Synthesizing Transducers from Complex Specifications

Anvay Grover
The University of Wisconsin-Madison
Madison, USA
anvayg@cs.wisc.edu

Ruediger Ehlers
Clausthal University of Technology
Clausthal, Germany
ruediger.ehlers@tu-clausthal.de

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

***Abstract*—Automating string transformations has been a driving application of program synthesis. Existing synthesizers that solve this problem produce programs in domain-specific languages (DSL) that are designed to simplify synthesis and therefore lack nice formal properties. This limitation prevents the synthesized programs from being used in verification applications (e.g., to check complex pre-post conditions) and makes the synthesizers hard to modify due to their reliance on the given DSL.**

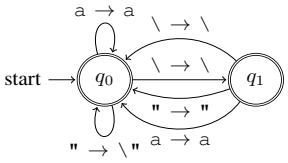
We present a constraint-based approach to synthesizing transducers, a model with strong closure and decidability properties. Our approach handles three types of specifications: input-output (*i*) examples, (*ii*) types expressed as regular languages, and (*iii*) distances that bound how many characters the transducer can modify when processing an input string. Our work is the first to support such complex specifications and it does so by using the algorithmic properties of transducers to generate constraints that can be solved using off-the-shelf SMT solvers. Our synthesis approach can be extended to many transducer models and it can be used, thanks to closure properties of transducers, to compute repairs for partially correct transducers.

I. Introduction

String transformations are used in data transformations [1], sanitization of untrusted inputs [2], [3], and many other domains [4]. Because in these domains bugs may cause serious security vulnerabilities [2], there has been increased interest in building tools that can help programmers verify [2], [3] and synthesize [1], [5], [6] string transformations.

Techniques for *verifying* string transformations rely on automata-theoretic approaches that provide powerful decidability properties [2]. On the other hand, techniques for *synthesizing* string transformations rely on domain-specific languages (DSLs) [1], [5]. These DSLs are designed to make synthesis practical and have to give up the closure and decidability properties enabled by automata-theoretic models. The disconnect between the two approaches raises a natural question: *Can one synthesize automata-based models and therefore retain and leverage their elegant properties?*

A *finite state transducer* (FT) is an automaton where each transition reads an input character and outputs a string of output characters. For instance, Figure 1 shows a transducer that 'escapes' instances of the " character. So, on input `a"\"a`, the transducer outputs the string `a\"\\"a`. FTs have found wide adoption in a variety of domains [3], [7] because of their many desirable properties (e.g., decidable equivalence check and closure under composition [8]). There has been



(a) Transducer EscapeQuotes

Examples: {a"a ↦ a\"a, a\\a ↦ a\\a, a\a ↦ a\a, a\"a ↦ a\"a, \ ↦ \}
Types: `[a"]*\?|([a"]*\[a"\][a"]*)*` → `a*\?|(a*\[a"\]a*)*`
Distance: At most 1 edit per input character

(b) Specification to synthesize EscapeQuotes

Fig. 1: Simplified version of EscapeQuotes from [2].

increasing work on building SMT solvers for strings that support transducers; the Ostrich tool [9] allows a user to write programs in SMT where string-transformations are modelled using transducers. One can then write constraints over such programs and use an SMT solver to automatically check for satisfiability or prove unsatisfiability of those constraints. For example, given a program like the following:

```
y = escapeQuotes(x)
z = escapeQuotes(y)
assert(y==z) //Checking idempotence
```

one can use Ostrich to write a set of constraints and use them to prove whether the assertion holds. However, to do so, one needs to first write a transducer T that implements the function `escapeQuotes`. However, writing transducers by hand is a cumbersome and error-prone task and what we present in this paper is an approach for synthesizing such transducers.

In this paper, we present a technique for synthesizing transducers from high-level specifications. We use three different specification mechanisms to quickly yield desirable transducers: input-output examples, input-output types, and input-output distances. When provided with the specification in Figure 1b, our approach yields the transducer in Figure 1. While none of the three specification mechanisms are effective in isolation, they work well altogether. Input-output examples are easy to provide, but only capture finitely many inputs. Similarly, input-output types are a natural way to prevent a transducer from generating undesired strings and can often be

https://doi.org/

obtained from function/API specifications. Last, input-output distances are a natural way to specify how much of the input string should be preserved by the transformation.

We show that if the size of the transducers is fixed, all such specifications can be encoded as a set of constraints whose solution directly provides a transducer. While the constraints for examples are fairly straightforward, to encode types and distances, we show that one can use constraints to "guess" the simulation relation and the invariants necessary to prove that the transducer has the given type and respects the given distance constraint.

Because our constraint-based approach is based on decision procedures and is modular, it can support more complex models of transducers: (*i*) Symbolic Finite Transducers (s-FTs), which support large alphabets [10], and (*ii*) FTs with lookahead, which can express functions that otherwise require non-determinism. In addition, closure properties of transducers allow us to reduce repair problems for string transformations to our synthesis problem.

Contributions: We make the following contributions.

- A constraint-based synthesis algorithm for synthesizing transducers from complex specifications (Sec. III).
- Extensions of our synthesis algorithm to more complex models—e.g., symbolic transducers and transducers with lookahead—and problems—e.g., transducer repair—that showcase the flexibility of our approach and the power of working with transducers, which enjoy strong theoretical properties—unlike domain-specific languages (Sec. IV).
- ASTRA: a tool that can synthesize and repair transducers and compares well with a state-of-the-art tool for synthesizing string transformations (Sec. V).

Proofs and additional results are available at [11].

II. Transducer Synthesis Problem

In this section, we define the transducer synthesis problem.

A *deterministic finite automaton* (DFA) over an alphabet Σ is a tuple $D = (Q_D, \delta_D, q_D^{init}, F_D)$: Q_D is the set of states, $\delta_D : Q_D \times \Sigma \to Q_D$ is the transition function, q_D^{init} is the initial state, and F_D is the set of final states. The extended transition function $\delta_D^* : Q_D \times \Sigma^* \to Q_D$ is defined as $\delta_D^*(q, \varepsilon) = q$ and $\delta_D^*(q, au) = \delta_D^*(\delta_D(q, a), u)$. We say that D accepts a string w if $\delta_D^*(q_D^{init}, w) \in F_D$. The *regular language* $\mathcal{L}(D)$ is the set of strings accepted by a DFA D.

A total *finite state transducer* (FT) is a tuple $T = (Q_T, \delta_T^{st}, \delta_T^{out}, q_T^{init})$, where Q_T are states and q_T^{init} is the initial state. Transducers have two transition functions: $\delta_T^{st} : q_T \times \Sigma \to q_T$ defines the target state, while $\delta_T^{out} : q_T \times \Sigma \to \Sigma^*$ defines the output string of each transition. The extended function for states δ_T^{st*} is defined analogously to the extended transition function for a DFA. The extended function for output strings is defined as $\delta_T^{out*}(q, \varepsilon) = \varepsilon$ and $\delta_T^{out*}(q, au) = \delta_T^{out*}(q, a) \cdot \delta_T^{out}(\delta_T^{st*}(q, a), u)$. Given a string w we use $T(w)$ to denote $\delta_T^{out*}(q_T^{init}, w)$, i.e., the output string generated by T on w. Given two DFAs P and Q, we write $\{P\}T\{Q\}$ for a transducer T iff for every string s in $\mathcal{L}(P)$, the output string $T(s)$ belongs to $\mathcal{L}(Q)$.

An *edit operation* on a string is either an insertion/deletion of a character, or a replacement of a character with a different one. For example, editing the string `ab` to the string `acb` requires one edit operation, which is inserting a `c` after the `a`. The *edit distance* $ed\_dist(s, t)$ between two strings s and t is the number of edit-operations required to reach t from s. We use $len(w)$ to denote the length of a string w. The *mean edit distance* $mean\_ed\_dist(s, t)$ between two strings s and t is defined as $ed\_dist(s, t)/len(s)$. For example, the mean edit distance from `ab` to `acb` is $1/2 = .5$.

We can now formulate the transducer synthesis problem. We assume a fixed alphabet Σ. If the specification requires that s is translated to t, we write that as $s \mapsto t$.

Problem Statement 1 (Transducer Synthesis)**.** *The transducer synthesis problem has the following inputs and output:*

Inputs

- *Number of states k and upper bound l on the length of the output of each transition.*
- *Set of input-output examples $E = [\overline{s \mapsto t}]$.*
- *Input-output types P and Q, given as DFAs.*
- *A positive upper bound $d \in \mathbb{Q}$ on the mean edit distance.*

Output *A total transducer $T = (Q_T, \delta_T^{st}, \delta_T^{out}, q_T^{init})$ with k states such that:*

- *Every transition of T has an output with length at most l, i.e., $\forall q_T \in Q_T, a \in \Sigma.\ len(\delta_T^{out}(q, a)) \leq l$.*
- *T is consistent with the examples: $\forall s \mapsto t \in E.\ T(s) = t$.*
- *T is consistent with input-output types, i.e., $\{P\}T\{Q\}$.*
- *For every string $w \in P$, $mean\_ed\_dist(w, T(w)) \leq d$.*

The synthesis problem that we present here is for FTs, and in Section III, we provide a sound algorithm to solve it using a system of constraints. One of our key contributions is that our encoding can be easily adapted to synthesizing richer models than FTs (e.g., symbolic transducers [8] and transducers with regular lookahead), while still using the same encoding building blocks (Section IV).

III. Constraint-based Transducer Synthesis

In this section, we present a way to generate constraints to solve the transducer synthesis problem defined in Section II. The synthesis problem can then be solved by invoking a *Satisfiability Modulo Theories* (SMT) solver on the constraints.

We use a constraint encoding, rather than a direct algorithmic approach because of the multiple objectives to be satisfied. Synthesizing a transducer that translates a set of input-output examples is already an NP-Complete problem [12]. On top of that, we also need to handle input-output types and distances. Our encoding is divided into three parts, one for each objective, which are presented in the following subsections. This division makes our encoding very modular and programmable. In Section IV we show how it can be adapted to different transducer models and problems. We include a brief description of the size of the constraint encoding in the extended version.

The transducer we are synthesizing has k (part of the problem input) states $Q_T = \{q_0, ..., q_{k-1}\}$. We often use q_T^{init} as an alternative for q_0, the initial state of T.

We illustrate how our encoding represents a transition $q_1 \xrightarrow{\texttt{a/bc}} q_2$. The target state is captured using an uninterpreted function $\texttt{d}^{\texttt{st}} : Q_T \times \Sigma \to Q_T$, e.g., $\texttt{d}^{\texttt{st}}(q_1, \texttt{a}) = q_2$. Representing the output of the transition is trickier because its length is not known a priori. The output bound l allows us to limit the number of characters that may appear in the output. We use an uninterpreted function $\texttt{d}^{\texttt{out}}_{\texttt{ch}} : Q_T \times \Sigma \times \{0, \ldots, l-1\} \to \Sigma$ to represent each character in the output string; in our example, $\texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_1, \texttt{a}, 0) = \texttt{b}$ and $\texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_1, \texttt{a}, 1) = \texttt{c}$. Since an output string's length can be smaller than l, we use an additional uninterpreted function $\texttt{d}^{\texttt{out}}_{\texttt{len}} : Q_T \times \Sigma \to \{0, \ldots, l\}$ to model the length of a transition's output; in our example $\texttt{d}^{\texttt{out}}_{\texttt{len}}(q_1, \texttt{a}) = 2$. We say an assignment to the above variables extends to a transducer T for the transducer T obtained by instantiating δ^{st} and δ^{out} as described above.

A. Input-output Examples

Goal: For each input output-example $s \mapsto t \in E$, T should translate s to t.

Translating s to the correct output string means that $\delta_T^{out*}(q_T^{init}, s) = t$. Generating constraints that capture this behavior of T on an example is challenging because we do not know a priori what parts of t are produced by what steps of the transducer's run. Suppose that we need to translate $s = a_0a_1$ to $t = b_0b_1b_2$. A possible solution is for the transducer to have the run $q_0 \xrightarrow{a_0/b_0} q1 \xrightarrow{a_1/b_1b_2} q_2$. Another possible solution might be to instead have $q_0 \xrightarrow{a_0/b_0b_1} q1 \xrightarrow{a_1/b_2} q_2$. Notice that the two runs traverse the same states but produce different parts of the output strings at each step. Intuitively, we need a way to "track" how much output the transducer has produced before processing the i-th character in the input and what state it has landed in. For every input example $s \mapsto t$ such that $s = a_0 \cdots a_n$ and $t = b_0 \cdots b_m$, we introduce an uninterpreted function $\texttt{config}_s : \{0, \ldots, n\} \to \{0, \ldots, m\} \times Q_T$ such that $\texttt{config}_s(i) = (j, q_T)$ iff after reading $a_0 \cdots a_{i-1}$, the transducer T has produced the output $b_0 \cdots b_{j-1}$ and reached state q_T—i.e., $\delta_T^{out*}(q_0, a_0 \cdots a_{i-1}) = b_0 \cdots b_{j-1}$ and $\delta_T^{st*}(q_0, a_0 \cdots a_{i-1}) = q_T$.

We describe the constraints that describe the behavior of $\texttt{config}_s$. Constraint 1 states that a configuration must start at the initial state and be at position 0 in the output.

$$\texttt{config}_s(0) = (0, q_T^{init}) \tag{1}$$

Constraint 2 captures how the configuration is updated when reading the i-th character of the input. For every $0 \leq i < n$, $0 \leq j < m$, $c \in \Sigma$, and $q_T \in Q_T$:

$$\begin{aligned} &\texttt{config}_s(i) = (j, q_T) \wedge a_i = c \Rightarrow \\ &\quad [\bigwedge_{0 \leq z < l} (\texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_T, c, z) = b_{j+z} \vee z \geq \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c)) \wedge \\ &\quad \texttt{config}_s(i+1) = (j + \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c), \texttt{d}^{\texttt{st}}(q_T, c))] \end{aligned} \tag{2}$$

Informally, if the i-th character is c and the transducer has reached state q_T and produced the characters $b_0 \cdots b_{j-1}$ so far, the transition reading c from state q_T outputs characters $b_j \cdots b_{j+f-1}$, where f is the output length of the transition. The next configuration is then $(j + f, \texttt{d}^{\texttt{st}}(q_T, c))$.

Finally, Constraint 3 forces T to be completely done with generating t when s has been entirely read. Recall that $len(s) = n$ and $len(t) = m$.

$$\bigvee_{q_T \in Q_T} \texttt{config}_s(n) = (m, q_T) \tag{3}$$

The encoding for examples is sound and complete [11].

B. Input-Output Types

Goal: T should satisfy the property $\{P\}T\{Q\}$.

Encoding this property using constraints is challenging because it requires enforcing that when T reads one of the (potentially) infinitely many strings in P it always outputs a string in Q. To solve this problem, we draw inspiration from how one proves that the property $\{P\}T\{Q\}$ holds—i.e., using a simulation relation that relates runs over P, T and Q. Intuitively, if P has read some string w, we need to be able to encode the behavior of T in terms of w, i.e., what state of T this transducer is in after reading w and what output string w' it produced. Further, we also need to be able to encode in which state Q would be after reading the output string w'. We do this by introducing a function $\texttt{sim}$: $Q_P \times Q_T \times Q_Q \to \{0, 1\}$, which preserves the following invariant: $\texttt{sim}(q_P, q_T, q_Q)$ holds if there exist strings w, w' such that $\delta_P^*(q_P^{init}, w) = q_P$, $\delta_T^{st*}(q_T^{init}, w) = q_T$, $\delta_T^{out*}(q_T^{init}, w) = w'$, and $\delta_Q^*(q_Q^{init}, w') = q_Q$.

Constraint 4 states the initial condition of the simulation—i.e., P, T, and Q are in their initial states.

$$\texttt{sim}(q_P^{init}, q_T^{init}, q_Q^{init}) \tag{4}$$

Constraint 5 encodes how we advance the simulation relation for states q_P, q_T, q_Q and for a character $c \in \Sigma$, using free variables $c_0 \ldots, c_{l-1}$ and $q_Q^0 \ldots, q_Q^l$ that are separate for each combination of q_P, q_T, q_Q, and c:

$$\begin{aligned} \texttt{sim}(q_P, q_T, q_Q) \Rightarrow &\bigwedge_{0 \leq z \leq l} (\texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) = z \Rightarrow \\ &[\bigwedge_{0 \leq x < z} \texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_T, c, x) = c_x] \wedge \\ &[q_Q^0 = q_Q \wedge \bigwedge_{1 \leq x < z} q_Q^x = \texttt{d}_Q(q_Q^{x-1}, c_{x-1})] \wedge \\ &\texttt{sim}(\delta_P(q_P, c), \texttt{d}^{\texttt{st}}(q_T, c), q_Q^z)) \end{aligned} \tag{5}$$

Intuitively, if $\texttt{sim}(q_P, q_T, q_Q)$ and we read a character c, P moves to $\delta_P(q_P, c)$ and T moves to $\texttt{d}^{\texttt{st}}(q_P, c)$. However, we also need to advance Q and the $\texttt{d}^{\texttt{out}}_{\texttt{len}}$ symbols produced by $\texttt{d}^{\texttt{out}}_{\texttt{ch}}$. We hard-code the transition relation δ_Q in an uninterpreted function $\texttt{d}_Q : Q_Q \times \Sigma \to Q_Q$, and apply it to compute the output state reached when reading the output string. E.g., if $\texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) = 2$ and $\texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_T, c, 0) = c_0$ and $\texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_T, c, 1) = c_1$, the next state in Q is $\texttt{d}_Q(\texttt{d}_Q(q_Q, c_0), c_1)$.

Lastly, Constraint 6 states that if we encounter a string in $\mathcal{L}(P)$—i.e., P is in a state $q_P \in F_P$—the relation does not

contain a state $q_Q \notin F_Q$. Since Q is deterministic, this means that Q accepts T's output.

$$\bigwedge_{q_P \in F_P} \bigwedge_{q_Q \notin F_Q} \neg \texttt{sim}(q_P, q_T, q_Q) \tag{6}$$

The constraint encoding for types is sound and complete [11].

C. Input-output Distance

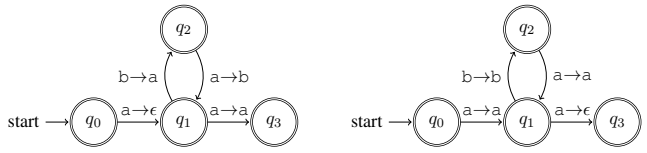
Goal: The mean edit distance between any input string w in $\mathcal{L}(P)$ and the output string $T(w)$ should not exceed d.

Capturing the edit distance for all the possible inputs in the language of P and the corresponding outputs produced by the transducer is challenging because these sets can be infinite. Furthermore, exactly computing the edit distance between an input and an output string may involve comparing characters appearing on different transitions in the transducer run. For example, consider the transducer shown in Figure 2a and suppose that we are only interested in strings in the input type $P = \texttt{a(ba)*a}$. The first transition from q_0 deletes the `a`, therefore making 1 edit. This transducer has a cycle between states q_1 and q_2, which can be taken any number of times. Each iteration, locally, would require that we make 2 edits: one to change the `b` to `a`, and the other to change the `a` to `b`. However, the total number of edits made over any string in the input type $P = \texttt{a(ab)*a}$ by this transducer is 1, because the transducer changes strings of the form $\texttt{a(ba)}^n\texttt{a}$ to be of the form $\texttt{(ab)}^n\texttt{a}$. Looking at the transitions in isolation, we are prevented from deducing that the edit distance is always 1 because the first transition delays outputting a character. If there was no such delay, as is the case for the transducer in Figure 2b, which is equivalent on the relevant input type to the one in Figure 2a, then this issue would not arise.

We take inspiration from Benedikt et al. [13] and focus on the simpler problem of synthesizing a transducer that has 'aggregate cost' that satisfies the given objective.[1] For a transducer T and string $s = a_0 \dots a_n$, let $q_T^{init} \xrightarrow{a_0/y_0} q_T^1 \dots q_T^n \xrightarrow{a_n/y_n} q_T^{n+1}$ be the run of s on T. Then, the *aggregate cost* of T on s is the sum of the edit distances $ed\_dist(a_i, y_i)$ over all indices $0 \leq i \leq n$. The *mean aggregate cost* of T on s is the aggregate cost divided by $len(s)$, the length of s. It follows that if T has a mean aggregate cost lower than some specified d for every string, then it also has a mean edit distance lower than d for every string.

However, the mean aggregate cost overapproximates the edit distance, e.g., the transducer in Figure 2a has mean aggregate cost 1, while the mean edit distance when considering only strings in $P = \texttt{a(ab)*a}$ is less than $1/2$. For this reason, if the mean edit distance objective was set to $1/2$, our constraint encoding can only synthesize the transducer in Figure 2b, and not the equivalent one in Figure 2a.

[1]Benedikt et al. [13] studied a variant of the problem where the distance is bounded by some finite constant. Their work shows that when there is a transducer between two languages that has some bounded global edit distance, then there is also a transducer that is bounded (but with a different bound) under a local method of computing the edit distance—i.e., one where the computation of the edit distance is done transition by transition.



(a) Transducer with delayed output *(b) Transducer without delay*

Fig. 2: Transducers with and without delay.

Our encoding is complete for transducers in which the aggregate cost coincides with the actual edit distance. We leave the problem of being complete with regards to global edit distance as an open problem. In fact, we are not even aware of an algorithm for *checking* (instead of synthesizing) whether a transducer satisfies a mean edit distance objective.[2] In Section IV-B, we present transducers with lookahead, which can mitigate this source of incompleteness. Furthermore, our evaluation shows that using the aggregate cost and enabling lookahead are both effective techniques in practice.

We can now present our constraints. First, we provide constraints for the edit distance of individual transitions (recall that transitions are being synthesized and we therefore need to compute their edit distances separately). Secondly, we provide constraints that implicitly compute state invariants to capture the aggregate cost between input and output strings at various points in the computation. We are given a rational number d as an input to the problem, which is the allowed distance bound.

Edit Distance of Individual Transitions. To compute the edit distance between the input and the output of each transition, we introduce a function $\texttt{ed}\colon Q_T \times \Sigma \to \mathbb{Z}$. For a transition from state q_T reading a character c, $\texttt{ed}(q_T, c)$ represents the edit distance between c and $\delta_T^{out}(q_T, c)$. Notice that this quantity is bounded by the output bound l. The constraints to encode the value of this function are divided into two cases: i) the output of the transition contains the input character c (Constraint 7), ii) the output of the transition *does not* contain the input character c (Constraint 8). In both cases, the values are set via a simple case analysis on whether the length of the output is 0 (edit distance is 1) or not (the edit distance is related to the length of the output).

$$\begin{aligned}&[\bigvee_{0 \leq z < \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c)} \texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_T, c, z) = c] \Rightarrow \\ &\quad [\texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) = 0 \Rightarrow \texttt{ed}(q_T, c) = 1 \wedge \\ &\quad \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) \neq 0 \Rightarrow \texttt{ed}(q_T, c) = \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) - 1]\end{aligned} \tag{7}$$

$$\begin{aligned}&[\bigwedge_{0 \leq z < \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c)} \texttt{d}^{\texttt{out}}_{\texttt{ch}}(q_T, c, z) \neq c] \Rightarrow \\ &\quad [\texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) = 0 \Rightarrow \texttt{ed}(q_T, c) = 1 \wedge \\ &\quad \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c) \neq 0 \Rightarrow \texttt{ed}(q_T, c) = \texttt{d}^{\texttt{out}}_{\texttt{len}}(q_T, c)]\end{aligned} \tag{8}$$

[2]The mean edit distance is similar to mean payoff [14], which discounts a cost by the length of a string and looks at the behavior of a transducer in the limit. Our distance is different because 1) it looks at finite-length strings, and 2) it requires computing the edit distance, which cannot be done one transition at a time.

Edit Distance of Arbitrary Strings. Suppose that T has the transitions $q_0 \xrightarrow{\texttt{a/a}} q1 \xrightarrow{\texttt{a/bc}} q_2$, and the specified mean edit distance is $d = 0.5$. The edit distance is 0 for the first transition and 2 for the second one. For the input string `aa`, the mean aggregate cost is $2/2$, which means that the specification is not satisfied. In general, we cannot keep track of every input string in the input type and look at its length and the number of edits that were made over it. So, how can we compute the mean aggregate cost over any input string? The first part of our solution is to scale the edit distance over a single transition depending on the specified mean edit distance. This operation makes it such that an input string is under the edit distance bound if the sum of the weighted edit distances of its transitions is ≥ 0. The invariant we need to maintain is that the sum of the weights at any stage of the run gives us where we are with regard to the mean aggregate cost. For each transition we compute the difference between the edit distance over the transition and the specified mean edit distance d. We introduce the uninterpreted function $\texttt{wed} : Q_T \times \Sigma \to \mathbb{Q}$, which stands for weighted edit distance. For a transition at q_T reading a character c, the weighted edit distance is given by $\texttt{wed}(q_T, c) = d - \texttt{ed}(q_T, c)$. The sum of the weights of all transitions tells us the cumulative difference. Going back to our example, the weighted edit distances of the two transitions are $\texttt{wed}(q_0, \texttt{a}) = 0.5$ and $\texttt{wed}(q_1, \texttt{a}) = -1.5$, making the cumulative distance -1 and implying that the specification is violated. We can now compute the mean edit distance over a run without keeping track of the length of the run and the number of edits performed over it.

We still need to compute the weighted edit distance for every string in the possibly infinite language $\mathcal{L}(P)$. Building on the idea of simulation from the previous section, we introduce a new function called $\texttt{en} : Q_P \times Q_T \times Q_Q \to \mathbb{Q}$, which tracks an upper bound on the sum of the distances so far at that point in the simulation. This function is similar to a *progress measure*, which is a type of invariant used to solve *energy games* [15], a connection we expand on in Section VI. In particular, we already know that if there exist strings w, w' such that $\delta_P^*(q_P^{init}, w) = q_P$, $\delta_T^{st*}(q_T^{init}, w) = q_T$, $\delta_T^{out*}(q_T^{init}, w) = w'$, and $\delta_Q^*(q_Q^{init}, w') = q_Q$, then we have $\texttt{sim}(q_P, q_T, q_Q)$. Let this run over T be denoted by $q_T^{init} \xrightarrow{a_0/y_0} q_T^1 \dots q_T^{n-1} \xrightarrow{a_{n-1}/y_{n-1}} q_T$, where $w = a_0 \cdots a_{n-1}$, $w' = y_0 \cdots y_{n-1}$, and $q_T = q_T^n$. We have that $\texttt{en}(q_P, q_T, q_Q) \geq \sum_{i=0}^{n-1} \texttt{wed}(q_T^i, a_i)$.

The `en` function is a budget on the number of edits we can still perform. At the initial states, we start with no 'initial credit' and the energy is 0.

$$\texttt{en}(q_P^{init}, q_T^{init}, q_Q^{init}) = 0 \tag{9}$$

Constraint 10 bounds the energy budget according to the weighted edit distance of a transition by computing the minimum budget required at any point to still satisfy the distance bound. For each combination of q_P, q_T, q_Q, and $c \in \Sigma$, the constraint uses free variables $c_0, \dots, c_l$ and $q_Q^0, \dots, q_Q^{l-1}$:

$$\begin{aligned}
&\bigwedge_{0 \leq z < l} (\texttt{d}_{\texttt{len}}^{\texttt{out}}(q_T, c) = z \Rightarrow \\
&[\bigwedge_{0 \leq x < z} \texttt{d}_{\texttt{ch}}^{\texttt{out}}(q_T, c, x) = c_x] \wedge [q_Q^0 = q_Q \wedge \bigwedge_{1 \leq x < z} q_Q^x = \texttt{d}_Q(q_Q^{x-1}, c_{x-1})] \wedge \\
&\texttt{en}(q_P, q_T, q_Q) \geq \texttt{en}(\delta_P(q_P, c), \texttt{d}^{\texttt{st}}(q_T, c), q_Q^z) - \texttt{wed}(q_T, c))
\end{aligned} \tag{10}$$

In our example, Constraint 10 encodes that the energy at q_0 can be 1 less than that at q_1, but that the energy at q_1 needs to be 3 greater than at q_2 since we need to spend 3 edit operations over the second transition.

At any point during a run, the transducer is allowed to go below the mean edit distance and then 'catch up' later because we only care about the edit distance when the transducer has finished reading a string in $\mathcal{L}(P)$. Therefore, when we reach a final state of P, the transducer should not be in 'energy debt'.

$$\bigwedge_{q_P \in F_P} \texttt{sim}(q_P, q_T, q_Q) \Rightarrow \texttt{en}(q_P, q_T, q_Q) \geq 0 \tag{11}$$

The encoding presented in this section is sound [11].

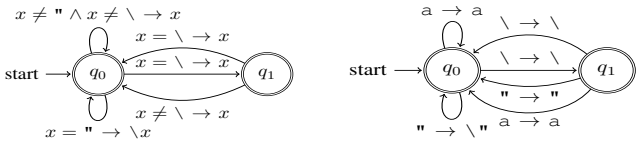
IV. Richer Models and Specifications

We extend our technique to more expressive models (Sections IV-A and IV-B) and show how our synthesis approach can be used not only to synthesize transducers, but also to repair them (Section IV-C). Furthermore, in the extended version of the paper, we describe an encoding of an alternative distance measure [11].

A. Symbolic Transducers

Symbolic finite automata (s-FA) and transducers (s-FT) extend their non-symbolic counterparts by allowing transitions to carry predicates and functions to represent (potentially infinite) sets of input characters and output strings. Figure 3a shows an s-FT that extends the escapeQuotes transducer from Figure 1a to handle alphabetic characters. The bottom transition from q_0 reads a character `"` (bound to the variable x) and outputs the string `\"` (i.e., a `\` followed by the character stored in x). Symbolic finite automata (s-FA) are s-FTs with no outputs. To simplify our exposition, we focus on s-FAs and s-FTs that only operate over ASCII characters that are ordered by their codes. In particular, all of our predicates are unions of intervals over characters (i.e., $x \neq \backslash$ is really the union of intervals [`NUL`-`[]` and `]`-`DEL`]); we often use the predicate notation instead of explicitly writing the intervals for ease of presentation. Furthermore, we only consider two types of output functions: constant characters and offset functions of the form $x + k$ that output the character obtained by taking the input x and adding a constant k to it—e.g., applying $x + (-32)$ to a lowercase alphabetic letter gives the corresponding uppercase letter.

In the rest of the section, we show how we can solve the transducer synthesis problem in the case where P and Q are s-FAs and the goal is to synthesize an s-FT (instead of an FT) that meets the given specification. Intuitively, we do this



(a) escapeQuotes s-FT *(b) $F(escapeQuotes)$*

minterms: $[x \neq \texttt{"} \wedge x \neq \backslash]$, $[x = \texttt{"}]$, $[x = \backslash]$
witness char: $wit([x \neq \texttt{"} \wedge x \neq \backslash])$=a, $wit([x = \texttt{"}])$=", $wit([x = \backslash])$=\

(c) Set of minterms and their witness elements

Fig. 3: Example of Finitization

by 'finitizing' the alphabet of the now symbolic input-output types, synthesizing a finite transducer over this alphabet using the technique presented in Section III, and then extracting an s-FT from the solution.

Finitizing the Alphabet. The idea of finitizing the alphabet of s-FAs is a known one [8] and is based on the concept of $minterms$, which is the set of maximal satisfiable Boolean combinations of the predicates appearing in the s-FAs. For an s-FA M, we can define its set of predicates as: $Predicates(M) = \{\phi \mid q \xrightarrow{\phi} q' \in \delta_M\}$. The set of minterms $mterms(M)$ is the set of satisfiable Boolean combinations of all the predicates in $Predicates(M)$. For example, for the set of predicates over the s-FT escapeQuotes in Figure 3a, we have that $mterms$(escapeQuotes) $= \{x \neq \texttt{"} \wedge x \neq \backslash, x = \texttt{"}, x = \backslash\}$. The reader can learn more about minterms in [8]. We assign each minterm a representative character, as indicated in Figure 3c, and then construct a finite automaton from the resulting finite alphabet Σ. For a character $c \in \Sigma$, we refer to its corresponding minterm by $mt(c)$. In the other direction, for each minterm $\psi \in minterms(M)$, we refer to its uniquely determined representative character by $wit(\psi)$.

For an s-FA M, we denote its corresponding FA over the alphabet $mterms(M)$ with $F(M)$. Given an s-FA M, the set of transitions of $F(M)$ is defined as follows:

$$\delta_{F(M)}=\{q \xrightarrow{wit(\psi)} q' | q \xrightarrow{\phi} q' \wedge \psi \in mterms(M) \wedge \text{IsSat}(\psi \wedge \phi)\}$$

This algorithm replaces a transition guarded by a predicate ϕ in the given s-FA with a set of transitions consisting of the witnesses of the minterms where ϕ is satisfiable. In interval arithmetic this is the set of intervals that intersect with the interval specified by ϕ. The transition from q_1 guarded by the predicate $[x \neq \backslash]$ in Figure 3a intersects with 2 minterms $[x \neq \texttt{"} \wedge x \neq \backslash]$ and $[x = \texttt{"}]$. As a result, we see that this transition is replaced by two transitions in Figure 3b, one that reads " and another that reads a.

From FTs to s-FTs. Once we have synthesized an FT T, we need to extract an s-FT from it. There are many s-FTs equivalent to a given FT and here we present one way of doing this conversion which is used in our implementation. Let the size of an interval I (the number of characters it contains) be given by $size(I)$, and the offset between 2 intervals I_1 and I_2 (i.e. the difference between the least elements of I_1 and I_2) be given by $offset(I_1, I_2)$. Suppose we have a transition $q \xrightarrow{c/y_0 \cdots y_n} q'$, where $c, y_i \in \Sigma$. Then, we construct a transition $q \xrightarrow{mt(c)/f_0 \cdots f_n} q'$, where for each y_i, the corresponding function f_i is determined by the following rules (x always indicates variable bound to the input predicate):

1) If $c = y_i$, then $f_i = (x)$, i.e. the identity function.
2) If $mt(c)$ and $mt(y_i)$ consist of single intervals I_1 and I_2, respectively, such that $size(I_1) = size(I_2)$, then $f_i = (x + offset(I_1, I_2))$. For instance, if the input interval is [a-z] and the output interval is [A-Z], then the output function is $(x + (-32))$, which maps lowercase letters to uppercase ones.
3) Otherwise $f_i = y_i$—i.e., the output is a character in the output minterm.

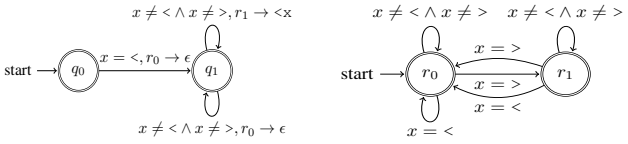
While our s-FT recovery algorithm is sound, it may apply case 3 more often than necessary and introduce many constants, therefore yielding a transducer that does not generalize well to unseen examples. Our evaluation shows that our technique works well in practice. The proof of soundness of this algorithm in the extended version [11].

B. Synthesizing Transducers with Lookahead

Deterministic transducers cannot express functions where the output at a certain transition depends on future characters in the input. Consider the problem of extracting all substrings of the form <x> (where x ≠ <) from an input string. This is the *getTags* problem from [16]. A deterministic transducer cannot express this transformation because when it reads < followed by x it has to output <x if the next character is a > and nothing otherwise. However, the transducer does not have access to the next character!

Instead, we extend our technique to handle deterministic transducers with lookahead, i.e., the ability to look at the string suffix when reading a symbol. Formally, a *Transducer with Regular Lookahead* is a pair (T, R) where T is an FT with $\Sigma_T = Q_R \times \Sigma$, and R is a total DFA with $\Sigma_R = \Sigma$. The transducer T now has another input in its transition function, although it still only outputs characters from Σ, i.e., $\delta_T^{out} : Q_T \times (Q_R \times \Sigma) \to \Sigma$, and $\delta_T^{st} : Q_T \times (Q_R \times \Sigma) \to Q_T$. The semantics is defined as follows. Given a string $w = a_0 \cdots a_n$, we define a function r_w such that $r_w(i) = \delta_R(q_R^{init}, a_n \cdots a_{i+1})$. In other words, $r_w(i)$ gives the state reached by R on the reversed suffix starting at $i+1$. At each step i, the transducer T reads the symbol $(a_i, r_w(i))$. The extended transition functions now take as input a lookahead word, which is a sequence of pairs of lookahead states and characters, i.e., from $(Q_R \times \Sigma)^*$.

To synthesize transducers with lookahead, we introduce uninterpreted functions $\texttt{d}_\texttt{R}$ for the transition function of R, and $\texttt{look}_w$ for the r-values of w on R. We also introduce a bound k_R on the number of states in the lookahead automaton R (our algorithm has to synthesize both T and R). The modified constraints needed to encode input-output types and input-output examples to use lookahead are described in the extended version of the paper [11]. Part of the transducer with lookahead we synthesize for the getTags problem is shown



(a) Subset of transitions in T *(b) Lookahead automaton R*

Fig. 4: Regular lookahead for getTags

in Figure 4. Notice that there are 2 transitions out of q_1 for the same input but different lookahead state: the string <x is outputted when the lookahead state is r_1.

Lookahead and aggregate cost: Lookahead can help representing transducers, even deterministic ones, in a way that has lower aggregate cost—i.e., the aggregate cost better approximates the actual edit distance. Suppose that we want to synthesize a transducer that translates the string abc to ab and the string abd to bd. This translation can be done using a deterministic transducer with transitions $q_0 \xrightarrow{a/\epsilon} q_1 \xrightarrow{b/\epsilon} q_2$, followed by two transitions from q_2 that choose the correct output based on the next character. Such a transducer would have a high aggregate cost of 4, even though the actual edit distance is 1. In contrast, using lookahead we can obtain a transducer that can output each character when reading it; this transducer will have aggregate cost 1 for either string. We conjecture that for every transducer T, there always exists an equivalent transducer with regular lookahead (T', R) for which the edit distance computation for aggregate cost coincides with the actual edit distance of T.

C. Transducer Repair

In this section, we show how our synthesis technique can also be used to "repair" buggy transducers. The key idea is to use the closure properties of automata and transducers—e.g., closure under union and sequential compositions [8]—to reduce repair problems to synthesis ones. The ability to algebraically manipulate transducers and automata is one of the key aspects that distinguishes our work from other synthesis works that use domain-specific languages [1], [5].

We describe two settings in which we can repair an incorrect transducer T_{bad}: **1.** Let $\{P\}T_{bad}\{Q\}$ be an input-output type violated by T_{bad} and let $Out_P(T_{bad})$ be the finite automaton describing the set of strings T_{bad} can output when fed inputs in P (this is computable thanks to closure properties of transducers). We are interested in the case where $Out_P(T_{bad})\backslash Q \neq \emptyset$—i.e., T_{bad} can produce strings that are not in the output type. **2.** Let $[\overline{s \mapsto t}]$ be a set of input-output examples. We are interested in the case where there is some example $s \mapsto t$ such that $T_{bad}(s) \neq t$.

Repairing from the Input Language. This approach synthesizes a new transducer for the inputs on which T_{bad} is incorrect. Using properties of transducers, we can compute an automaton describing the exact set of inputs $P_{bad} \subseteq P$ for which T_{bad} does not produce an output in Q (see pre-image computation in [10]). Let $restrict(T, L)$ be the transducer that behaves as T if the input is in L and does not produce an output otherwise (closure under restriction [10]). If we synthesize a transducer T_1 with type $\{P_{bad}\}T_1\{Q\}$, then the transducer $restrict(T_1, P_{bad}) \cup restrict(T_{bad}, P \backslash P_{bad})$ satisfies the desired input-output type (closure under union).

Fault Localization from Examples. We use this technique when T_{bad} is incorrect on an example. We can compute a set of "suspicious" transitions by taking all the transitions traversed when $T(s) \neq t$ for some $s \mapsto t \in E$ (i.e., one of these transitions is wrong) and removing all the transitions traversed when $T(s) = t$ for some $s \mapsto t \in E$ (i.e., transitions that are likely correct). Essentially, this is a way of identifying P_{bad} when T_{bad} is wrong on some examples. We can also use this technique to limit the transitions we need to synthesize when performing repair.

V. Evaluation

We implemented our technique in a Java tool ASTRA (Automatic Synthesis of TRAnsducers), which uses Z3 [17] to solve the generated constraints. We evaluate using a 2.7 GHz Intel Core i5, RAM 8 GB, with a 300s timeout.

Q1: Can ASTRA *synthesize practical transformations?*

Benchmarks. Our first set of benchmarks is obtained from Optician [5], [6], a tool for synthesizing lenses, which are bidirectional programs used for keeping files in different data formats synchronized. We adapted 11 of these benchmarks to work with ASTRA (note that we only synthesize one-directional transformations), and added one additional benchmark extrAcronym2, which is a harder variation (with a larger input type) of extrAcronym. We excluded benchmarks that require some memory, e.g., swapping words in a sentence, as they cannot be modeled with transducers. Our second set of benchmarks (Miscellaneous) consists of 6 problems we created based on file transformation tasks (unixToDos, dosToUnix and CSVSeparator), and s-FTs from the literature–escapeQuotes from [18], getTags and quicktimeMerger from [16]. All of the benchmarks require synthesizing s-FTs and getTags requires synthesizing an s-FT with lookahead (details in Table I).

To generate the examples, we started with the examples that were used in the original source when available. In 5 cases, ASTRA synthesized a transducer that was not equivalent to the one synthesized by Optician. In these cases, we used ASTRA to synthesize two different transducers that met the specification, computed a string on which the two transducers differed, and added the desired output for that string as an example. We repeated this task until ASTRA yielded the desired transducer and we report the time for such sets of examples. The ability to check equivalence of two transducers is yet another reason why synthesizing transducers is useful. For each benchmark we chose a mean edit distance of 0.5 when the transformation could be synthesized with this distance and of 1 otherwise.

Effectiveness of ASTRA. ASTRA can solve $15/18$ benchmarks (13 in <1s and 2 under a minute) and times out on 3 benchmarks where both P and Q are big.

TABLE I: ASTRA*'s performance on the synthesis benchmarks. The right-most set of columns gives the synthesis time for* ASTRA *and Optician (under 2 different configurations). The middle set of columns gives the sizes of the parameters to the synthesis problem: Q_P and Q_Q denote the number of input and output states, and δ_P and δ_Q denote the number of transitions in the input and output types, respectively. A ✗ represents a benchmark that failed. — stands in for data that is not available; this is because we only re-ran Optician on the benchmarks that were already encoded in its benchmark set, plus a few additional ones for comparing between the tools that we wrote ourselves.*

	Benchmark	Q_P	Q_Q	δ_P	δ_Q	Σ	E	k	l	d	ASTRA (s)	Optician (s)	Optician-re (s)
Optician	extrAcronym	6	3	10	3	3	2	1	1	.5	0.11	0.05	✗
	extrAcronym2	6	3	16	3	3	3	2	1	1	0.42	—	—
	extrNum	15	13	17	12	3	1	1	1	1	0.93	0.05	0.07
	extrQuant	4	3	8	5	2	1	2	1	1	0.19	0.09	✗
	normalizeSpaces	7	6	19	10	2	2	2	1	1	0.46	16.64	✗
	extrOdds	15	9	29	13	5	3	3	2	1	15.87	0.12	✗
	capProb	3	3	3	3	2	2	2	1	1	0.05	0.05	✗
	removeLast	6	3	8	3	3	3	2	1	.5	0.21	0.15	0.07
	sourceToViews	18	7	26	15	5	3	3	2	1	50.92	0.06	✗
	normalizeNamePos	19	7	35	24	13	1	6	2	1	✗	0.05	0.10
	titleConverter	22	13	41	41	15	1	3	1	1	✗	0.07	✗
	bibtextToReadable	14	11	41	35	12	1	5	1	1	✗	0.64	0.15
Miscellaneous	unixToDos	5	7	17	19	4	4	2	2	.5	1.24	—	—
	dosToUnix	7	5	19	17	4	4	2	1	.5	0.41	—	—
	CSVSeparator	5	5	9	9	4	1	1	1	1	0.142	—	—
	escapeQuotes	2	2	6	5	3	5	2	2	1	0.188	✗	✗
	quicktimeMerger	7	3	9	3	2	2	1	1	.5	0.075	—	—
	getTags	3	3	9	4	3	5	2	2	1	0.95	✗	✗

While the synthesized transducers have at most 3 states, we note that this is because ASTRA synthesizes total transducers and then restricts their domains to the input type P. This is advantageous because synthesizing small total transducers is easier than synthesizing transducers that require more states to define the domain. For instance, when we restrict the solution of extrAcronym2 to its input type, the resulting transducer has 11 states instead of the 2 required by the original solution!

Comparison with Optician. We do not compare ASTRA to tools that only support input-output examples. Instead, we compare ASTRA to Optician on the set of benchmarks common to both tools. Like ASTRA, Optician supports input-output examples and types, but the types are expressed as regular expressions. Furthermore, Optician also attempts to produce a program that minimizes a fixed information theoretical distance between the input and output types [5].

Optician is faster when the number of variables in the constraint encoding increases, while ASTRA is faster on the normalizeSpaces benchmark. Optician, which uses regular expressions to express the input and output types, does not work so well with unstructured data. To confirm this trend, we wrote synthesis tasks for the escapeQuotes and getTags benchmarks in Optician and it was unable to synthesize those as well—e.g., escapeQuotes requires replacing every `"` character with `\"`.

To further look at the reliance of Optician on regular expressions, we converted the regular expressions used in the lens synthesis benchmarks to automata and then back to regular expressions using a variant of the state elimination algorithm that acts on character intervals. This results in regular expressions that are not very concise and might have redundancies. Optician could only solve 4/11 benchmarks that it was previously synthesizing (Optician-re in Table I).

Answer to Q1: ASTRA can solve real-world benchmarks and has performance comparable to that of Optician for similar tasks. Unlike Optician, ASTRA does not suffer from variations in how the input and output types are specified.

Q2: Can ASTRA *repair transducers in practice?*

Benchmarks. We considered the benchmarks in Table II. The only pre-existing benchmark that we found was escapeQuotes, through the interface of the Bek programming language used for verifying transducers [18]. We generated 11 additional faulty transducers to repair in the following two ways: (*i*) Introducing faults in our synthesis benchmarks: We either replaced the output string of a transition with a constant character, inserted an extra character, or deleted a transition altogether. (*ii*) Incorrect transducers: We intentionally provided fewer input-output examples and used only example-based constraints on some of our synthesis benchmarks.

All the benchmarks involve s-FTs. Three benchmarks are wrong on both input-output types and examples and the rest are only wrong on examples. Additionally, we note that to repair a transducer, we need the "right" set of minterms. Typically, the set of minterms extracted from the transducer predicates is the right one, but in the case of the escapeBrackets problems, ASTRA needs a set of custom minterms we provide manually—i.e., repairing the transducer requires coming up with a new predicate. We are not aware of another tool that solves transducer repair problems and so do not show any comparisons.

Effectiveness of ASTRA. We indicate the number of suspicious transitions identified by our fault localization procedure (Section IV-C) in the column labeled $\delta_{T_{bad}}$. In many cases,

TABLE II: ASTRA*'s performance on the repair benchmarks. Default is the case where a new transducer is synthesized for P_{bad} and Template is the case where a partial solution to the solver is provided. The $\delta_{T_{bad}}$ column gives the number of transitions that were localized by the fault-localization procedure as a fraction of the total number of transitions in the transducer. The other columns that describe the parameters of the synthesis problem in the default case are the same as for Table I.*

	Benchmark	Q_P	Q_Q	δ_P	δ_Q	Σ	E	k	l	d	$\delta_{T_{bad}}$	Default (s)	Template (s)
Fault injected	swapCase1	2	1	6	3	3	2	1	1	1	3/3	0.04	0.02
	swapCase2	2	1	4	3	3	2	1	1	1	1/2	✗	✗
	swapCase3	2	1	6	3	3	2	1	1	1	1/3	0.06	0.05
	escapeBrackets1	2	6	16	36	8	4	1	4	4	1/3	0.69	0.42
	escapeBrackets2	1	6	1	7	6	5	1	4	4	1/2	✗	✗
	escapeBrackets3	2	7	8	36	9	5	1	4	4	2/3	1.12	0.34
	caesarCipher	2	1	4	2	3	1	1	1	1	1/1	✗	✗
Synth.	extrAcronym2	11	3	30	3	3	3	2	1	1	12/30	0.59	10.15
	capProb	3	3	3	3	2	2	2	1	1	3/3	0.04	0.04
	extrQuant	8	3	16	5	2	1	2	1	1	5/10	0.37	0.51
	removeLast	6	3	8	3	3	2	2	1	.5	7/8	0.40	1.08
	escapeQuotes	3	2	9	5	3	5	2	1	1	3/5	0.17	0.10

ASTRA can detect 50% of the transitions or more as being likely correct, therefore reducing the space of unknowns.

We compare 2 different ways of solving repair problems in ASTRA. One uses the repair-from-input approach described in Section IV-C (Default in Table II). The second approach involves using a 'template', where we supply the constraint solver with a partial solution to the synthesis problem, based on the transitions that were localized as potentially buggy (Template in Table II).

ASTRA can solve 9/12 repair benchmarks (all in less than 1 second). The times using either approach are comparable in most cases. While one might expect templates to be faster, this is not always the case because the input-output specification for the repair transducer is small, but providing a template requires actually providing a partial solution, which in some cases happens to involve many constraints.

Answer to Q2: ASTRA can repair transducers with varying types of bugs.

VI. Related Work

Synthesis of string transformations. String transformations are one of the main targets of program synthesis. Gulwani showed they could be synthesized from input-output examples [1] and introduced the idea of using a DSL to aid synthesis. Optician extended the DSL-based idea to synthesizing lenses [5], [6], which are programs that transform between two formats. Optician supports not only examples but also input-output types. While DSL-based approaches provide good performance, they are also monolithic as they rely on the structure of the DSL to search efficiently. ASTRA does not rely on a DSL and can synthesize string transformations from complex specifications that cannot be handled by DSL-based tools. Moreover, transducers allow applying verification techniques to the synthesized programs (e.g., checking whether two solutions are equivalent). One limitation of transducers is that they do not have 'memory', and consequently ASTRA cannot be used for data-transformation tasks where this is required—e.g., mapping the string `Firstname Lastname` to `Lastname, Firstname`—something Optician can do. We remark that there exist transducer models with such capabilities [19] and our work lays the foundations to handle complex models in the future.

Synthesis of transducers. Benedikt et al. studied the 'bounded repair problem', where the goal is to determine whether there exists a transducer that maps strings from an input to an output type using a bounded number of edits [13]. Their work was the first to identify the relation between solving such a problem and solving games, an idea we leverage in this paper. However, their work is not implemented, cannot handle input-output examples, and therefore shies away from the source of NP-Completeness. Hamza et al. studied the problem of synthesizing minimal non-deterministic Mealy machines (transducers where every transition outputs exactly one character), from examples [12]. They prove that the problem of synthesizing such transducers is NP-complete and provide an algorithm for computing minimal Mealy machines that are consistent with the input-output examples. ASTRA is a more general framework that incorporates new specification mechanisms, e.g., input-output types and distances, and uses them all together. Mealy machines are also synthesized from temporal specifications in reactive synthesis and regular model checking, where they are used to represent parameterized systems [20], [21]. This setting is orthogonal to ours as the specification is different and the transducer is again only a Mealy machine.

The constraint encoding used in ASTRA is inspired by the encoding presented by Daniel Neider for computing minimal separating DFA, i.e. a DFA that separates two disjoint regular languages [22]. ASTRA's use of weights and energy to specify a mean edit distance is based on energy games [23], a kind of 2-player infinite game that captures the need for a player to not exceed some available resource. One way of solving such games is by defining a *progress measure* [15]. To determine whether a game has a winning strategy for one of the players, it can be checked whether such a progress measure exists in the game. We showed that the search for such a progress measure can be encoded as an SMT problem.

REFERENCES

[1] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011.

[2] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *USENIX Security Symposium, vol. 58*. USENIX, 2012.

[3] L. D'Antoni and M. Veanes, "Static analysis of string encoders and decoders," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 209–228.

[4] Y. Zhang, A. Albarghouthi, and L. D'Antoni, "Robustness to programmable string transformations via augmented abstract training," in *International Conference on Machine Learning*. PMLR, 2020, pp. 11 023–11 032.

[5] A. Miltner, S. Maina, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic, "Synthesizing symmetric lenses," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–28, 2019.

[6] A. Miltner, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic, "Synthesizing bijective lenses," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–30, 2017.

[7] M. Mohri, "Finite-state transducers in language and speech processing," *Computational linguistics*, vol. 23, no. 2, pp. 269–311, 1997.

[8] L. D'Antoni and M. Veanes, "Automata modulo theories," *Communications of the ACM*, vol. 64, no. 5, pp. 86–95, 2021.

[9] T. Chen, M. Hague, J. He, D. Hu, A. W. Lin, P. Rümmer, and Z. Wu, "A decision procedure for path feasibility of string manipulating programs with integer data type," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2020, pp. 325–342.

[10] L. D'Antoni and M. Veanes, "The power of symbolic automata and transducers," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 47–67.

[11] A. Grover, R. Ehlers, and L. D'Antoni, "Synthesizing transducers from complex specifications," 2022. [Online]. Available: https://arxiv.org/abs/2208.05131

[12] J. Hamza and V. Kunčak, "Minimal synthesis of string to string functions from examples," in *Verification, Model Checking, and Abstract Interpretation*, C. Enea and R. Piskac, Eds. Cham: Springer International Publishing, 2019, pp. 48–69.

[13] M. Benedikt, G. Puppis, and C. Riveros, "Regular repair of specifications," in *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE, 2011, pp. 335–344.

[14] R. Bloem, K. Chatterjee, and B. Jobstmann, "Graph games and reactive synthesis," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 921–962. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_27

[15] L. Brim, J. Chaloupka, L. Doyen, R. Gentilini, and J.-F. Raskin, "Faster algorithms for mean-payoff games," *Formal methods in system design*, vol. 38, no. 2, pp. 97–118, 2011.

[16] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 137–150. [Online]. Available: https://doi.org/10.1145/2103656.2103674

[17] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," http://rise4fun.com/Bek/, 2012.

[19] R. Alur, "Streaming string transducers," in *Logic, Language, Information and Computation*, L. D. Beklemishev and R. de Queiroz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–1.

[20] O. Markgraf, C.-D. Hong, A. W. Lin, M. Najib, and D. Neider, "Parameterized synthesis with safety properties," in *Asian Symposium on Programming Languages and Systems*. Springer, 2020, pp. 273–292.

[21] A. W. Lin and P. Rümmer, "Liveness of randomised parameterised systems under arbitrary schedulers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 112–133.

[22] D. Neider, "Computing minimal separating dfas and regular invariants using sat and smt solvers," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2012, pp. 354–369.

[23] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and M. Stoelinga, "Resource interfaces," in *Embedded Software, Third International Conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003, Proceedings*, 2003, pp. 117–133. [Online]. Available: https://doi.org/10.1007/978-3-540-45212-6_9