified data, bounds on out-degree (fan-out), arbitrary regular patterns on data, etc. We assume that a transformation can be expressed as an *update procedure* that iteratively traverses a given input structure and applies localized updates, i.e., updates restricted to small neighborhoods within the structure. We allow specification of a set of permissible localized updates and a cost associated with each update. The cost of an update procedure is the sum of the costs of the localized updates used by the procedure. Associating costs with updates enables us to choose between multiple update procedures. For instance, consider the problem of transforming a string so that it satisfies the regular property $(ab)^*$. Candidate outputs: ϵ, ab, $abab\ldots$, all satisfy this property. However, if the input is the string $abcabcabc$, then the update procedures that convert it to ϵ or ab potentially lose information, and may be less preferred.

Within the automatic transformation generation paradigm, we identify two separate sub-problems.

Sub-Problem 1. Given a single input structure, we wish to minimally modify it to *obtain an output structure* that satisfies properties of interest. Real-world instantiations of this problem include: repair of a faulty linked list to satisfy some invariant, repair of a file to satisfy its syntax, transformation of a given net-list so that it meets some design constraints, and repair of the synchronization skeleton of a concurrent program.

Sub-Problem 2. Given an input property and an output property, we wish to automatically *synthesize a (small) update program* that transforms any input satisfying the input property into some output satisfying the output property. Real-world instantiations of this problem include: synthesis of a program that repairs all faulty linked lists, synthesis of a program that can repair syntax errors in all faulty files, and repair of a program where the program states are trees.

We propose an automata-theoretic framework to address both of the above problems. We encode all possible update procedures composed of sequences of permissible updates in an *update automaton* $\mathcal{A}_{\mathcal{U}}$. We express the desired output properties as a *property automaton* $\mathcal{A}_{\varphi}$. We encode the given input structure (first case) or the given input property (second case) as an *input automaton*. We construct the synchronous product of these automata, denoted as $\mathcal{A}_{\otimes}$. Intuitively, $\mathcal{A}_{\otimes}$ captures all update procedures that transform the given inputs to desired outputs. Depending on the sub-problem, we use $\mathcal{A}_{\otimes}$ to yield a desired output structure or a small update program.

In the first case, we extract the output structure that satisfies the properties of interest and can be obtained using the least cost update procedure. In the second case, we extract a small set of update procedures that can transform

any input satisfying the input property to a desired output. This set of update procedures can be statically translated to a high-level program that implements the desired transformations.

Our key contributions in this paper are as follows: (1) We formulate an automata-theoretic framework to formally reason about automatic transformation of structured data, and present the necessary constructions and algorithms required. (2) We address two useful sub-problems that encompass numerous application scenarios within the above framework. (3) We provide the ability to choose transformations in a cost-aware manner. (4) We demonstrate that our framework naturally scales to transformations of input structures of arbitrary sizes.

The paper is organized as follows: We provide definitions and notation in Sec. 2. We address the first sub-problem outlined above in Sec. 3, and the second sub-problem in Sec. 4. Experimental results with our prototype tool are presented in Sec. 5. We end with a discussion of related work and extensions.

2 Preliminaries

In what follows, we introduce required terminology for our framework. We assume that all structures encountered are finite.

Rooted Digraphs, Trees. A rooted digraph (directed graph) is defined as a tuple (V, E), where V is a set of vertices with a designated vertex known as the *root vertex* r and $E \in V \times V$ is a set of edges. If $(v_1, v_2) \in E$, then we say that v_1 is a *parent* of v_2, and v_2 is a child of v_1. The root vertex has no parent. In the rest of the presentation we use the term digraph to mean a rooted digraph.

A tree t is a digraph (V, E), with the property that each vertex (except the root vertex) has a unique parent. We say that a vertex is *terminal* (denoted $\perp$) if it has no children. All non-terminal vertices have at least one child. A sub-tree st of a tree t is a rooted, connected digraph (V', E'), such that $V' \subseteq V$, and $E' \subseteq E$. We say that a tree is finite if every path in the tree ends in a terminal vertex. The height of a finite tree is the length of the longest path in the tree that begins at the root vertex.

Every finite digraph can be unwound into a finite tree[1]. For an input digraph t_i and an output tree t_o, we define a composite structure $t_c = t_i \circ t_o$, obtained by super-imposing t_o on the unwound t_i.

Tree Automata over Finite Trees. A non-deterministic finite state tree automaton $\mathcal{A}$ running over a finite tree t of maximum out-degree K, is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ where Σ is a finite, non-empty input alphabet labeling the nodes of t, Q is a finite, non-empty set of states, $\delta : Q \times \Sigma \to 2^{Q^K}$ is the non-deterministic transition function, $q_0 \in Q$ is the initial state, and F is a set of *final* states. The run ρ of $\mathcal{A}$ on a Σ-labeled t is an annotation of t with the states in Q, compatible with δ. We say that ρ is an accepting run if every path in ρ

[1] If the digraph has cycles, we unwind it only a bounded number of times.

contains a node labeled with a state from F. We say that a tree t is accepted by $\mathcal{A}$ if there exists an accepting run of $\mathcal{A}$ on t. The language of $\mathcal{A}$ (denoted $\mathcal{L}(\mathcal{A})$) is the set of all trees accepted by $\mathcal{A}$. Note that a tree automaton can be meaningfully defined to operate over an arbitrary digraph [7].

Window, Local Updates. A procedure that updates a rooted digraph can be typically visualized as traversing the digraph starting at the root vertex, and *iteratively* applying updates to each vertex. We wish to focus on procedures in which each such update is *local*, i.e., restricted to a small neighborhood of a vertex. To formally define a local update, we introduce the notion of a *window*.

For a given digraph/tree t, a window $w(v)$ is a rooted subgraph/sub-tree of t of some fixed *height* h, rooted at the vertex v. In effect, a window of height h rooted at v_1 contains all vertices v_2 s.t. $(v_1, v_2) \in E$ or $(v_1, v_2) \in E^2$ or $\ldots$ or $(v_1, v_2) \in E^h$. We denote a window by just w, when v is obvious from the context. We define the functions $root(w)$ and $ht(w)$ to denote the root node of w and the height of w, respectively. A terminal window is either a window with $\perp$ as its root node or a window with a blank symbol as its root node, denoted by $w\perp$ and $w\emptyset$, respectively.

An update primitive is a statement that could (a) change the data value of a vertex, (b) add or delete edges between vertices, (c) delete an existing vertex, or (d) add a new vertex. A *local update* u is any finite composition of update primitives that are constrained such that: (1) the vertices and edges that any constituent primitive refers to are (strictly) contained within the window w, and (2) any new vertex that is added is inserted as a child of a vertex in the original w. The first condition, for instance, disallows adding an edge to a vertex outside the window. The second condition precludes a possible long sequence of insertions where vertices keep getting added as children of newly added vertices. Thus, (2) ensures that the height of the window is only increased by a constant factor by any finite composition of update primitives. The attractive feature of local updates is that changes are confined to a local region, and do not affect other parts of the tree. In the rest of the presentation, we assume that all updates are local.

Permissible Updates, Update Procedure. We allow users to specify a *set of permissible updates*, $\mathcal{U} = \{u_1, u_2, \ldots, u_N, u_I\}$. Each update $u_n \in \mathcal{U}$ can be viewed as a code snippet composed of a combination of update primitives defined before. The update u_n maps an input window w_i to an output window w_o. Let h_i, h_o be the heights of input windows and output windows respectively. Let $W(h)$ denote the set of all windows of height h. Thus, every update u_n is the set $\{(w_{1i}, w_{1o}), \ldots, (w_{Mi}, w_{Mo})\}$, where each $w_{mi} \in W(h_i)$, and each $w_{mo} \in W(h_o)$. The *identity update* u_I maps a window to itself. Since the global objective is to modify a rooted digraph to obtain a tree, we impose certain basic conditions on each u_n that ensure that the modified window is a connected sub-tree. For instance, an update cannot cause w_i to become disconnected (i.e. no garbage), and cannot introduce sharing or cycles in w_o.

Let t_i be a given input digraph. An *update procedure* $\mathcal{P}$ iteratively visits each vertex v in t_i starting from the root node, choosing and applying some permissible update to $w(v)$. An update procedure thus yields an output tree t_o upon execution on an input digraph t_i. We write $t_o = \mathcal{P}(t_i)$.

Update Costs. To enable a quantitative comparison of updates and corresponding modified trees obtained by an iterative application of such updates, we define a cost function $c : \mathcal{U} \rightarrow \mathbb{N}$ that associates a positive integer cost $c(u_n)$ with each $u_n \in \mathcal{U}$. Thus, the cost of any (w_i, w_o) pair in u_n is $c(u_n)$. The cost of an update is typically a function of the system resources and complexity required to implement the low-level update. For instance, updates that involve a greater number of changes within a window are likely to be more expensive, updates involving insertion of nodes may be assigned higher costs, and so on. The identity update is assigned a cost of 0.

Let u_v be the short hand for the update chosen by procedure $\mathcal{P}$ for the window $w(v)$ rooted at the vertex v in the input digraph t_i. The total update cost for $\mathcal{P}$ (denoted as $C(\mathcal{P})$) is the sum of update costs over all vertices v in t_i, i.e., $C(\mathcal{P}) = \sum_{v \in t_i} c(u_v)$.

Regular Tree Properties. We say that a property ψ over trees is a regular tree property if there is an equivalent non-deterministic tree automaton (with final state acceptance) $\mathcal{A}_\psi$ such that every tree t satisfying ψ is accepted by $\mathcal{A}_\psi$, and conversely every tree accepted by $\mathcal{A}_\psi$ satisfies ψ. In the rest of the presentation, we assume that properties of interest are specified as tree automata of the form described above. Note that a regular tree property is a well-defined notion for digraphs, [7].

Windowed Trees/Digraphs. In order to facilitate the view of an update procedure as a sequence of local updates on windows, we define a a special representation called a *windowed tree/digraph* (denoted by $\widetilde{t}$) for a given tree/digraph t. Each vertex in $\widetilde{t}$ is a window $w(v)$ rooted at vertex v in t, and each edge $(w(v_1), w(v_2))$ in $\widetilde{t}$ corresponds to an edge (v_1, v_2) between the root vertices of $w(v_1)$ and $w(v_2)$. We illustrate this representation pictorially in Fig. 1 in lieu of a rigorous definition.

Special Case: Linear structures. Observe that when the desired output is a linear structure, the above definitions become considerably simpler. A digraph with a maximum out-degree of 1 is termed a *linear digraph*. We call a tree with maximum out-degree 1 a *list*. In the case of linear digraphs, a window is a linear sub-graph (or sub-list) of t of fixed height h. The definition of a finite state non-deterministic tree automaton with final state acceptance reduces to the well-known non-deterministic finite state *string* automaton (*nfa*), when $K = 1$. Thus, regular properties for lists are simply specified as regular expressions or in the equivalent formalism of *nfa*s.

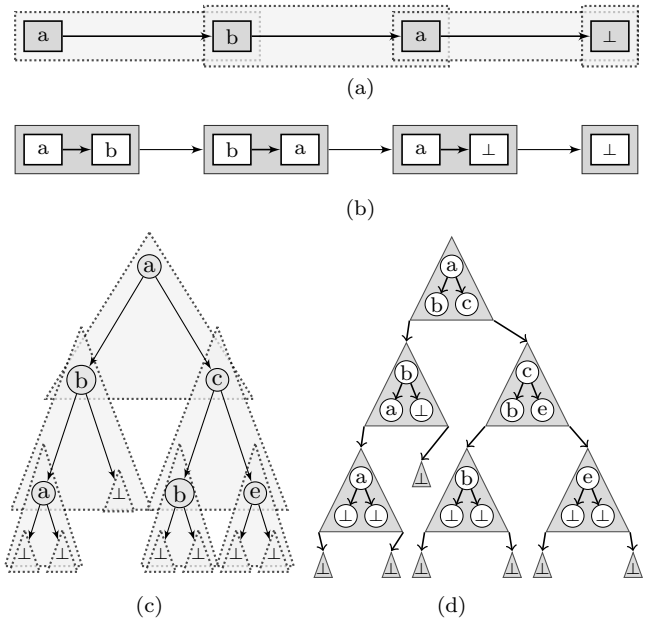


Fig. 1: Windowed Representation. (a) List l_i, (b) Windowed List $\tilde{l}_i$, (c) Tree t_i, (d) Windowed Tree $\widetilde{t_i}$

3 Transforming an Input Structure

In this section, we address *Sub-Problem* 1 from Sec. 1 in which the objective is to *minimally* modify a single input digraph to obtain a desired output tree. We present all the necessary steps in our methodology for achieving this goal.

3.1 Problem Definition.

The inputs to our framework are: a regular tree property φ, an input digraph t_i (that does not satisfy φ), and a set of permissible updates $\mathcal{U}$ with an associated cost function c. We wish to obtain an output tree t_o such that:

1. t_o satisfies φ,
2. there exists an update procedure $\mathcal{P}$ s.t. $\mathcal{P}(t_i) = t_o$, and,
3. for every update procedure $\mathcal{P}'$ which yields an output tree $t'_o = \mathcal{P}'(t_i)$ that satisfies φ, $C(\mathcal{P}) \leq C(\mathcal{P}')$.

In other words, we wish to obtain a suitable output tree t_o that can be obtained from t_i with the least cost[2].

3.2 Solution Outline.

The key steps in our approach are as follows:

1. Construct an *input automaton* $\mathcal{A}_I$ that encodes the windowed digraph $\widetilde{t_i}$ corresponding to t_i.
2. Construct an *update automaton* $\mathcal{A}_\mathcal{U}$ that encodes all possible update procedures using updates from $\mathcal{U}$.
3. Construct a *property automaton* $\mathcal{A}_\varphi$ that encodes the windowed trees $\widetilde{t_o}$ that correspond to output trees t_o that satisfy φ.
4. Construct a *query automaton* $\mathcal{A}_\otimes$, defined as the synchronous product $\mathcal{A}_I \otimes \mathcal{A}_\mathcal{U} \otimes \mathcal{A}_\varphi$.
5. Check if there exists *any* update procedure $\mathcal{P}$ s.t. $\mathcal{P}(\widetilde{t_i}) = \widetilde{t_o}$, and $\widetilde{t_o}$ is accepted by $\mathcal{A}_\varphi$. In effect, this check is equivalent to checking the non-emptiness of the language of $\mathcal{A}_\otimes$.
6. If the previous step succeeds, extract the least-cost update procedure, and the corresponding output tree t_o from $\mathcal{A}_\otimes$.

In the rest of this section, we use the following example to illustrate our methodology.

Example 1. We are given the list t_i from Fig. 1(a) as our input. The desired output property (φ) is sorted-ness of the list elements. If there are only two data values a, b allowed in the list, with $a < b$, we can express φ as the regular

[2] With an appropriate cost function c, one may expect the output t_o obtained from the least cost update procedure to correspond to a minimal perturbation of the original input t_i.

expression a^*b^*. We wish to obtain an output list t_o that satisfies φ, given the following set of updates over windows of height 2: $U = \{u_s, u_r, u_I\}$, where $u_s = \{(\boxed{\text{ba}}, \boxed{\text{ab}})\}$ swaps the list elements in the window $\boxed{\text{ba}}$, $u_r = \{(\boxed{\text{ab}}, \boxed{\text{aa}})\}$ sets both list elements to a in the window $\boxed{\text{ab}}$, and u_I is the identity update. The costs of the updates are: $c(u_s) = 1$, $c(u_r) = 10$ and $c_{u_I} = 0$.

3.3 Automata Constructions

We first discuss the construction of the automata outlined above. For simplicity, we explain each construction assuming that the heights of windows in input and output structures are the same, i.e., $h_i = h_o = h$. Note that this does not compromise the generality of our framework. When $h_i < h_o$, i.e., when there exist updates that insert nodes within a window, we can pad the relevant positions in the input window with a special placeholder symbol, say $-$, such that the size of the input window equals the size of the output window. Also, we can ensure that updates that delete nodes within a window never decrease the size of the window, by similarly marking all deleted nodes with another special symbol.

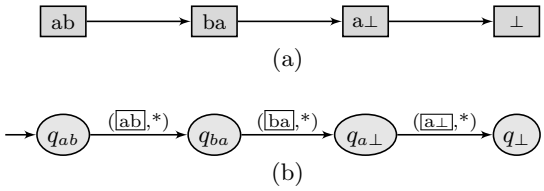


Fig. 2: (a) Windowed list from Fig. 1(b), (b) Input Automaton $\mathcal{A}_I$

Input Automaton, $\mathcal{A}_I$. Given an input digraph t_i, we encode it as a tree automaton that effectively accepts the corresponding windowed digraph $\widetilde{t_i}$. In particular, the *input automaton* $\mathcal{A}_I$ only accepts composite structures $\widetilde{t_c} = \widetilde{t_i} \circ \widetilde{t_o}$, where $\widetilde{t_i}$ is the input windowed digraph and $\widetilde{t_o}$ is some arbitrary output structure.

Let $\omega_0, \omega_1, \omega_2, \ldots$ denote the vertices of $\widetilde{t_i}$, where ω_0 is the root node. Additionally, let $w\perp$ and $w\emptyset$ be terminal vertices as defined in Sec. 2. If t_i has cycles, let us assume that each cycle has diameter less than or equal to the window-height h. In this case, we can observe each cycle within a window, termed a cycle-containing window. We denote cycle-containing windows where the root vertex is part of some cycle using a special notation: $w_1^{\circlearrowleft}, w_2^{\circlearrowleft}, \ldots$. Note that the root vertex of a window $w_j^{\circlearrowleft}$ may be a part of cycles along multiple successors. We denote the cycle along the k^{th} successor of $root(w_j^{\circlearrowleft})$ as $\circlearrowleft_j^k$.

We encode each (windowed) vertex in $\widetilde{t_i}$ into a state of $\mathcal{A}_I$ and each edge in $\widetilde{t_i}$ into the transition relation of $\mathcal{A}_I$. Let the states of $\mathcal{A}_I$ corresponding to vertices ω_j be q_{ω_j}, vertices $w_j^{\circlearrowleft}$ be $q_{\circlearrowleft_j}$, vertex $w\perp$ be $q_\perp$ and vertex $w\emptyset$ be $q_\emptyset$. We denote

the k^{th} successor of a vertex ω_j (or $w_j^{\circlearrowleft}$) in $\widetilde{t}_i$ as $succ(k, \omega_j)$ (or $succ(k, w_j^{\circlearrowleft})$). Note that if $root(w_j^{\circlearrowleft})$ is part of the cycle $\circlearrowleft_j^k$, then $succ(k, w_j^{\circlearrowleft})$ is some $w_m^{\circlearrowleft}$ such that $root(w_j^{\circlearrowleft})$ and $root(w_m^{\circlearrowleft})$ are successive vertices in the cycle $\circlearrowleft_j^k$. When $\mathcal{A}_I$ is in a state corresponding to vertex w_i in $\widetilde{t}_i$, it transitions to the K-tuple of states corresponding to the K successors of w_i in $\widetilde{t}_i$ upon reading the vertex $(w_i, *)$ of $\widetilde{t}_c$. If $\mathcal{A}_I$ reads a window with the special symbol $-$, it reconstructs the window without any occurences of the special symbol, and transitions accordingly. The accepting states of $\mathcal{A}_I$ depend on the structure of $\widetilde{t}_i$. If $\widetilde{t}_i$ is acyclic, $\mathcal{A}_I$ transitions to an accepting state upon reading the terminal window $w\perp$, and stays in the accepting state henceforth. If $\widetilde{t}_i$ has cycles, $\mathcal{A}_I$ transitions to a special accepting state, denoted as $q_{\circlearrowleft}$, the first time upon revisiting a vertex in a cycle. To enforce this, for each cycle, we focus on the vertex in the cycle that is the furthest distance from the root node of t_i. When $\mathcal{A}_I$ reads the corresponding windowed vertex $w_j^{\circlearrowleft}$, it transitions to $q_{\circlearrowleft}$ along the relevant successor.

Formally, $\mathcal{A}_I$ is the tuple $(\Sigma, Q, q_0, \delta, acc)$, where $\Sigma = W_i \times W_o$, $Q = \{q_{\omega_0}, q_{\omega_1}, \ldots, q_\perp, q_\emptyset, q_{\circlearrowleft}, q_{\circlearrowleft 1}, q_{\circlearrowleft 2}, \ldots, rej\}$, $q_0 = q_{\omega_0}$, $acc = \{q_\emptyset, q_{\circlearrowleft}\}$, and δ is described as follows:

$$\delta(q_{\omega_j}, (w_i, *)) = \begin{cases} (q_{succ(1,\omega_j)}, \ldots, q_{succ(K,\omega_j)}) & \text{if } \omega_j = w_i, \\ (rej, \ldots, rej) & \text{otherwise} \end{cases}$$

$$\delta(q_{\circlearrowleft_j}, (w_i, *)) = \begin{cases} (q_{succ(1,w_j^{\circlearrowleft})}, \ldots, q_{succ(K,w_j^{\circlearrowleft})}) & \text{if } w_j^{\circlearrowleft} = w_i, \\ (rej, \ldots, rej) & \text{otherwise} \end{cases}$$

$$\delta(q_\perp, (w_i, *)) = \begin{cases} (q_\emptyset, \ldots, q_\emptyset) & \text{if } w_i = w\perp, \\ (rej, \ldots, rej) & \text{otherwise} \end{cases}$$

$$\delta(q_\emptyset, (w_i, *)) = \begin{cases} (q_\emptyset, \ldots, q_\emptyset) & \text{if } w_i \text{ is blank} \\ (rej, \ldots, rej) & \text{otherwise} \end{cases}$$

$$\delta(q_{\circlearrowleft}, (w_i, *)) = \{(q_{\circlearrowleft}, \ldots, q_{\circlearrowleft})$$

$$\delta(rej, (w_i, *)) = (rej, \ldots, rej).$$

In the above definition of δ, if $succ(k, \omega_j)$ or $succ(k, w_j^{\circlearrowleft})$ is $w\perp$, the next state along the k^{th} successor is $q_\perp$. Similarly, if $succ(k, \omega_j)$ or $succ(k, w_j^{\circlearrowleft})$ is $w_m^{\circlearrowleft}$, the next state along the k^{th} successor is $q_{\circlearrowleft_m}$. Finally, if $root(w_j^{\circlearrowleft})$ is part of the cycle $\circlearrowleft_j^k$ and is the furthest distance from the root node of t_i amongst all vertices that are part of $\circlearrowleft_j^k$, then $succ(k, w_j^{\circlearrowleft}) = q_{\circlearrowleft}$. If t_i is a linear digraph, i.e. $K = 1$, upon reading (w_i, w_o), each state transitions to only the first state in the K-tuple of states shown above. See Fig. 2 for an illustration of $\mathcal{A}_I$ for Example 1 ($\mathcal{A}_I$ encodes $\widetilde{t}_i$ from Fig. 1(b)). For simplicity in the diagram, we do not show the rejecting or accepting states. Essentially, $q_\perp$ transitions to the final accepting state $q_\emptyset$ upon reading the symbol $(w\emptyset, *)$.

Update Automaton, $\mathcal{A}_\mathcal{U}$. An *update automaton* $\mathcal{A}_\mathcal{U}$ is an encoding of all iterative update procedures that transform a given input digraph t_i to a desired output tree t_o. $\mathcal{A}_\mathcal{U}$ operates on a composite structure $\widetilde{t}_c = \widetilde{t}_i \circ \widetilde{t}_o$, where every vertex of $\widetilde{t}_c$ is a pair of windows (w_i, w_o). The states of $\mathcal{A}_\mathcal{U}$ are designed to check the following: (1) is every input/output window (w_i, w_o) read in a state q *consistent* with those read in the previous state? (2) does (w_i, w_o) read in state q correspond to some valid update in $\mathcal{U}$? If the checks are successful, then the transition labeled with $(w_i, w_o)/u$ is enabled in state q. We illustrate the way in which $\mathcal{A}_\mathcal{U}$ encodes these checks in its transition

diagram with an example. For simplicity, consider the case where the input digraph is linear (i.e. the expected output is a list). Suppose,

$$q_1 \xrightarrow{(x_i,x_o)/u_1} q_2 \xrightarrow{(y_i,y_o)/u_2} q_3$$

is a pair of consecutive transitions in $\mathcal{A}_{\mathcal{U}}$. Since x_i and y_i are consecutive windows of the input digraph, recall that x_i and y_i have an overlapping portion. If x_i and y_i have height h, then the overlap consists of $h-1$ vertices.

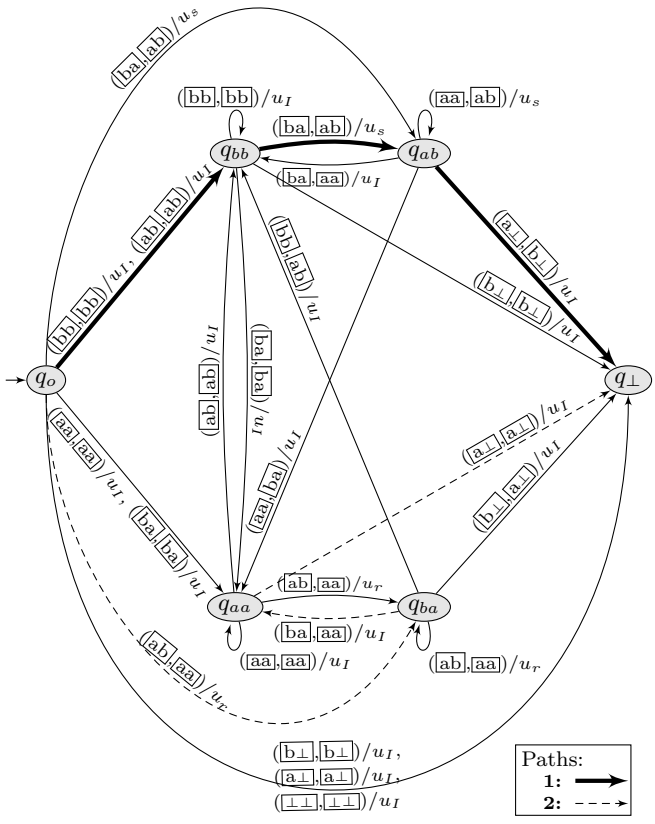


Fig. 3: Update Automaton $\mathcal{A}_{\mathcal{U}}$

When in state q_2, the first check tries to ascertain that the overlapping portions are identical; in the above example, $\mathcal{A}_{\mathcal{U}}$ can check for consistency by "remembering" the trailing $h-1$ vertices of x_i (denoted r_i) in q_2 and checking if it matches the first $h-1$ vertices in y_i (denoted $head(y_i)$). The second check in state q_2 tries to ascertain

if (y_i, y_o) corresponds to the update u_2. Note that when $\mathcal{A}_\mathcal{U}$ is in state q_1, $\mathcal{A}_\mathcal{U}$ applies the update u_1 to possibly change the overlapping portion of x_i and y_i. Thus, y_i is now changed to a modified window, say y_i', in which the first $h-1$ vertices of y_i (i.e. $head(y_i)$) are replaced by the trailing $h-1$ vertices of x_o (denoted r_o). Hence, instead of checking if $(y_i, y_o) \in u_2$, we check if $(y_i', y_o) \in u_2$. We write $y_i' = ((r_o \rightsquigarrow y_i)$ to emphasize how y_i' is obtained from y_i. To be able to do this check, $\mathcal{A}_\mathcal{U}$ has to also remember the trailing $h-1$ vertices of x_o (i.e. r_o) in state q_2. See Fig. 3 for an illustration of $\mathcal{A}_\mathcal{U}$ for Example 1. As before, we do not show rejecting states and accepting states for simplicity in the diagram. The state $q_\perp$ transitions to the accepting state $acc_\mathcal{U}$ upon seeing a (w_i, w_o) where both w_i, w_o are $w\perp$ or one is $w\emptyset$ while the other is $w\perp$.

For branching structures, where the windows are sub-trees of height h, there are K sub-trees of height $h-1$, each corresponding to an overlapping portion along one of the K successors. Each state of $\mathcal{A}_\mathcal{U}$ thus remembers a pair of windows (sub-trees) of height $h-1$ (denoted as r_{ki}, r_{ko} along the k^{th} successor), and $\mathcal{A}_\mathcal{U}$'s transitions mimic the two checks outlined above.

Let Q_Σ denote the set of all input-output pairs of windows of height $h-1$. States of $\mathcal{A}_\mathcal{U}$ have a one-to-one correspondence with elements of Q_Σ, and we use them interchangeably. Formally, $\mathcal{A}_\mathcal{U}$ is defined as the tuple $(\Sigma, Q, q_0, \delta, acc_\mathcal{U})$, where:

- $\Sigma = W_i \times W_o$, is a finite, non-empty set of updates,
- q_0 is a designated initial state,
- $Q = \{q_0, acc_\mathcal{U}, rej\} \cup Q_\Sigma$ is a finite, non-empty set of states,
- $\delta : Q \times \Sigma \rightarrow Q^k$ is a deterministic transition function, defined below[3],[4]:

$$\delta(q_0\ , (w_i, w_o)/u_n) = \begin{cases} (acc_\mathcal{U}, \ldots, acc_\mathcal{U}) & \text{if } w_i, w_o = w_\perp \text{ or } w\emptyset, \\ ((r_{1i}, r_{1o}), \ldots, (r_{Ki}, r_{Ko})) & \text{if } w_i, w_o \neq w_\perp \text{ or } w\emptyset \\ & \text{and } (w_i, w_o) \in u_n, \\ (rej, \ldots, rej) & \text{otherwise} \end{cases}$$

$$\delta(q_{r_i, r_o}, (w_i, w_o)/u_n) = \begin{cases} (acc_\mathcal{U}, \ldots, acc_\mathcal{U}) & \text{if } w_i, w_o = w_\perp \text{ or } w\emptyset, \\ ((r_{1i}, r_{1o}), \ldots, (r_{Ki}, r_{Ko})) & \text{if } w_i, w_o \neq w_\perp \text{ or } w\emptyset, \\ & head(w_i) = r_i \text{ and} \\ & ((r_o \rightsquigarrow w_i), w_o) \in u_n \\ (rej, \ldots, rej) & \text{otherwise} \end{cases}$$

$$\delta(acc_\mathcal{U}\ , (w_i, w_o)) = \begin{cases} (acc_\mathcal{U}, \ldots, acc_\mathcal{U}) & \text{if } (w_i, w_o) = (w\emptyset, w\emptyset), \\ (rej, \ldots, rej) & \text{otherwise.} \end{cases}$$

$$\delta(rej\ , (w_i, w_o)) = (rej, \ldots, rej)$$

- $acc_\mathcal{U}$ is the accepting state.

Property Automaton, $\mathcal{A}_\varphi$. We wish to define the *property automaton* $\mathcal{A}_\varphi$ to run on a composite structure $\widetilde{t_c} = \widetilde{t_i} \circ \widetilde{t_o}$, where $\widetilde{t_i}$ is an arbitrary input structure and $\widetilde{t_o}$ *contains* some desired output tree. Note that updates in successive windows may over-write

[3] We define transitions enabled on blank symbols, i.e. $w\emptyset$, to help reason about updates that change the size of the input structure.

[4] To each transition that is enabled on reading the symbol (w_i, w_o), we add a label u_n corresponding to the underlying update to help with the extraction of an update procedure later. We emphasize that these labels can be inferred from each transition, and hence, should not be viewed as additional symbols read by $\mathcal{A}_\mathcal{U}$.

the overlapping portions of the windows. In particular, if the height of a window is h, any vertex v (and its outgoing edges) may be over-written as many as $h-1$ times. As a result, the output component of $\widetilde{t}_c$, in general, does not directly correspond to the windowed representation of some output tree. Nevertheless, we observe that once a vertex is at the root of an output window it cannot be changed any further, and can exploit this to suitably construct an $\mathcal{A}_\varphi$ that accepts a $\widetilde{t}_o$ iff the *contained* t_o satisfies φ.

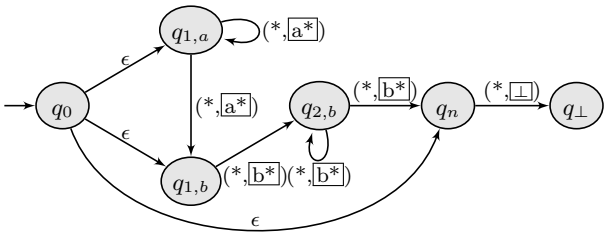
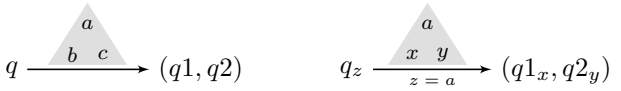


Fig. 4: Property Automaton $\mathcal{A}_\varphi$

We note that at any vertex, information about the sub-trees rooted at the sibling vertices (other children of a vertex's parent) is not available. To mitigate this, we use non-determinism. At every vertex, $\mathcal{A}_\varphi$ "guesses" the sub-tree (of height $h-1$) rooted at each of its K children. It then transitions to the K next states that in addition to the transition rules in φ, "remember" the corresponding guess. When $\mathcal{A}_\varphi$ reads a new output window, it first checks if it had guessed the sub-tree correctly when at the parent node, failing which it transitions to a reject state. For instance, in the example below, some φ specifies that upon reading the window as shown in state q, the expected transitions are to states $q1$ and $q2$ along the left and right successors of this window. To mimic this in $\mathcal{A}_\varphi$ we add transitions from all states q_z that have guessed the value of a correctly in the previous step (i.e. $z = a$), to states $q1$ and $q2$. States q_1 and q_2 in turn remember the next pair of guesses of $\mathcal{A}_\varphi$ for the root nodes of the child windows (i.e. x and y).



Rule in φ Matching Rule in $\mathcal{A}_\varphi$

Thus, every path along which $\mathcal{A}_\varphi$ guesses the next sub-tree correctly, and which reaches an accepting state according to the rules in φ is accepted. Note that for a given $\widetilde{t}_c$, the run of $\mathcal{A}_\varphi$ on $\widetilde{t}_c$ is deterministic, and $\mathcal{A}_\varphi$ accepts some $\widetilde{t}_o$ if and only if the contained output tree t_o satisfies φ. See Fig. 4 for an illustration of $\mathcal{A}_\varphi$ for *Example 1*. As before, we omit the accepting and rejecting states for simplicity. The state $q_\perp$ transitions to the accepting state $q_\emptyset$ upon reading any symbol $(*, w\emptyset)$.

We now formally define $\mathcal{A}_\varphi$. Recall that φ is itself specified as an automaton $(\Sigma_\varphi, Q_\varphi, \delta_\varphi, F_\varphi)$, where each component is defined in standard fashion. The transition upon reading a window w in φ (i.e. $\delta_\varphi(q, w)$), is a K-tuple of next states. We denote by $\delta_\varphi(q, w)[j]$ the j^{th} component of this tuple. Let $(ch_1, ch_2, \ldots, g_1, g_2, \ldots)$ denote elements of $W(h-1)$, i.e., windows of height $h-1$ (g denotes "guess", and ch denotes "check"). Let $Q_o = Q_\varphi \times W(h-1)$. This set, along with the initial and final states defines the set of states of $\mathcal{A}_\varphi$. Essentially, each state in Q_o is a pair (q, ch_j), where: $q \in Q_\varphi$ and ch_j is a "memorized" window in the state.

Recall from our informal discussion that in every state (q, ch) in Q_o, upon reading the symbol (w_i, w_o) $\mathcal{A}_\varphi$ does the following: (1) checks if ch matches the sub-tree of height $h-1$ rooted at w_o, rejects if not, (2) makes a guess for the sub-trees of height $h-1$ rooted at the successors of (w_i, w_o), (3) constructs a guessed window (say $w_{o,g}$) by splicing the root node of w_o with the K guessed sub-trees, (4) computes the first component of the next state according to $\delta_\varphi(q, w_{o,g})$, (5) sets the second component of the next state along the j^{th} successor in accordance with its guess for the j^{th} sub-tree of the current node.

Formally, let $G = \{(g_{11}, \ldots, g_{K1}), \ldots, (g_{1L}, \ldots, g_{KL})\}$ denote the set of the L possible, distinct, "guess-tuples". We use $G[\ell][j]$ to denote the j^{th} state in the ℓ^{th} guess in the set G. Let $w_{o,\ell}$ denote the window where $root(w_{o,\ell}) = root(w_o)$ and for each j, $1 \leq j \leq K$, $succ(j, w_{o,\ell}) = G[\ell][j]$. Thus, in state (q, ch), upon reading (w_i, w_o), $\mathcal{A}_\varphi$ non-deterministically transitions to a tuple of states $\{(q'_1, ch'_{1,\ell}), \ldots, (q'_K, ch'_{K,\ell})\}$, where: $q'_j = \delta_\varphi(q, w_{o,\ell})[j]$, and, $ch'_{j,\ell} = G[\ell][j]$.

We use $\Delta_{ij}(q, w_o)$ as short-hand to denote the next-state $(\delta_\varphi(q, w_{o,\ell})[j], G[\ell][j])$. Finally, $\mathcal{A}_\varphi$ is defined as the tuple $(\Sigma, Q, q_0, \delta, q_\emptyset)$, where:

- $\Sigma = W_i \times W_o$, is a finite, non-empty set of updates,
- q_0 is a designated initial state,
- $Q = \{q_0, rej\} \cup Q_o$ is a finite, non-empty set of states,
- $\delta : Q \times \Sigma \to 2^{Q^k}$ is a non-deterministic transition function, defined below:

$$\delta(q_0\ , (w_i, w_o)) = \begin{cases} \{(q_\emptyset, \ldots, q_\emptyset)\} & \text{if } w_o = w_\perp \text{ or } w\emptyset, \\ \{(\Delta_{11}(q_0, w_o), \ldots, \Delta_{K1}(q_0, w_o)), & \\ \ldots\ , & \\ (\Delta_{1L}(q_0, w_o) \ldots, \Delta_{KL}(q_0, w_o))\} & \text{otherwise} \end{cases}$$

$$\delta((q, ch)\ , (w_i, w_o)) = \begin{cases} \{(q_\emptyset, \ldots, q_\emptyset)\} & \text{if } w_o = w_\perp \text{ or } w\emptyset, \\ \{(\Delta_{11}(q_0, w_o), \ldots, \Delta_{K1}(q_0, w_o)), & \\ \ldots\ , & \\ (\Delta_{1L}(q_0, w_o) \ldots, \Delta_{KL}(q_0, w_o))\} & \text{if } head(w_o) = ch, \\ \{(rej, \ldots, rej)\} & \text{otherwise} \end{cases}$$

$$\delta(rej\ , (w_i, w_o)) = \{(rej, \ldots, rej)\}$$

- $q_\emptyset$ is the accepting state.

Query automaton, $\mathcal{A}_\otimes$. We construct a query automaton $\mathcal{A}_\otimes$ as the synchronous product $\mathcal{A}_I \otimes \mathcal{A}_\mathcal{U} \otimes \mathcal{A}_\varphi$. Thus, $\mathcal{A}_\otimes$ accepts a composite structure $\widetilde{t}_c = \widetilde{t}_i \circ \widetilde{t}_o$ iff $(\widetilde{t}_i, *)$ is accepted by $\mathcal{A}_I$, $(*, \widetilde{t}_o)$ is accepted by $\mathcal{A}_\varphi$ and $(\widetilde{t}_i, \widetilde{t}_o)$ is accepted by $\mathcal{A}_\mathcal{U}$. In other words, $\mathcal{A}_\otimes$ automaton encodes all windowed output trees which can be obtained from the given windowed digraph $\widetilde{t}_i$ using an update procedure based on updates from $\mathcal{U}$ and which satisfy φ.

Handling Cycles of Arbitrary Diameter. To encode an input digraph with cycles of diameter greater than the window-height, we can design an input automaton that has Büchi acceptance. Essentially, every state of the input automaton that corresponds to a vertex in the windowed input digraph is made accepting, and final states are made into "trap" states by adding self-transitions. Thus, the automaton accepts even if it gets "stuck" in a cycle, by virtue of seeing an accepting state infinitely often. Note that the Büchi condition only makes sense if the input digraph is an *infinite* tree; when the input digraph has cycles, the infinite unwinding of each cycle is a desired infinite tree, thus enabling the use of Büchi acceptance.

However, defining the input automaton to accept infinite trees is not enough; we would require modifications to the update and the property automata as well. Without going into specifics, at a high-level, a complication arises in product construction as the update and property automata are defined to run over finite trees with final state acceptance, while the input automaton is defined to run over infinite trees with Büchi acceptance. This can be resolved by defining the update and property automata to accept "pseudo-trees", i.e., trees that are cycle-free except for the leaf-nodes that have self-loops. We can show that if the product automaton accepts a composite structure that is the superposition of input digraph with cycles and a pseudo-tree, then there is a corresponding finite tree (obtained by erasing self-loops on the leaf nodes) that is the desired output. We avoid the full formal treatment of this case as it requires significant additional terminology and notation.

3.4 Update Extraction

Our construction of $\mathcal{A}_\otimes$ ensures that if $\mathcal{L}(\mathcal{A}_\otimes)$ is empty, then there is no update procedure that can transform the given input tree t_i to a desired output. Hence, the first step is to check $\mathcal{A}_\otimes$ for non-emptiness, failing which our update attempt terminates, reporting failure to modify t_i using updates from $\mathcal{U}$. If $\mathcal{L}(\mathcal{A}_\otimes)$ is non-empty, then each accepting run of $\mathcal{A}_\otimes$ is a pair $(\widetilde{t_i}, \widetilde{t_o})$ such that $\widetilde{t_o}$ is an acceptable transformation of $\widetilde{t_i}$ under the given $\mathcal{U}$. In our current framework, we choose the transformation that minimizes the total update cost using Algo. 1, which is an adaptation of Knuth's extension of Dijkstra's shortest path algorithm [8] to our framework.

The run of $\mathcal{A}_\otimes$ on a Σ-labeled tree $\widetilde{t_c}$ is an annotated tree $\rho(\widetilde{t_c})$ in which the root vertex of $\widetilde{t_c}$ is labeled with the initial state q_0, and each successive vertex σ is annotated with a state q compatible with the transitions in $\mathcal{A}_\otimes$. We denote a vertex σ labeled with state q as the labeled vertex (q, σ). The run $\rho(\widetilde{t_c})$ is an accepting run for $\mathcal{A}_\otimes$ if each path in $\rho(\widetilde{t_c})$ terminates in a vertex labeled with some final state, and the corresponding Σ-labeled tree is termed as a *witness* to the non-emptiness of $\mathcal{A}_\otimes$. A sub-tree of an accepting run rooted at (q, σ) is denoted by $\widetilde{t}_{(q,\sigma)}$, and is termed as a *witness sub-tree.* We say that the state labeling the root of $\widetilde{t}_{(q,\sigma)}$, i.e. q, has the *witness property.* Note that $\mathcal{A}_\otimes$ is non-empty iff q_0 labels the root vertex of some witness sub-tree.

Recall that a transition in $\mathcal{A}_\mathcal{U}$ from state q which is enabled on symbol σ is labeled σ/u and is associated with a cost $c(u)$ corresponding to the update u. Thus each labeled vertex, (q, σ), in $\rho(\widetilde{t_c})$ has an underlying update u, and can be assigned a cost $c(q, \sigma) = c(u)$. The cumulative update cost for a sub-tree $\widetilde{t}_{(q,\sigma)}$ is the sum of $c(q, \sigma)$ and the cumulative update costs for the sub-trees rooted at each successor of (q, σ). The update cost for the vertices labeled with final states is set to 0.

Our algorithm traverses $\mathcal{A}_\otimes$ starting from the final states, and maintains a set D of (roots of) witness sub-trees for which the minimum cumulative update cost has been

Algorithm 1: Extract Repair From Query Automaton

Input: Query automaton $\mathcal{A}_{\otimes}$
Output: Output tree $\widetilde{t}_o$ with least update cost

1 **begin**
2 **foreach** q *in* F **do**
3 $c_{min}(q)$:= 0
4 D := F
   /* *Obtain optimal witness* $\widetilde{t}_c$ */
5 **while** $(q_0 \notin D)$ **do**
6 **foreach** q *in* D **do**
7 $pre(q)$:= $\emptyset$
8 **foreach** $((q'$ *predecessor of* $q) \wedge (q' \notin D))$ **do**
9 **if** $((\delta(q', \sigma') = (q_1, \ldots, q_K)) \wedge (\forall k \in [1, K] : q_k \in D))$ **then**
10 $pre(q)$:= $pre(q) \cup \{(q', \sigma')\}$
11 **foreach** q'*such that* $(q', \sigma') \in pre(q)$ **do**
   /* $\delta(q', \sigma') = (q_{\sigma' 1}, \ldots, q_{\sigma' K})$ */
12 $c_m(q')$:= $\texttt{min}_{\sigma'} \left(c(q', \sigma') + \sum_{k=1}^{K} c_{min}(q_{\sigma' k}) \right)$
13 $\sigma_m(q')$:= σ' for least $c_m(q')$
14 q_{min} := q' with least $c_m(q')$
15 $c_{min}(q_{min})$:= $c_m(q_{min})$; $\sigma_{min}(q_{min})$:= $\sigma_m(q_{min})$
16 D := $D \cup \{q_{min}\}$
   /* *Breadth-first traversal for extracting* $\widetilde{t}_o = (\widetilde{V}_o, \widetilde{E}_o)$ */
   /* *Each* $\sigma_{min}(q) = (w_{i_{min}}(q), w_{o_{min}}(q))$. */
17 $\widetilde{V}_o$:= $w_{o_{min}}(q_0)$, $\widetilde{E}_o$:= $\emptyset$
18 stack := $\emptyset$, push q_0 on stack
19 **while** $(stack \neq \emptyset)$ **do**
20 q = top of stack, σ := $\sigma_{min}(q)$
21 pop stack
22 **if** $q \notin F$ **then**
23 $\widetilde{V}_o$:= $\widetilde{V}_o \cup \{w_{o_{min}}(q)\}$
   /* $\delta(q, \sigma) = (q_1, \ldots, q_K)$ */
24 **foreach** q_k *in* $\delta(q, \sigma)$ **do**
25 push q_k on stack
26 $\widetilde{E}_o$:= $\widetilde{E}_o \cup (w_{o_{min}}(q), w_{o_{min}}(q_k))$
27 **end**

computed, along with the corresponding minimum cost. In each iteration, it traverses every transition into a state within D *backwards*, trying to discover new states q' which satisfy the witness property (Line 10). For all such q', it computes the following (Lines 12-13):

– $\sigma_m(q')$: the transition symbol corresponding to the witness sub-tree $\widetilde{t}_{(q',\sigma_m)}$ with the least cumulative update cost among all witness sub-trees rooted at q'.
– $c_m(q')$: the minimum cumulative update cost for $\widetilde{t}_{(q',\sigma_m)}$.

It then inspects all such newly obtained states q', and determines the state q_{min} that has the least $c_m(q')$, records the values of $\sigma_{min}(q_{min})$ and $c_{min}(q_{min})$ (Line 15), and adds q_{min} to D (Line 16). The algorithm stops computing D once the root node q_0 is added to D. Finally, it extracts the desired composite tree $\widetilde{t}_c$ by traversing the states q in D in a breadth-first fashion, starting at q_0, using the information recorded in $\sigma_{min}(q)$ along the way. It simultaneously obtains the windowed output tree $\widetilde{t}_o$ by eliding the input-component of $\widetilde{t}_c$ for each σ_{min}. Once we obtain $\widetilde{t}_o$, we can obtain t_o by linking the root vertices of the windows corresponding to the vertices of $\widetilde{t}_o$ using the edges of $\widetilde{t}_o$[5].

Example 2. Consider $\mathcal{A}_\otimes$ for *Example 1*, constructed from $\mathcal{A}_I$, $\mathcal{A}_\mathcal{U}$ and $\mathcal{A}_\varphi$ from Figs. 2,3,4, resp. The paths in $\mathcal{A}_\mathcal{U}$ corresponding to the witnesses of $\mathcal{A}_\otimes$ are highlighted in Fig. 3. Path 2 corresponds to the update procedure $u_r; u_I; u_I;$ of cumulative cost 10, and yields the output list $a \rightarrow a \rightarrow a \rightarrow \perp$. Path 1 corresponds to the update procedure $u_I; u_s; u_I;$ of cumulative cost 1, and yields the output list $a \rightarrow a \rightarrow b \rightarrow \perp$. Algo. 1 chooses Path 1 as the least cost witness and yields the list $a \rightarrow a \rightarrow b \rightarrow \perp$ as the desired output.

Lemma 1 (Completeness). *If there exists an output tree $\widetilde{t}_o$ such that $\widetilde{t}_i$ can be transformed to $\widetilde{t}_o$ using some sequence of updates from $\mathcal{U}$, and the corresponding $t_o \models \varphi$, then Algo. 1 finds it.*

4 Transforming a Class of Input Structures

In this section, we address *Sub-Problem* 2 from Sec. 1. Here, the objective is to obtain an *update program* to transform every input digraph from a given class to some output tree that satisfies the output property. We assume that the class of input structures is specified as some regular tree property, termed the *input property*.

In general, automatically synthesized programs should be comparable in size and performance to manually written code, without compromising on their readability by a human. We hope to make appropriate choices in our formal problem definition and in each step of our solution to achieve this implicit goal. For simplicity, we restrict our attention to linear structures.

Problem Definition. The inputs to our framework are: regular expressions $\imath$ and φ that respectively specify a given input class of linear digraphs and a desired class of output lists, a set of permissible updates $\mathcal{U}$, and a cost function c that assigns costs to updates in $\mathcal{U}$. We define an *update program* $\mathbf{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_Z\}$, as a collection of update procedures $\mathcal{P}_1, \ldots, \mathcal{P}_Z$. We use $|\mathbf{P}|$ to denote the number of update procedures in $\mathbf{P}$. The cost of an update program $C(\mathbf{P}) = \sum_z C(\mathcal{P}_z)$, is the sum of costs of its constituent procedures. We wish to obtain an update program $\mathbf{P}$ such that:

1. for each linear digraph t_i accepted by $\mathcal{A}_\imath$, there exists an update procedure $\mathcal{P}_z \in \mathbf{P}$ such that $\mathcal{P}_z(t_i) = t_o$, and t_o satisfies φ,

[5] Note that for certain $\mathcal{U}$, $\mathcal{A}_\otimes$ may accept an update procedure that transforms an input to an empty output. However, the cumulative cost of such an update procedure can prevent it from being selected as the least-cost procedure.

2. for each $\mathbf{P}'$ that satisfies (1), $|\mathbf{P}'| \geq |\mathbf{P}|$,
3. for each $\mathbf{P}'$ that satisfies (1), and for which $|\mathbf{P}'| = |\mathbf{P}|$, $C(\mathbf{P}) \leq C(\mathbf{P}')$.

Thus, we wish to obtain an update program consisting of a *minimal* set of update procedures that can transform any linear digraph satisfying $\imath$ to a list satisfying φ. This minimal program consists of the least number of update procedures among all such programs, *and*, has the least cost among all programs with the same number of update procedures.

Solution Outline. In what follows, we present the necessary steps required to adapt our framework to address the above problem. We can proceed as before by constructing suitable automata to encode the input property, all possible update procedures and the desired output property. We then check the existence of an update program as before. Note that the product automaton, in general, contains (possibly multiple) update programs that we desire. However, to realize our goal of minimizing the size and cost of the update program, we need to be able to identify a suitable sub-automaton of the product.

Automata Constructions. As before, all automata are defined to run over a composite linear structure $\widetilde{t_c} = \widetilde{t_i} \circ \widetilde{t_o}$. The update automaton, $\mathcal{A}_\mathcal{U}$, and the property automaton $\mathcal{A}_\varphi$, for encoding output lists can be defined as in Sec. 3. Here, the input automaton $\mathcal{A}_\imath$ captures the input property $\imath$. A query automaton $\mathcal{A}_\otimes$, defined as the synchronous product $\mathcal{A}_\imath \otimes \mathcal{A}_\mathcal{U} \otimes \mathcal{A}_\varphi$, encodes all update procedures that can collectively transform linear digraphs satisfying $\imath$ to output lists satisfying φ.

Before we proceed, we clarify certain notions that will help reason over these automata. A path from the initial to a final state of an automaton may contain cycles, which in turn could be unrolled finitely many times to obtain a large set of accepting paths. However, all such accepting paths can be represented by a smallest *irreducible accepting path*. It is easy to see that there are only a finite number of such irreducible paths in a given finite state automaton. Each edge of an accepting path in $\mathcal{A}_\imath$ is labeled with some $(w_i, *)$ [6]. Thus, each irreducible accepting path in $\mathcal{A}_\imath$ corresponds to a finite-length linear digraph $\widetilde{t_i}$. We use such a path and its corresponding input structure interchangeably. Similarly, each irreducible accepting path of $\mathcal{A}_\otimes$ corresponds to a tuple $(\widetilde{t_i}, \widetilde{t_o})/\mathcal{P}$, where the update procedure $\mathcal{P}$ transforms the input $\widetilde{t_i}$ to a suitable output $\widetilde{t_o}$. We can also use an irreducible accepting path in $\mathcal{A}_\otimes$ interchangeably with its corresponding tuple.

Extraction of Update Program. As before, we first need to check if $\mathcal{L}(\mathcal{A}_\otimes)$ is non-empty. If $\mathcal{L}(\mathcal{A}_\otimes)$ is empty, our update attempt would terminate, stating that no input in the given class of inputs t_i can be modified suitably using the given set of updates $\mathcal{U}$. If $\mathcal{L}(\mathcal{A}_\otimes)$ is non-empty, we could extract the minimal update program in the following three steps:

Step 1. The goal in Step 1 is to check if for each irreducible accepting path $\widetilde{t_i}$ in $\mathcal{A}_\imath$, there exists some irreducible accepting path $(\widetilde{t_i}, \widetilde{t_o})/\mathcal{P}$ in $\mathcal{A}_\otimes$. In other words, we wish to check if each input that satisfies $\imath$ can be transformed suitably by some permissible update procedure. We can do this by constructing the synchronous product $\mathcal{A}_\imath \otimes \neg\mathcal{A}_\otimes$, and checking its language for non-emptiness. If the language is non-empty, then there

[6] In reality, this transition may correspond to multiple transitions labeled with $(w_i, w_{o1}), (w_i, w_{o1})$ etc., all of which are enabled from the same state, and transition into the same next state.

exists some accepting path $\widetilde{t_i}$ in $\mathcal{A}_\imath$ for which there is no corresponding irreducible accepting path $(\widetilde{t_i}, \widetilde{t_o})/\mathcal{P}$ in $\mathcal{A}_\otimes$. Our update attempt would terminate in this case, stating that there is no update program that can transform the entire class of inputs suitably. If the language is empty, we can proceed to Step 2.

Step 2. Multiple irreducible accepting paths in $\mathcal{A}_\otimes$ may be labeled with the same update procedure $\mathcal{P}_z$. In other words, we can associate with $\mathcal{P}_z$ a set of inputs, each of which labels some irreducible accepting path in $\mathcal{A}_\imath$, and is successfully transformed by $\mathcal{P}_z$. We denote this set as $\mathcal{I}_z$. Since there are only a finite number of irreducible accepting paths in $\mathcal{A}_\otimes$, there can only be a finite number, say Z, of distinct update procedures labeling these paths in $\mathcal{A}_\otimes$. Thus, there are Z sets $\mathcal{I}_1, \ldots, \mathcal{I}_Z$, of (possibly overlapping) sets of inputs which are successfully transformed by each of the Z update procedures captured in $\mathcal{A}_\otimes$. To obtain these sets, we can traverse each distinct irreducible accepting path in $\mathcal{A}_\otimes$, clustering paths labeled with the same update procedure $\mathcal{P}_z$ and extracting the desired $\mathcal{I}_z$ from each cluster.

The goal in Step 2 is to find the *smallest set covers*[7] for $\mathcal{I}_1, \ldots, \mathcal{I}_Z$. A smallest set cover represents the least number of distinct update procedures that can transform all possible inputs to suitable outputs. We can compute the smallest set covers by iteratively examining all possible subsets of the set $\{\mathcal{I}_1, \ldots, \mathcal{I}_Z\}$ of cardinality 1, 2, 3, and so on, until we obtain a set cover. We remark that computing the smallest set cover is a well-studied NP-hard problem [9], and several approximation algorithms have been proposed over the years [10]. Thus, it is possible to substitute our naive algorithm with some efficient approximation algorithm from the literature.

If there are more than one set covers of the smallest size, we can proceed to Step 3. If not, the update attempt should terminate successfully, and output the smallest set cover as the minimal update program.

Step 3. The cost of a set cover is simply the sum of the costs of the update procedures comprising it. Given a set of smallest set covers, each with an associated cost, we can obtain the set cover with the minimum cost in this final step. We can then output this smallest, least-cost set cover as the minimal update program.

Finally, we remark that as each update procedure in the update program consists of a sequence of updates (as in *Ex. 2*), it can be coded using high-level programming constructs. For instance, repeated sequences of updates are compiled to loops, and choices between update procedures are compiled to guards.

Reasoning over Irreducible Accepting Paths.

We formalize the notion that it is enough to reason over irreducible accepting paths in the following lemma:

Lemma 2. *If there is an update program* $\mathbf{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_Z\}$*, where each update procedure* $\mathcal{P}_z$ *transforms an input corresponding to an irreducible accepting path in* $\mathcal{A}_\imath$*, then there is an update program* $\mathbf{P}' = \{\mathcal{P}'_1, \ldots, \mathcal{P}'_Z\}$ *that transforms each input accepted by* $\mathcal{A}_I$*.*

Proof Outline. The steps of the proof are as follows:

[7] Recall that a set cover for sets $T_1, T_2, \ldots T_J$ is a subset C of the set $\{T_1, T_2, \ldots T_J\}$ such that $C = \cup_j T_j$. A smallest set cover is one with the least number of distinct sets in it.

(a) Define an equivalence relation $\equiv_\pi$ over strings - strings are equivalent if they are accepted by the same path π (modulo repetitions) in the automaton. Show that for each equivalence class there is a unique smallest string equivalent to all other strings. Show that an irreducible accepting path is the same as the unique smallest string for each such equivalence class.

(b) Each procedure can be viewed as a string of updates. Show that if inputs i_1 and i_2 are $\equiv_\pi$-equivalent, $\mathcal{P}_1$ transforms i_1 to a desired output, and $\mathcal{P}_2$ transforms i_2 to a desired output, then $\mathcal{P}_1$ is $\equiv_\pi$-equivalent to $\mathcal{P}_2$.

(c) Combine (a) and (b) to show that given an input i, the procedure $\mathcal{P}'_z$ to transform i is obtained by: obtain the irreducible accepting path $\widetilde{t_i}$ to which i is $\equiv_\pi$-equivalent; obtaining the corresponding $\mathcal{P}_z$ for $\widetilde{t_i}$, and repeating the updates in $\mathcal{P}_z$ corresponding to the repeated symbols in i.

In simple terms, update procedures for the irreducible accepting paths can be thought of as straight line code. Repeated parts of paths correspond to adding loops in the update procedure for those symbols that repeat (i.e. loops in the product automaton).

Table 1: Experimental Results

Input Size (Num. nodes)	Window Size	Property	$\mathcal{A}_\mathcal{U}$ Time (ms) [a]	$\mathcal{A}_\mathcal{U}$ Mem. (MB)	$\mathcal{A}_\otimes$ Time (secs)	$\mathcal{A}_\otimes$ Mem. (MB)	Extraction Time (secs)	Selected Updates
10	2	Sorted-ness	35	0.3	0.01	0.09	0.008	`swap`
300	2	Sorted-ness	32	0.3	0.18	0.8	0.746	`swap`
1500	2	Sorted-ness	37	0.3	0.86	6.4	2.727	`swap`
10	3	Sorted-ness	561	0.9	0.21	6.8	0.011	`swap2`
300	3	Sorted-ness	575	0.9	4.94	14.1	0.390	`swap2`
1500	3	Sorted-ness	530	0.9	12.58	22.8	3.926	`swap2`
10	2	Acyclicity	50	0.3	0.01	0.1	0.010	`removeEdge`
10	3	Property A	167	0.3	0.02	1.3	0.010	`u1`, `u2`

[a] All experiments were performed on a Linux machine with an AMD Athlon 64x2 2.2 GHz processor, and 6GB RAM.

5 Experimental Results

We have implemented the first sub-problem discussed in this paper as a prototype tool in `Java`. For now, we have focused on the problem instance where the desired output is a list. Some of the key results obtained with this tool are shown in Table 1. One of the highlights of our tool is its ability to scale to inputs of large sizes. We can combine the two main steps in the solution: (a) checking non-emptiness of $\mathcal{A}_\otimes$ and (b) extracting an output tree from $\mathcal{A}_\otimes$ into a single step, as the automaton $\mathcal{A}_\otimes$ is empty if the initial

state $q0$ is not an element of the set D (in Algo. 1). The complexity of Algo. 1 is linear in the size of $\mathcal{A}_\otimes$, which in turn is linear in the size of the input, the update automaton and the property. As we can see from Table 1, the time taken and memory consumption for $\mathcal{A}_\mathcal{U}$ construction is a static cost for a given $\mathcal{U}$, independent of the input length. The times taken for constructing $\mathcal{A}_\otimes$, checking it for emptiness and extracting the output scale linearly with the input size - a desirable characteristic. The right-most column indicates which of the updates from the library of updates were chosen by our tool to obtain the least cost output list. The updates `swap`, `swap2` essentially swap the data values of vertices if they do not appear sorted, and `removeEdge` deletes an edge from a cycle. In the last example, we check for a data-centric property: "every vertex labeled b is followed by exactly two vertices, each labeled a". To enforce this property, we use two updates: `u1` that removes undesirable trailing vertices, and `u2` that changes one label from b to a.

Our tool may report that an input cannot be repaired with a given set of permissible updates $\mathcal{U}$. Since $\mathcal{U}$ fixes a window size, it is possible to repair the same input by considering a larger window size, and correspondingly, a larger set of updates $\mathcal{U}$. Thus, our tool could be used as a module in a loop that tries to find the cheapest fix by increasing the window size in each iteration.

6 Discussion

Related Work. There has been work on computing the minimum tree-edit distance between a *given pair* of labeled trees, and the corresponding edit script [11]. In contrast, in our first sub-problem, we focus on computing an *output tree* satisfying some regular tree property that can be obtained from an input digraph by a permissible set of updates.

Approaches for program repair based on repairing Boolean abstractions of concrete programs have been explored in [12, 13]. However, for programs with heap-allocated data structures, Boolean abstractions are often too imprecise, and hence ineffective for repairing such programs.

In [14, 5, 6], the authors focus on the dynamic repair of a single faulty data structure to ensure certain consistency properties. Such properties are specified in a suitable relational calculus, or are provided in the form of executable specifications (known as `repOk` methods) built into the data structure. A common theme in these approaches is to model the failure of a consistency property as a constraint that can be solved with a theorem prover or a specific constraint solver. The solution to the first sub-problem in this paper addresses a large subclass of data structures and consistency conditions, and can be used for dynamic repair.

The problems considered in this paper can be interpreted as an instance of parameterized reasoning [15]. In [16, 17], the authors develop an automata-theoretic framework for parameterized *verification* of data structure methods. While we use a similar framework in this paper, we focus on parameterized *transformations* of structured data and *synthesis* of methods.

Extensions and Future Work. The ultimate objective of synthesizing an update program as outlined in Sec. 4 is to obtain a compact program, written using constructs from high-level programming languages. We wish to explore different ways to steer our update programs towards this goal. For instance, it may be preferable to use update procedures with fewer number of distinct updates in them, and this could be incorporated into the

current framework by suitably changing the cost computation of an update procedure. Further, our framework can be extended to synthesize transformations involving r passes over the input structure ($\mathcal{A}^r_{\mathcal{U}}$).

References

1. Sait, S.M., Youssef, H.: VLSI Physical Design Automation. World Scientific (1999)
2. Hachtel, G.D., Somenzi, F.: Logic Synthesis and Verification Algorithms. Springer (1996)
3. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying programs with dynamic 1-selector-linked structures in regular model checking. In: Proc. of TACAS. (2005) 13–29
4. Bouajjani, A., Habermehl, P., Rogalewicz, A.: Abstract regular tree model checking of complex dynamic data structures. In: Proc. of SAS. (2006) 52–70
5. Elkarablieh, B., Khurshid, S.: Juzi: A tool for repairing complex data structures. In: Proc. of ICSE, ACM (2008) 855–858
6. Demsky, B., Rinard, M.C.: Goal-directed reasoning for specification-based data structure repair. IEEE Tran. on Soft. Eng. **32** (2006) 931–951
7. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. SIAM J. Comput. **Vol. 29** (1999) 132–158
8. Knuth, D.E.: A generalization of Dijkstra's algorithm. Inf. Process. Lett. **6** (1977) 1–5
9. Karp, R., Miller, R., Thatcher, J.: Reducibility among combinatorial problems. In: Complexity of Computer Computations. Plenum Press (1972) 103, 85
10. Vazirani, V.V.: Approximation algorithms. Springer (2001)
11. Bille, P.: A survey on tree edit distance and related problems. THEOR. COMPUT. SCI **337** (2005) 217—239
12. Samanta, R., Deshmukh, J.V., Emerson, E.A.: Automatic generation of local repairs for Boolean programs. In: Proc. of FMCAD. (2008) 213–222
13. Griesmayer, A., Bloem, R., Cook, B.: Repair of Boolean programs with an application to C. In: Proc. of CAV. (2006) 358–371
14. Elkarablieh, B., Garcia, I., Suen, Y.L., Khurshid, S.: Assertion-based repair of complex data structures. In: Proc. of ASE. (2007) 64–73
15. Emerson, E.A., Kahlon, V.: Model checking large-scale and parameterized resource allocation systems. In: Proc. of TACAS. (2002) 251–265
16. Deshmukh, J., Emerson, E., Gupta, P.: Automatic verification of parameterized data structures. In: Proc. of TACAS. (2006) 27–41
17. Deshmukh, J., Emerson, E.A.: Verification of recursive methods on tree-like data structures. Accepted for publication in Proc. of FMCAD (2009)