Decision Procedures for Composition and Equivalence of Symbolic Finite State Transducers

Margus Veanes
Microsoft Research, Redmond
margus@microsoft.com

David Molnar
Microsoft Research, Redmond
dmolnar@microsoft.com

Benjamin Livshits
Microsoft Research, Redmond
livshits@microsoft.com

ABSTRACT

Finite automata model a wide array of applications in software engineering, from regular expressions to specification languages. Finite transducers are an extension of finite automata to model functions on lists of elements, which in turn have uses in fields as diverse as computational linguistics and model-based testing. Symbolic finite transducers are a further generalization of finite transducers where transitions are labeled with formulas in a given background theory. Compared to classical finite transducers, symbolic transducers are far more succinct in the case of finite alphabets, because they have no need to enumerate all cases of a transition; symbolic transducers can also use theories, such as the theory of linear arithmetic over integers or reals, with infinite alphabets.

Given a decision procedure for the background theory, we show novel algorithms for composition and equivalence checking for a large class of symbolic finite transducers, namely the class of single-valued transducers. Our algorithms give rise to a complete decidable algebra of symbolic transducers. Unlike previous work, we do not need any syntactic restriction of the formulas on the transitions, only a decision procedure; in practice we leverage recent advances in satisfiability modulo theory (SMT) solvers. We show how to decide single-valuedness, which means that symbolic finite transducers arising from practice can be checked to see if our algorithms apply. Our base algorithms are unusual in that they are nonconstructive, so we exhibit a separate model generation algorithm that can quickly find counterexamples in the case two symbolic finite transducers are not equivalent.

As a key example, string manipulation is a particularly important application of our theoretical results with immediate benefits to bug finding. Our work makes symbolic finite transducers a practical approach for software engineering applications, such as the analysis of security-critical sanitization functions in web pages and model based testing.

Microsoft Research Technical Report
MSR-TR-2011-32
March 14, 2011

1. INTRODUCTION

Finite automata are used in a wide range of applications in software engineering, from regular expressions to specification languages. Nearly every programmer has used a regular expression at one point or another to parse logs or manipulate text. Finite transducers are an extension of finite automata to model functions on lists of elements, which in turn have uses in fields as diverse as computational linguistics and model-based testing. While these objects are of immense practical use, they suffer from certain drawbacks: in the presence of large alphabets, they can "blow up" in the number of states, as each transition can encode only one choice of element from the alphabet. Furthermore, their most common forms cannot handle infinite alphabets.

Symbolic finite transducers are an extension of traditional transducers that attempt to solve these problems by allowing transitions to be labeled with arbitrary formulas in a specified theory. While the concept is straightforward, traditional algorithms for deciding composition, equivalence, and other properties of finite transducers *do not* immediately generalize to the symbolic case. In particular, previous work on symbolic finite transducers has needed to impose syntactic restrictions on formulas to achieve decidable analysis. Our work breaks this barrier and allows for arbitrary formulas from *any decidable background theory.* In practice, we leverage the recent progress in satisfiability modulo theory (SMT) solvers to provide this decision procedure. We find that our algorithms are fast when used with Z3, a state of the art SMT solver.

The restriction we do make is a semantic one: that the symbolic finite transducer is *single-valued.* This restriction is needed because equivalence is undecidable even for standard finite transducers. We show that the single-valuedness property is decidable for symbolic finite transducers. This gives us a way to check transducers arising from practical applications before applying our algorithms.

While it was previously known that equivalence was decidable for single-valued transducers, again it does not immediately follow that equivalence should be decidable for single-valued *symbolic* finite transducers because typically even very restricted extensions of finite automata and finite transducers lead to undecidability of the core decision problems. In fact, our proof requires a delicate separation between the "automata theoretic" parts of our algorithms and the use of the decision procedure. Unusually, our algorithm for deciding equivalence is *nonconstructive*: while we can determine that two symbolic finite automata are not equivalent, our proof does not provide a way to find a counterexample. Fortunately, we provide a separate *model generation* algorithm that can find counterexamples once it is known that two automata are not equivalent.

Overall, our algorithms enable the use of symbolic finite automata and transducers as first class objects for designing new program analyses, just as in the case of regular expressions and traditional finite automata.

1.1 Applications

Below we describe specific applications that would benefit from this work, but we expect there are many more to follow.

- **Security sanitizer checking.** Our algorithms are used in the BEK project, which consists of a new domain specific language and analysis engine for "sanitizer" routines in web applications. A sanitizer is a string to string transformation that attempts to remove "bad" elements from data contributed to an application by an untrusted user. For example, with Web applications typically we want to remove strings that may be interpreted by the browser as JavaScript.
- **List comprehension programs.** Our approach allows for analysis of programs that iterate over lists of elements, such as string transformations or graphics transformations. We can express these programs as symbolic finite transducers, then combine them using our new algorithms.
- **Integrated decision procedure for SMT solvers.** We can apply our algorithms to SMT solvers to yield symbolic finite transducers as a first class decidable theory for future SMT solvers.
- **Symbolic model analysis in the context of model-based testing.** Symbolic transducers allow us to represent side effects from calling APIs or functions. The symbolic formulas in turn let us represent potential constraints on the arguments to such APIs without leading to a huge blowup in the size of the transducer.
- **Natural language processing.** Finite transducers were studied extensively in the context of natural language processing. Symbolic finite transducers allow for the use of infinite alphabets.

1.2 Contributions

Our contributions are the following:

- We show that single-valued symbolic finite transducers can be used for software engineering applications, by giving practical algorithms for manipulating such transducers.
- In particular, we give novel algorithms for *composition* and *equivalence checking* of symbolic finite transducers. Our algorithms, unlike previous work, make no restrictions on the formulas used in the transducers. Instead, we require only a decision procedure for the background theory.
- We show that the single-valuedness property of symbolic transducers is decidable. This gives rise to a decidable complete algebra of symbolic transducers.
- We outline how our new algorithms for symbolic finite transducers apply to practical software engineering problems.

```
private static string EncodeHtml(string t)
{
    if (t == null) { return null; }
    if (t.Length == 0) { return string.Empty; }
    StringBuilder builder =
         new StringBuilder("", t.Length * 2);
    foreach (char c in t)
    {
      if ((((c > '`') && (c < '{')) ||
      ((c > '@') && (c < '['))) || (((c == ' ') ||
      ((c > '/') && (c < ':'))) || (((c == '.') ||
      (c == ',')) || ((c == '-') || (c == '_'))))){
        builder.Append(c);
      } else {
        builder.Append("&#" +
          ((int) c).ToString() + ";");
      }
    }
    return builder.ToString();
}
```

Figure 1: Code for `AntiXSS.EncodeHtml` from version 2.0 in C#.

2. MOTIVATING EXAMPLE

Figure 1 shows code in C# for `EncodeHtml`, an example of a string-to-string transformation that is used in real web services on data from untrusted users. This routine is designed as a protection against cross-site scripting attacks (XSS), in which a malicious user submits data to the web service that may later be mistakenly interpreted as JavaScript code by another user's browser. The goal of this function is to escape all characters that might lead the browser to start executing JavaScript.

Example 1. We show how `EncodeHtml` can be translated to a symbolic finite transducer. Let $\varphi[\ell]$ be the following formula, encoding the `if` condition in Figure 1:

$$(\text{`a'} \le \ell \le \text{`z'}) \lor (\text{`A'} \le \ell \le \text{`Z'}) \lor (\text{`0'} \le \ell \le \text{`9'}) \lor \\ \ell = \text{`\ '} \lor \ell = \text{`.'} \lor \ell = \text{`,'} \lor \ell = \text{`-'} \lor \ell = \text{`\_'}$$

The symbolic finite transducer has a single state because no Boolean variables change inside the body of the `foreach` statement. The `if` statement in the original function translates into two self-loops. In our encoding below, $\mathbf{d}_i(\ell)$ is the map from character ℓ to its i'th decimal digit of the integer encoding of ℓ, modeling the C# code `((int)c).ToString()` in `EncodeHtml`.

The resulting SFT contains the following six transitions:

$$q \xrightarrow{\varphi/[\ell]} q$$

$$(0{\le}n{\le}4) \quad q \xrightarrow{\neg\varphi[\ell]\wedge 10^n \le \ell < 10^{n+1}/[\text{`\&'},\text{`\#'},\mathbf{d}_n(\ell),\ldots,\mathbf{d}_0(\ell),\text{`;'}]} q \quad (1)$$

where

$$\mathbf{d}_i(\ell) \overset{\text{def}}{=} ((\ell \div 10^i)\%10) + 48$$

is a term in linear (bitvector) arithmetic computing the i'th decimal digit of ℓ (as a character), where $\div$ is bitvector division, $+$ is bitvector addition, and $\%$ computes the remainder after dividing its first operand by its second. The collection of five transitions in (1) corresponds to the program statement `"&#" + ((int) c).ToString() + ";"`. Note that the

SFT obtained from `EncodeHtml` is deterministic and thus single-valued because all the transition formulas are mutually exclusive. ⊠

As an example of applying this transducer, consider the input `"c&e"` . Because '`&`' = 38 and $\varphi[38]$ does not hold, it follows that

$$\mathbf{T}_{EncodeHtml}(\texttt{"c\&e"}) = \{\texttt{"c\&\#38;e"}\}.$$

Once the SFT representation of this function is produced, we can proceed to test this implementation to determine if

- it is idempotent, which is important because web developers may accidentally apply a sanitizer multiple times to untrusted data;
- if it commutes with other sanitizers used in the program, which is important because programs may apply multiple different sanitizers to data [35];
- and, most importantly, given a target dangerous output, is it possible to produce an input that would generate it. This allows checking whether known "dangerous" strings are properly ruled out by the sanitizer or not.

3. PRELIMINARIES

We use basic notions from classical automata theory [21], classical logic, and model theory [17]. Our notions regarding finite state transducers are consistent with [46] and [14] regarding the view of finite state transducers as finite state tree transducers over linear trees.

3.1 Background universe

We work modulo a *background* structure $\mathcal{U}$ over a language that is multi-sorted and write $\mathcal{U}$ also for the universe or domain of $\mathcal{U}$. For each sort σ, $\mathcal{U}^\sigma$ denotes a nonempty subdomain of $\mathcal{U}$. There is a basic Boolean sort BOOL, $\mathcal{U}^{\text{BOOL}} = \{\mathit{true}, \mathit{false}\}$, and the standard logical connectives are assumed to be part of the background. *Terms* are defined by induction as usual and are assumed to be well-sorted. Function symbols with range sort BOOL are called relation symbols. Boolean terms are called formulas or predicates. A term without free variables is *ground*. Elements of $\mathcal{U}$ are also considered as ground terms. A term t of sort σ is indicated by $t : \sigma$. Given a term t and a substitution θ from variables to terms, $\mathit{Subst}(t, \theta)$ or $t\theta$ denotes the term resulting from applying the substitution θ to t. An *if-then-else* term $\mathit{Ite}(\varphi, t_1{:}\sigma, t_2{:}\sigma)$ equals t_1 if φ is true; it equals t_2 otherwise.

We also use the basic sorts INT of integers, BV^n of *n-bit-vectors*, for $n \geq 1$ and $\text{TUPLE}\langle\sigma_0, \ldots, \sigma_{n-1}\rangle$, for $n \geq 1$, of *n-tuples* of elements of basic sorts σ_i for $i < n$. All sorts are associated with *built-in* (predefined) functions and built-in theories. For example, there is a built-in Boolean function (predicate) $< : \text{BV}^7 \times \text{BV}^7 \to \text{BOOL}$ that provides a strict total order of all 7-bit-vectors that is assumed to match with the standard lexicographic order over ASCII characters. For each n-tuple sort there is a constructor and a projection function $\pi_i : \text{TUPLE}\langle\sigma_0, \ldots, \sigma_{n-1}\rangle \to \sigma_i$, for $i < n$, that projects the i'th element from an n-tuple.

For each sort σ, $\text{LIST}\langle\sigma\rangle$ is the *list sort with element sort* σ. Lists are algebraic data types. There is an empty list $\epsilon : \text{LIST}\langle\sigma\rangle$ and for all $e : \sigma$ and $l : \text{LIST}\langle\sigma\rangle$, $[e\,|\,l] : \text{LIST}\langle\sigma\rangle$. The accessors are $\mathit{hd} : \text{LIST}\langle\sigma\rangle \to \sigma$ and $\mathit{tl} : \text{LIST}\langle\sigma\rangle \to \text{LIST}\langle\sigma\rangle$ with their usual meaning. We also adopt the convention that $[a, b, c]$ stands for the list $[a \mid [b \mid [c \mid \epsilon]]]$ and we write $l_1 \cdot l_2$ for the concatenation of l_1 with l_2.

Lists of a given element sort σ are also called *words* and the list elements are called *characters* when the sort σ is basic or corresponds to a basic sort. In the context of string analysis, characters have typically the sort BV^n for some fixed $n > 0$, e.g., $n = 7$ if words represent strings of ASCII characters. Constant characters are written as '`a`' assuming for example ASCII encoding. When σ is a sort for characters that is clear from the context, we use the standard string notation `"abc"` for ['`a`', '`b`', '`c`']. In general however, characters may have compound basic sorts such as $\text{TUPLE}\langle\text{BV}^7, \text{BV}^7, \text{BOOL}\rangle$, while, e.g., lists of unbounded length are not considered as characters.

3.2 Finite state transducers

A finite transducer is a generalization of a Mealy machine that, in addition to its input and output symbols, has the special symbol ϵ denoting the empty word making it possible to omit characters in the input and output words. We start with the classical definition of *finite transducers*. The particular subclass of finite transducers that we are considering here are also called *generalized sequential machines* or GSMs [46], however, this definition is not standardized in the literature, and we therefore continue to say finite transducers for this restricted case. The restriction is that, GSMs read one symbol at each transition, while a more general definition allows transitions that may skip inputs.

Definition 1. An *input-ϵ-free finite transducer* A is defined as a six-tuple $(Q, q^0, F, \Sigma, \Gamma, \Delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, Σ is the *input alphabet*, Γ is the *output alphabet*, and Δ is the *transition function* from $Q \times \Sigma$ to $2^{Q\times\Gamma^*}$.

In this paper we say *finite transducer* for input-ϵ-free finite transducer. We indicate a component of a finite transducer A by using A as a subscript. For $(q, v) \in \Delta_A(p, a)$ we define the notation $p \xrightarrow{a/v}_A q$, where $p, q \in Q_A$, $a \in \Sigma_A$ and $v \in \Gamma_A^*$. We write $p \xrightarrow{a/v} q$ when A is clear from the context. Given words v and w we let $v \cdot w$ denote the concatenation of v and w. Note that $v \cdot \epsilon = \epsilon \cdot v = v$.

Given $q_i \xrightarrow{a_i/v_i}_A q_{i+1}$ for $i < n$ we write $q_0 \xrightarrow{u/v}_A q_n$ where $u = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1}$ and $v = v_0 \cdot v_1 \cdot \ldots \cdot v_{n-1}$. We write also $q \xrightarrow{\epsilon/\epsilon}_A q$. A induces the *transduction* or *transformation*, $\mathbf{T}_A : \Sigma_A^* \to 2^{\Gamma_A^*}$:

$$\mathbf{T}_A(u) \stackrel{\text{def}}{=} \{v \mid \exists q \in F_A\, (q_A^0 \xrightarrow{u/v} q)\}$$

We lift the definition to sets, $\mathbf{T}_A(U) \stackrel{\text{def}}{=} \bigcup_{u\in U} \mathbf{T}_A(u)$.

We say that A is *deterministic* if it has no two transitions with the same source state and same label but different outputs or target states. (While weaker definitions of deterministic finite state transducers exist, the given definition is consistent with [14].)

The following subclass of finite transducers plays a central role in the section that introduces a decision procedure for equivalence of symbolic finite transducers.

Definition 2. A is *single-valued* if $(\forall u \in \Sigma_A^*)|\mathbf{T}_A(u)| \leq 1$.

Note that determinism implies single-valuedness, while the converse is not true. Moreover, there exist nondeterministic single-valued finite state transducers without any equivalent deterministic finite state transducer.

4. SYMBOLIC FINITE TRANSDUCERS

In this section we describe an extension of finite transducers through a symbolic representation of labels by predicates. The advantage of the extension is succinctness and modularity with respect to the background theory of labels. It naturally separates the finite state transition graph from the theory of labels.

In the following let $\ell{:}\sigma$ be a *fixed* variable of sort σ. We refer to $\ell{:}\sigma$ as the *input variable* (for the given sort σ). $\mathit{Term}^{\sigma}(\bar{x})$ denotes the set of terms of sort σ with free variables contained in $\bar{x}$ and $\mathit{Pred}(\bar{x}) \stackrel{\mathrm{def}}{=} \mathit{Term}^{\mathrm{BOOL}}(\bar{x})$.

Definition 3. A *Symbolic Finite Transducer (SFT)* is a six-tuple $(Q, q^0, F, \sigma, \gamma, \delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, σ is the *input sort*, γ is the *output sort*, and δ is the finite *symbolic transition function* from $Q \times \mathit{Pred}(\ell)$ to subsets of $Q \times \mathit{Term}^{\mathrm{LIST}\langle\gamma\rangle}(\ell)$.

We use the notation $p \xrightarrow{\varphi/\mathbf{u}}_A q$ for $(q, \mathbf{u}) \in \delta_A(p, \varphi)$ and call $p \xrightarrow{\varphi/\mathbf{u}}_A q$ a *symbolic transition*, $\varphi/\mathbf{u}$ is called its *label*, φ is called its *input (guard)* and $\mathbf{u}$ its *output*.

An SFT $A = (Q, q^0, F, \sigma, \gamma, \delta)$ denotes the finite state transducer

$$[\![A]\!] \stackrel{\mathrm{def}}{=} (Q, q^0, F, \mathcal{U}^{\sigma}, \mathcal{U}^{\gamma}, \cup\{[\![\rho]\!] \mid \rho \in \delta_A)$$

where

$$[\![p \xrightarrow{\varphi[\ell]/\mathbf{u}[\ell]}_A q]\!] \stackrel{\mathrm{def}}{=} \{p \xrightarrow{a/\mathbf{u}[a]}_{[\![A]\!]} q \mid a \in \mathcal{U}^{\sigma},\ \varphi[a] \text{ is true}\}$$

Note that $[\![\rho]\!]$ may be infinite when $\mathcal{U}^{\iota}$ is infinite. Thus, we allow infinite alphabets in $[\![A]\!]$. For an SFT A let the underlying *transduction* $\mathbf{T}_A$ be $\mathbf{T}_{[\![A]\!]}$. For a state $q \in Q_A$ let $\mathbf{T}^q_A(v)$ denote the set of outputs generated from state q from the enabled input v. An SFT A is *single-valued* (resp. *deterministic*) if $[\![A]\!]$ is single-valued (resp. deterministic).

Note also that a transition $p \xrightarrow{\psi/\mathit{Ite}(\varphi,t,u)} q$ is equivalent to having two transitions $p \xrightarrow{\psi\wedge\varphi/t} q$ and $p \xrightarrow{\psi\wedge\neg\varphi/u} q$. We make use of this *if-then-else* representation in our examples.

In the following examples, all SFTs are single-valued. The first example illustrates a few simple functional list transformations, expressed as deterministic SFTs that illustrate how global properties of SFTs depend on the label theory.

Example 2. Let the input sort and the output sort be INT. All SFTs have a single state here. *Negate* multiplies all elements by -1. *Increment* adds 1 to each element. *DeleteZeros* deletes all zeros from the input.

$$\begin{aligned}
\delta_{\mathit{Negate}} &= \{p \xrightarrow{\mathit{true}/[-\ell]} p\} \\
\delta_{\mathit{Increment}} &= \{q \xrightarrow{\mathit{true}/[1+\ell]} q\} \\
\delta_{\mathit{DeleteZeros}} &= \{r \xrightarrow{\ell=0/[]} r,\quad r \xrightarrow{\ell\neq 0/[\ell]} r\}
\end{aligned}$$

Properties such as commutativity and idempotence of SFTs depend on the theory of labels. For example properties like, *Negate* and *DeleteZeros* commute, *DeleteZeros* is idempotent, and *Negate* and *Increment* do not commute, depend on properties of integer addition and multiplication. Note that none of the examples can be expressed as traditional finite state transducers over a finite alphabet. ⊠

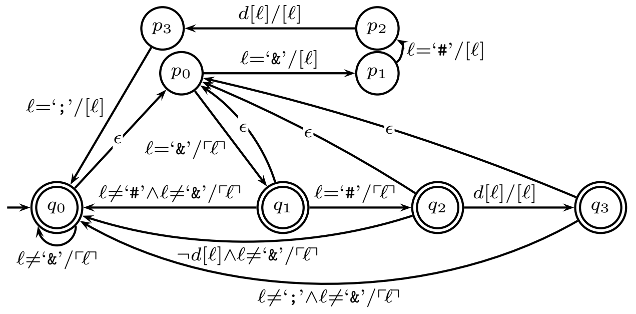
Example 1 demonstrates a more involved use of the label theory. It shows a precise representation of the sanitizer `AntiXSS.EncodeHtml` from version 2.0 of the Microsoft AntiXSS library. The sanitizer is shown in Figure 1. The sanitizer applies HtmL encoding to an input string of Unicode characters: for each character ℓ, either ℓ is kept unchanged or encoded through numeric escaping.

It is well-known that nondeterministic finite state transducers are more expressive than deterministic finite state transducers. The following example illustrates a class of common list transformations when a deterministic SFT does not exist if $\mathcal{U}^{\iota}$ is infinite or is up to $|\mathcal{U}^{\iota}|$ times larger than the equivalent nondeterministic SFT.

Example 3. The example illustrates an SFT *EncodeHtml2* that transforms an input string similar to *EncodeHtml* except that any substring of the input string that matches the regular expression P = `&#[0-9];` is mapped to itself. In other words, double-encoding of some of the already encoded characters is avoided. It is straightforward to generalize the pattern P so that it characterizes all the possible cases of the encoded outputs, but such a generalization would make the example unnecessarily hard to follow. Let $\ulcorner\ell\urcorner$ be the following term, where $\varphi[\ell]$ and $\mathbf{d}_0(\ldots)$ are defined in Example 1,

$$\ulcorner\ell\urcorner \stackrel{\mathrm{def}}{=} \mathit{Ite}(\varphi[\ell], [\ell], \mathit{Ite}(0 \le \ell < 10, [\text{`\texttt{\&}'}, \text{`\texttt{\#}'}, \mathbf{d}_0(\ell), \text{`\texttt{;}'}], \ldots))$$

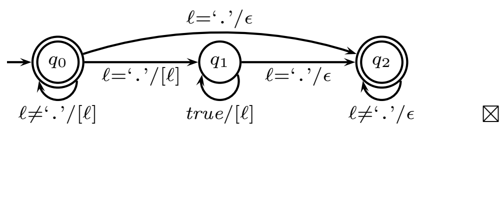
Thus $\ulcorner\ell\urcorner$ is a single term describing concisely all the possible character encodings of *EncodeHtml* and aids in visualizing *EncodeHtml2* as the following SFT. Let $d[\ell]$ be the formula '0' $\le \ell \wedge \ell \le$ '9'.


p_3
p_2
p_1
p_0
$d[\ell]/[\ell]$
ℓ='#'/$[\ell]$
ℓ='&'/$[\ell]$
ℓ=';'/$[\ell]$
ϵ
ℓ='&'/$\ulcorner\ell\urcorner$
ϵ
ϵ
ϵ
q_0
q_1
q_2
q_3
$\ell\neq$'#'$\wedge\ell\neq$'&'/$\ulcorner\ell\urcorner$
ℓ='#'/$\ulcorner\ell\urcorner$
$d[\ell]/[\ell]$
$\ell\neq$'&'/$\ulcorner\ell\urcorner$
$\neg d[\ell]\wedge\ell\neq$'&'/$\ulcorner\ell\urcorner$
$\ell\neq$';'$\wedge\ell\neq$'&'/$\ulcorner\ell\urcorner$

For P = `&#[0-9]{0,4};` there would be eight more states, while a deterministic version would need a state to remember each concrete path in P in order to determine if that path ends with '`;`' before knowing what to output, thus it would have in the order of 10^5 more states. ⊠

The final example illustrates a case when a deterministic SFT does not exist independent of the size of the alphabet. The example illustrates a common class of list transformations referring to the occurrences of certain elements or patterns in the input relative to the end of the list.

Example 4. Let *UptoLastDot* be an SFT that deletes all characters after (including) the last occurrence of '`.`'. For example $\mathbf{T}_{\mathit{UptoLastDot}}$(`"www.abc.org"`) = {`"www.abc"`}.



⊠

5. SFT ALGORITHMS

In this section we study algorithms for *composition* and *equivalence* of SFTs. First, we prove that SFTs are closed under composition and give a practical composition algorithm for SFTs. Next, we provide an efficient equivalence algorithm for single-valued SFTs that uses as an oracle a constraint solver for an *arbitrary* theory of input labels, where a typical theory of labels is quantifier free integer linear arithmetic, but it could also be linear arithmetic over rationals, or bitvectors. The immediate implications of these two algorithms are decision procedures for *commutativity* and *idempotence* of single-valued SFTs.

In the algorithms we make use of the definitions

$$\begin{aligned}
\mathit{Source}(p \xrightarrow{\varphi/\mathbf{u}} q) &\stackrel{\text{def}}{=} p,\\
\mathit{Target}(p \xrightarrow{\varphi/\mathbf{u}} q) &\stackrel{\text{def}}{=} q,\\
\mathit{Guard}(p \xrightarrow{\varphi/\mathbf{u}} q) &\stackrel{\text{def}}{=} \varphi,\\
\mathit{Out}(p \xrightarrow{\varphi/\mathbf{u}} q) &\stackrel{\text{def}}{=} \mathbf{u},\\
\delta_A(q) &\stackrel{\text{def}}{=} \{\rho \in \delta_A \mid \mathit{Source}(\rho) = q\}.
\end{aligned}$$

5.1 Composition of SFTs

Given two transductions $\mathbf{T}_1$ and $\mathbf{T}_2$, $\mathbf{T}_1 \circ \mathbf{T}_2$ denotes the transduction that maps an input word u to the set $\mathbf{T}_2(\mathbf{T}_1(u))$. This definition is consistent with [14]. Notice that $\circ$ applies first $\mathbf{T}_1$, then $\mathbf{T}_2$, contrary to how $\circ$ is used for standard function composition. Note that single-valuedness is trivially preserved by composition.

Let A and B be finite transducers. A fundamental composition of A and B is the *join composition* of A and B.

Definition 4. The *(join) composition of A and B* is the finite transducer

$$A \circ B \stackrel{\text{def}}{=} (Q_A \times Q_B, (q_A^0, q_B^0), F_A \times F_B, \Sigma_A, \Gamma_B, \Delta_{A\circ B})$$

where, for all $(p,q) \in Q_A \times Q_B$ and $a \in \Sigma_A$:

$$\begin{aligned}
\Delta_{A\circ B}((p,q),a) \stackrel{\text{def}}{=} & \{((p',q),\epsilon) \mid p \xrightarrow{a/\epsilon}_A p'\}\\
& \cup \{((p',q'),v) \mid (\exists u \in \Gamma_A^+)\\
& \quad p \xrightarrow{a/u}_A p',\ q \xrightarrow{u/v}_B q'\}
\end{aligned}$$

It follows easily from the definition that $\mathbf{T}_{A\circ B} = \mathbf{T}_A \circ \mathbf{T}_B$.

Let now A and B be fixed SFTs and assume that $\rho = \gamma_A = \sigma_B$. The join composition algorithm constructs an SFT $A \circ B$ such that $\mathbf{T}_{A\circ B} = \mathbf{T}_A \circ \mathbf{T}_B$.

The algorithm is shown in Figure 2. The algorithm uses a procedure $\mathit{GetPaths}(\varphi, \mathbf{u}, q, B)$ that is essentially a backtracking search procedure over feasible paths. Given a symbolic label $\varphi/\mathbf{u}$ of A, and a state q of B, it returns the collection of all joined paths in B of length $|\mathbf{u}|$ that start from p and whose guards are feasible for corresponding members of $\mathbf{u}$. For example, suppose

$$p_1 \xrightarrow{\varphi/[u_1,u_2]}_A q_1, \quad p_2 \xrightarrow{\psi_1/[v_1]}_B p' \xrightarrow{\psi_2/[v_2]}_B q_2$$

then (given $\theta_i = \{\ell \mapsto u_i\}$ for $i = 1, 2$),

$$(p_1, p_2) \xrightarrow{\varphi\wedge\psi_1\theta_1\wedge\psi_2\theta_2/[v_1\theta_1, v_2\theta_2]}_{A\circ B} (q_1, q_2)$$

Example 5. Consider the SFTs $A = \mathit{Negate}$ and $B = \mathit{DeleteZeros}$ from Example 2. We construct $A\circ B$. There is a single pair state (p, r) and a single transition $p \xrightarrow{\mathit{true}/[-\ell]}_A p$. $\mathit{GetPaths}(\mathit{true}, [-\ell], r, B)$ returns

$$\{([], -\ell = 0, r),\ ([-\ell], -\ell \neq 0, r)\}$$

Thus, there are two transitions in $A \circ B$:

$$\delta_{A\circ B} = \{(p,r) \xrightarrow{-\ell=0/[]} (p,r), \quad (p,r) \xrightarrow{-\ell\neq 0/[-\ell]} (p,r)\}.$$

If we construct $B\circ A$ instead, we start with the B transitions and we end up with transitions

$$\delta_{B\circ A} = \{(r,p) \xrightarrow{\ell=0/[]} (r,p), \quad (r,p) \xrightarrow{\ell\neq 0/[-\ell]} (r,p)\}.$$

Note that $A \circ B$ and $B \circ A$ are equivalent in this case. ⊠

```
GetPaths(φ, u, q, B) ≝
1  if $\mathbf{u} = []$ yield $([], \varphi, q)$;
2  else foreach $tr \in \delta_B(q)$
3    let $\varphi_1 = \varphi \wedge \mathit{Subst}(\mathit{Guard}(tr), \{\ell \mapsto hd(\mathbf{u})\})$;
4    if $\mathit{IsSat}(\varphi_1)$
5      foreach $(y, \psi, p)$ in $\mathit{GetPaths}(\varphi_1, tl(\mathbf{u}), \mathit{Target}(tr), B)$
6        yield $(\mathit{Subst}(\mathit{Out}(tr), \{\ell \mapsto hd(\mathbf{u})\}) \cdot y, \psi, p)$;
```

```
A ∘ B ≝
1  let $q^0 = (q_A^0, q_B^0)$; $Q = \{q^0\}$; $\delta = \emptyset$;
2  let $S$ be a stack with initial element $q^0$;
3  while $S$ is nonempty
4    pop $p = (p_1, p_2)$ from $S$;
5    foreach $p_1 \xrightarrow{\varphi/\mathbf{u}} q_1$ in $\delta_A(p_1)$
6      foreach $(\mathbf{v}, \psi, q_2)$ in $\mathit{GetPaths}(\varphi, \mathbf{u}, p_2, B)$
7        let $q = (q_1, q_2)$;
8        add $p \xrightarrow{\psi/\mathbf{v}} q$ to $\delta$;
9        if $q \notin Q$ then add $q$ to $Q$ and push $q$ to $S$;
10 end of while;
11 let $F = \{(q_1, q_2) \in Q \mid q_1 \in F_A \wedge q_2 \in F_B\}$;
12 eliminate states in $Q \setminus \{q^0\}$ that do not reach $F$;
13 return $(Q, q_0, F, \sigma_A, \gamma_B, \delta)$;
```

Figure 2: Join composition algorithm for SFTs.

GetPaths uses satisfiabilty checking to yield only those paths for which the corresponding output from A, when used as input of B, does not cause the resulting guard to become unsatisfiable. If the satisfiability check is removed, the procedure will still be correct in the context of the join algorithm, but this may cause a combinatorial explosion of the paths, as would happen in the next example.

Example 6. Consider *EncodeHtml* from Example 1, let $A = B = \mathit{EncodeHtml}$ but rename q to p in A. We construct $A\circ B$. There is only a single pair state (p, q). First, consider the transition

$$p \xrightarrow{\varphi[\ell]/[\ell]}_A p$$

There is only one transition in B that does not cause the composed guard to be unsatisfiable, the composed transition is (after simplifying the formulas):

$$(p,q) \xrightarrow{\varphi[\ell]/[\ell]}_{A\circ B} (p,q)$$

Next, consider the transition

$$p \xrightarrow{\neg\varphi[\ell]\wedge 0\leq\ell<10/[\text{‘\&’},\text{‘\#’},\mathbf{d}_0(\ell),\text{‘;’}]}_{A} p$$

In B we need to consider all paths of length 4 and there are a total of 6^4 such paths (since B has 6 transitions from q to q). Only one of these paths is possible. In particular, for all ℓ and i, $\varphi[\mathbf{d}_i(\ell)]$ holds because ‘0’ $\leq \mathbf{d}_i(\ell) \leq$ ‘9’, while for $\ell \in \{\text{‘\&’},\text{‘\#’},\text{‘;’}\}$, $\neg\varphi[\ell] \wedge 10 \leq \ell < 100$ holds, and thus double-encoding occurs for these characters in $A \circ B$.

$$(p,q) \xrightarrow{\neg\varphi[\ell]\wedge 0\leq\ell<10\wedge\varphi[\mathbf{d}_0(\ell)]/\texttt{"\&\#38;\&\#35;"}\cdot[\mathbf{d}_0(\ell)]\cdot\texttt{"\&\#59;"}}_{A\circ B} (p,q)$$

The remaining cases are similar. The benefit of early pruning of the search space using satisfiability checks is obvious in this example. $\boxtimes$

Theorem 1. $\mathbf{T}_{A\circ B} = \mathbf{T}_A \circ \mathbf{T}_B$.

Proof. First note that the satisfiability check in the composition algorithm removes transitions that are infeasible and therefore do not affect the transduction. We can also ignore states in $A \circ B$ that are not reachable from the initial state, and states that do not reach a final state. Suppose (p_1, p_2) is reachable from the initial state (q^0_A, q^0_B). The definition of *GetPaths* follows exactly the construction of the composed transitions in Definition 4 from (p_1, p_2), which implies the theorem together with the main while-loop in the join algorithm, that considers all possible (p_1, p_2) that are reachable from the initial state by virtue of DFS. $\square$

5.2 Equivalence of SFTs

We introduce an algorithm for deciding equivalence of single-valued SFTs. While general equivalence of finite state transducers is undecidable [16] the undecidability is caused by allowing unboundedly many different outputs for a given input. The case that is practically most relevant for us is when transducers are single-valued, since this case corresponds closely to functional transformations over lists computed by concrete programs. As illustrated above, this does (in general) not rule out nondeterministic SFTs, i.e., SFTs whose underlying finite automaton is nondeterministic. This is important because, several useful single-valued transductions, are either not expressible as deterministic SFTs (e.g. $\mathbf{T}_{\mathit{UptoLastDot}}$) or cause a blowup in the size of the SFT (e.g. $\mathbf{T}_{\mathit{EncodeHtml2}}$).

In the following let A and B be two SFTs such that $\sigma = \sigma_A = \sigma_B$ and $\gamma = \gamma_A = \gamma_B$. A and B are *equivalent* if $\mathbf{T}_A = \mathbf{T}_B$. Let

$$\mathbf{D}(A) \overset{\text{def}}{=} \{v \mid \mathbf{T}_A(v) \neq \emptyset\}.$$

Checking equivalence of A and B reduces to two independent tasks:

Domain equivalence : $\mathbf{D}(A) = \mathbf{D}(B)$.

Partial equivalence : $(\forall v \in \mathbf{D}(A)\cap\mathbf{D}(B))\ \mathbf{T}_A(v)=\mathbf{T}_B(v)$

Checking domain-equivalence is decidable for all SFTs over a decidable label background. This follows from results known for *symbolic finite automata* or *SFAs* that generalize finite automata by allowing predicates as labels.

Functional compatibility $(A \overset{1}{=} B)$:

$$\forall x\, y\, z\, ((y \in \mathbf{T}_A(x) \wedge z \in \mathbf{T}_B(x)) \Rightarrow y = z)$$

Functional compatibility, or simply compatibility, is a weak form of partial equivalence. The following properties of compatibility follow directly from the definitions.

Proposition 1. *A is single-valued iff $A \overset{1}{=} A$. If A and B are single-valued then $A \overset{1}{=} B$ iff A and B are partially equivalent.*

The *product* $A \times B$ of A and B is a *2-output-SFT* that is a variation of the product of SFAs [41]. The product $A \times B$ is defined as the least fixpoint of pair states $Q \subseteq Q_A \times Q_B$ and transitions of $A \times B$ under the following conditions (e.g., by using DFS):

- $(q^0_A, q^0_B) \in Q$,
- if $(p_1, p_2) \in Q$, $p_1 \xrightarrow{\varphi/u}_A q_1$, and $p_2 \xrightarrow{\psi/v}_B q_2$, then
  - $(q_1, q_2) \in Q$ and
  - $(p_1, p_2) \xrightarrow{\varphi\wedge\psi/(u,v)}_{A\times B} (q_1, q_2)$,

  provided that *IsSat*$(\varphi \wedge \psi)$.

All *deadends*, states from which $F_A \times F_B$ is not reachable, are eliminated from $A \times B$. Given $(p, q) \in Q_{A\times B}$, we write $\mathbf{T}^{(p,q)}_A(v)$ for the set of outputs produced by A for the input v accepted from the state (p, q). Similarly for B. The following property of the construction follows from definitions.

$$\begin{array}{l}(\forall v \in \mathbf{D}(A) \cap \mathbf{D}(B))\\ \qquad \mathbf{T}_A(v) = \mathbf{T}_B(v) \Leftrightarrow \mathbf{T}^{(q^0_A,q^0_B)}_A(v) = \mathbf{T}^{(q^0_A,q^0_B)}_B(v)\end{array} \tag{2}$$

An *SFA* is an SFT such that all transitions have an empty output. We use the following definition that follows from the product construction.

Definition 5. Let A be an SFT and D an SFA. The *domain restriction* of A with respect to D, denoted by $A \restriction D$ is the SFT obtained from $A \times D$ by eliminating the second output component ϵ from the transitions.

The following property follows from the definitions.

$$\mathbf{T}_{A\restriction D}(v) = \begin{cases} \mathbf{T}_A(v), & \text{if } v \in \mathbf{D}(D);\\ \emptyset, & \text{otherwise.}\end{cases} \tag{3}$$

A *promise* of a state $(p, q) \in Q_{A\times B}$ is a pair of concrete lists (α, β) such that either $\alpha = \epsilon$ or $\beta = \epsilon$ and if (p, q) is reached from (q^0_A, q^0_B) on input v and matching output prefix w then the outputs from A and B so far are $w \cdot \alpha$ and $w \cdot \beta$ respectively. Intuitively, there is a promise to output α (resp. β) before any further output from A (resp. B).

Example 7. Let A and B be the following SFTs:

$$\begin{tikzcd} A: \arrow[r] & p_0 \arrow[rr, "0\leq\ell\leq 1/{[\ell,\ell,\ell]}"] & & p_1 \arrow[loop above, "0\leq\ell\leq 1/{[\ell]}"] \arrow[rr, "0\leq\ell\leq 1/{[\ell]}"] & & \boxed{p_2} \\ B: \arrow[r] & q_0 \arrow[rr, "1\leq\ell\leq 2/{[\ell]}"] & & q_1 \arrow[loop above, "1\leq\ell\leq 2/{[\ell]}"] \arrow[rr, "1\leq\ell\leq 2/{[\ell,\ell,\ell]}"] & & \boxed{q_2} \end{tikzcd}$$

Then $A \times B$ is the following 2-output-SFT. Note that other possible pair states are either not reachable (e.g. (p_0, q_1)) or deadends (e.g. (p_1, q_2)).

$$\begin{tikzcd}
{} \arrow[r] & (p_0, q_0) \arrow[r, "\ell{=}1/([\ell,\ell,\ell],[\ell])"] & (p_1, q_1) \arrow[loop above, "\ell{=}1/([\ell],[\ell])"] \arrow[r, "\ell{=}1/([\ell],[\ell,\ell,\ell])"] & \boxed{(p_2, q_2)}
\end{tikzcd}$$

One promise of the state (p_1, q_1) is the pair $([1, 1], \epsilon)$, for example for the input prefix $[1, 1, 1]$, both A and B have output $[1, 1, 1]$ but A still has the pending output $[1, 1]$. ⊠

Lemma 1. *If there exists a state in $A \times B$ that is reached with two different promises then $A \overset{1}{\neq} B$.*

Proof. Suppose we can reach a state (p, q) in $A \times B$ from the initial state first time with some input sequence z_1, common output o_1, and promise (α_1, β_1), and a second time with some input sequence z_2, common output o_2, and promise (α_2, β_2) such that $(\alpha_2, \beta_2) \neq (\alpha_1, \beta_1)$. Let w be any input sequence from (p, q) to a final state of $A \times B$, w exists because $A \times B$ has no deadends. Suppose (by way of contradiction) that $A \overset{1}{=} B$. It follows that

$$\begin{aligned}\mathbf{T}_A(z_1 \cdot w) = \{o_1 \cdot \alpha_1 \cdot o_3\} = \mathbf{T}_B(z_1 \cdot w) = \{o_1 \cdot \beta_1 \cdot o_4\}\\ \mathbf{T}_A(z_2 \cdot w) = \{o_2 \cdot \alpha_2 \cdot o_3\} = \mathbf{T}_B(z_2 \cdot w) = \{o_2 \cdot \beta_2 \cdot o_4\}\end{aligned}$$

for some $\{o_3\} = \mathbf{T}_A^{(p,q)}(w)$ and $\{o_4\} = \mathbf{T}_B^{(p,q)}(w)$. Note that if $|\mathbf{T}_A^{(p,q)}(w)| > 1$ or $|\mathbf{T}_B^{(p,q)}(w)| > 1$ then this would violate $A \overset{1}{=} B$ for example for the input $z_1 \cdot w$. So

$$\alpha_1 \cdot o_3 = \beta_1 \cdot o_4, \quad \alpha_2 \cdot o_3 = \beta_2 \cdot o_4.$$

We reach contradiction by case analysis.

- Case $\alpha_1 = \epsilon, \beta_1 = \epsilon, \alpha_2 \neq \epsilon, \beta_2 = \epsilon$. Then $o_3 = o_4$ and $\alpha_2 \cdot o_3 = o_4$, but this contradicts that $\alpha_2 \neq \epsilon$.
- Case $\alpha_1 \neq \epsilon, \beta_1 = \epsilon, \alpha_2 \neq \epsilon, \beta_2 = \epsilon$. Then $\alpha_1 \cdot o_3 = o_4$ and $\alpha_2 \cdot o_3 = o_4$, but this contradicts that $\alpha_1 \neq \alpha_2$.
- Case $\alpha_1 = \epsilon, \beta_1 \neq \epsilon, \alpha_2 \neq \epsilon, \beta_2 = \epsilon$. Then $o_3 = \beta_1 \cdot o_4$ and $\alpha_2 \cdot o_3 = o_4$, and thus $o_3 = \beta_1 \cdot \alpha_2 \cdot o_3$, but this contradicts that $\beta_1 \cdot \alpha_2 \neq \epsilon$.

The remaining cases are symmetrical. □

Lemma 1 is a key component in the main algorithm by providing an $O(|Q_{A\times B}|)$ search bound to detect if $A \overset{1}{\neq} B$. What is quite unusual about the lemma, which sets it apart from typical automata based properties, is that it is not constructive, it does not provide a concrete input witness v that shows $A \overset{1}{\neq} B$. The main algorithm is given in Figure 3. In the algorithm, the elements of S are the states still to be verified. Q is the map from reached states to promises associated with those states.

The following theorem states the correctness of the compatibility algorithm.

Theorem 2. *CheckCompatibility(A, B) fails iff $A \overset{1}{\neq} B$.*

Proof. For each state p, the while-loop verifies locally, that for any input enabled in p outputs will match up to maximum prefix of outputs from A and B, where p is associated with prior promises $Q(p) = (\alpha, \beta)$, where at least one of α or β is ϵ. The cases that cause violation of compatibility are the following, in the order of **FAIL**s:

```
CheckCompatibility(A, B) $\overset{\text{def}}{=}$
1  let $C = A \times B$;
2  let $Q = \{q_C^0 \mapsto (\epsilon, \epsilon)\}$;
3  let $S$ be a stack with initial element $q_C^0$;
4  while $S$ is nonempty
5    pop $p$ from $S$;
6    let $(\alpha, \beta) = Q(p)$;
7    foreach $p \xrightarrow{\varphi/(\mathbf{u},\mathbf{v})} q$ in $\delta_C(p)$
8      let $\mathbf{x} = \alpha \cdot \mathbf{u}$; $\mathbf{y} = \beta \cdot \mathbf{v}$;
9      if $q \in F_C \wedge |\mathbf{x}| \neq |\mathbf{y}|$ FAIL;
10     let $m = min(|\mathbf{x}|, |\mathbf{y}|)$;
11     let $\psi = \varphi \wedge (\bigvee_{i<m} \mathbf{x}(i) \neq \mathbf{y}(i))$;
12     if $IsSat(\psi)$ FAIL;
13     let $\mathbf{x}' =$ if $|\mathbf{x}| > m$ then $[\mathbf{x}(m), \ldots]$ else $\epsilon$;
14     let $\mathbf{y}' =$ if $|\mathbf{y}| > m$ then $[\mathbf{y}(m), \ldots]$ else $\epsilon$;
15     if $\mathbf{x}' = \epsilon \wedge \mathbf{y}' = \epsilon$
16       if $q \notin Dom(Q)$ push $q$ to $S$ and set $Q(q) = (\epsilon, \epsilon)$;
17       else if $Q(q) \neq (\epsilon, \epsilon)$ FAIL;
18     else if $\mathbf{y}' = \epsilon$
19       let $\ell'$ be a fresh variable of sort $\sigma$;
20       let $\varphi_1 = (\bigvee_{i<|\mathbf{x}'|} \mathbf{x}'(i) \neq Subst(\mathbf{x}'(i), \{\ell \mapsto \ell'\}))$;
21       let $\varphi_2 = \varphi \wedge Subst(\varphi, \{\ell \mapsto \ell'\}) \wedge \varphi_1$;
22       if $IsSat(\varphi_2)$ FAIL;
23       let $\mathbf{a}' = (\mathbf{x}')^M$ where $M \models \varphi$;
24       if $q \notin Dom(Q)$ push $q$ to $S$ and set $Q(q) = (\mathbf{a}', \epsilon)$;
25       else if $Q(q) \neq (\mathbf{a}', \epsilon)$ FAIL;
26     else ... (symmetrical case for $\mathbf{x}' = \epsilon$)
27 end of while;
28 SUCCEED;
```

Figure 3: Compatibility algorithm for SFTs.

1. There exist outputs of different length when q is final.
2. Some prefix of outputs differ.
3. Promises differ for q, use Lemma 1.
4. If φ_2 is satisfiable there exist two different values for the (symbolic) pending output x' from A, use Lemma 1.
5. When φ_2 is not satisfiable, any model $M \models \varphi$ gives the same interpretation α' for the (symbolic) pending output $\mathbf{x}'$. If there is already a pending output for q that differs from (α', ϵ), use Lemma 1.

If all local verifications hold, then compatibility follows, since the scope of the label variable is a single symbolic transition. Termination of the algorithm follows from termination of product, termination of satisfiability checks, finiteness of $Q_A \times Q_B$, and that each member of $Q_A \times Q_B$ is explored at most once. □

Note that all satisfiability checks use at most two free variables. (Two variables are needed in line 20 in Figure 3. All other satisfiability checks, including the ones performed in the construction of $A \times B$ need only a single free variable.) What is surprising is that the algorithm is

effectively uniform in the theory of labels.

No assumptions are needed about the theory of labels besides the standard assumptions of being closed under Boolean operations, substitutions, and equality. In contrast, the recent extension [1] of finite state transducers restricts the theories to be total orders with single order predicate and no other symbols to avoid undecidability. The other main advantage of SFTs over finite state transducers is

succinctness.

The expansion of A to $[\![A]\!]$ is either infinite or may increase the size exponentially. Thus, while partial-equivalence of single-valued finite state transducers is solvable $O(n^2)$ [10] steps, an expansion from an SFT over a large alphabet to a finite state transducer leads to $O(2^n)$ complexity (recall Example 3). However, the partial-equivalence algorithm for single-valued SFTs is $O(n^2)$ provided that satisfiability of the label theory is at most $O(n^2)$. A particular label theory whose satisfiability has the given complexity is linear arithmetic without addition and with at most two free variables. In general, assuming that that sizes of the individual label formulas are small relative to the total sizes of the SFTs, the local satisfiability checks may be considered to be $O(1)$ operations. For a precise complexity analysis one would first need to fix the label theory as well as the representation of the formulas.

Example 8. We illustrate the partial-equivalence checking algorithm on the SFTs A and B in Example 7. The search starts from (p_0, q_0) with $Q = \{(p_0, q_0) \mapsto (\epsilon, \epsilon)\}$ and proceeds as follows from line 7. Consider the transition $(p_0, q_0) \to (p_1, q_1)$. We have $\mathbf{x} = [\ell, \ell, \ell]$ and $\mathbf{y} = [\ell]$ and (p_1, q_1) is not final. So $m = 1$ and $\psi = (\ell = 1 \wedge \ell \neq \ell)$. Since ψ is unsatisfiable let $\mathbf{x}' = [\ell, \ell]$ and $\mathbf{y}' = \epsilon$ and continue from line 19. The formula φ_2 is $\ell = 1 \wedge \ell' = 1 \wedge (\ell \neq \ell' \vee \ell \neq \ell')$ and is thus unsatisfiable. The promise calculated for (p_1, q_1) is $([1, 1], \epsilon)$, thus

$$Q = \{(p_0, q_0) \mapsto (\epsilon, \epsilon), \quad (p_1, q_1) \mapsto ([1, 1], \epsilon)\}$$

and the algorithm continues from (p_1, q_1) at line 7. Consider the transition $(p_1, q_1) \to (p_1, q_1)$ first. We have $\mathbf{x} = [1, 1, \ell]$, $\mathbf{y} = [\ell]$, $m = 1$ and $\psi = (\ell = 1 \wedge 1 \neq \ell)$, so $\mathbf{x}' = [1, \ell]$ and $\mathbf{y}' = \epsilon$. The formula φ_2 is $\ell = 1 \wedge \ell' = 1 \wedge (1 \neq 1 \vee \ell \neq \ell')$ and is thus unsatisfiable. The promise calculated for (p_1, q_1) is again $([1, 1], \epsilon)$, that is identical with the existing promise. Analysis of the other transition $(p_1, q_1) \to (p_2, q_2)$ is similar and leads to line 16, so

$$Q = \{\ldots, (p_2, q_2) \mapsto (\epsilon, \epsilon)\}$$

Finally, there are no transitions from (p_2, q_2), so the algorithm terminates with the verdict $A \overset{1}{=} B$. ⊠

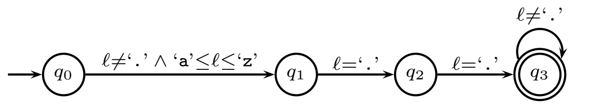
5.3 Algebra of SFTs

We consider an algebra of SFTs, in Figure 4, that allows us to express several useful decision problems involving SFTs and SFAs. Note that the join composition $T \circ A$ of an SFT T with an SFA A is again an SFA because all the outputs of $T \circ A$ are empty, $T \circ A$ is the *inverse image of T under A*.

$$\begin{array}{lcl} A & ::= & \mathit{sfa} \mid A \setminus A \mid A \cap A \mid T \circ A \\ T & ::= & \mathit{sft} \mid T \circ T \mid T \upharpoonright A \\ F & ::= & A \subseteq A \mid T \overset{1}{=} T \mid F \wedge F \mid \neg F \end{array}$$

Figure 4: Algebra of SFTs; A is a valid SFA expression; T is a valid SFT expression; F is a valid formula; *sfa* stands for an explicit definition of an SFA; *sft* stands for an explicit definition of an SFT.

Example 9. The following SFA is the inverse image of *UptoLastDot* (from Example 4) under the SFA(`"^[a-z]\.$"`):



For SFAs the empty output is typically omitted. ⊠

The formulas in Figure 4 are assumed to be well-sorted with respect to the input sorts and the output sorts of the SFAs and SFTs. The following theorem is the main decidability result of this paper.

Theorem 3. *The algebra of SFTs modulo a decidable label theory is decidable.*

Proof. The decidability of the SFA operations follows from the SFA properties studied in [30, 41]. The decidability of the SFT operations follows from Theorem 1, Theorem 2, (3), and Boolean satisfiability. □

The following corollary identifies a collection of practically relevant decision problems that follow from Theorem 3. *Subsumption* of SFTs, $A \sqsubseteq B$, is the problem of deciding if $\mathbf{T}_A(v) \subseteq \mathbf{T}_B(v)$ for all v. *Reachability* is the problem of existence of an input that is transformed to an output in a given set of outputs accepted by an SFA.

Corollary 1. *The following decision problems over single-valued SFTs modulo a decidable label theory are decidable.*

- *Subsumption.*
- *Equivalence.*
- *Idempotence.*
- *Commutativity.*
- *Reachability.*

Proof. Assume A and B are single-valued and recall Proposition 1. Subsumption, $A \sqsubseteq B$, is $\mathbf{d}(A) \subseteq \mathbf{d}(B) \wedge A \overset{1}{=} B$. Equivalence, $A \equiv B$, is $A \sqsubseteq B \wedge B \sqsubseteq A$. Idempotence is $A \equiv A \circ A$. Commutativity is $A \circ B \equiv B \circ A$. Reachability of a given output SFA D is $A \circ D \neq \emptyset$. □

It also follows, from Proposition 1 and Theorem 2, that we can decide single-valuedness of SFTs. This is a practically useful property, because it helps to verify that an SFT is well-defined, when it is supposed to be single-valued, e.g., as a result of a translation from a concrete program.

Corollary 2. *Single-valuedness of SFTs modulo a decidable label theory is decidable.*

The following example illustrates a class of common list transformations that involves a somewhat simplified case of the *regular lookahead* problem that occurs in Example 3 above. The point is to illustrate a simple scenario involving the use of the algorithms. Such SFTs are in general not intended to be handwritten but provide an intermediate representation for analysis, in particular, the compilation from Bek programs to SFTs is explained in [18]. We write $A(v) = w$ for $\mathbf{T}_A(v) = \{w\}$ for $v \in \mathbf{D}(A)$.

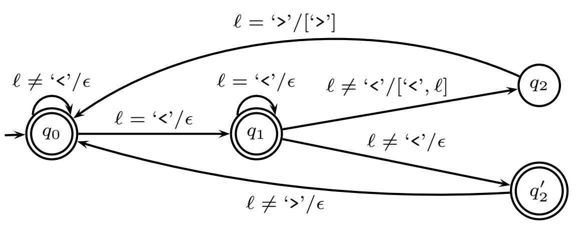
Example 10. The example illustrates an SFT *GetTags*. The intension is to extract from a given input string all substrings of the form $[\text{‘<’}, \ell, \text{‘>’}]$, where $\ell \neq \text{‘<’}$. For example

$$\mathit{GetTags}(\texttt{"<<s><<>><f><t"}) = \texttt{"<s><>><f>"}$$

Suppose that the following C# code is intended to implement *GetTags* where q is a variable that is used to keep track of the current position in the pattern $[\text{‘<’}, \ell, \text{‘>’}]$:

```
1:  string output = ""; int q = 0; char d = '\0';
2:  foreach (char c in input)
3:  {
4:    if (q == 0) q = (c == '<' ? 1 : 0);
5:    else if (q == 1) q = (c == '<' ? 1 : 2);
6:    else
7:    {
8:      if (c == '>') output = output + "<" + d + ">";
9:      q = 0;
10:   }
11:   d = c;
12: }
13: return output;
```

Then *GetTags* is the following SFT:[1]

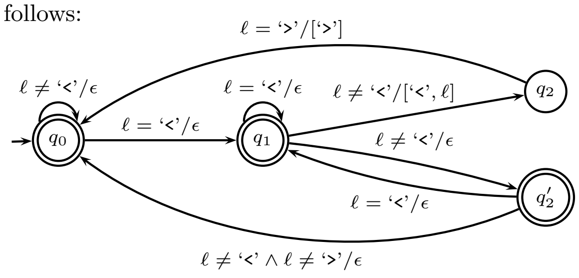


$\ell = \text{‘>’}/[\text{‘>’}]$
$\ell \neq \text{‘<’}/\epsilon$
$\ell = \text{‘<’}/\epsilon$
$\ell \neq \text{‘<’}/[\text{‘<’}, \ell]$
q_2
q_0
$\ell = \text{‘<’}/\epsilon$
q_1
$\ell \neq \text{‘<’}/\epsilon$
q_2'
$\ell \neq \text{‘>’}/\epsilon$

First, *GetTags* is expected to be idempotent. This is confirmed by checking $\mathit{GetTags} \circ \mathit{GetTags} \equiv \mathit{GetTags}$. Second, does *GetTags* detect all tags? In other words, does there exist an input v that contains the pattern $P = \texttt{"<[\^{}<]>"}$ but $\mathit{GetTags}(v) = \epsilon$?[2] The problem is to decide the reachability problem (4)

$$\underbrace{(\mathit{GetTags} \restriction \mathit{SFA}(P)) \circ E}_{D} \neq \emptyset \tag{4}$$

where $Q_E = F_E = \{q_E^0\}$ and $\delta_E = \emptyset$, i.e., $L(E) = \{\epsilon\}$. It turns out that D (when minimized) is an SFA with 8 states, and, e.g., $\texttt{"<a<a>"} \in L(D)$, i.e., $\mathit{GetTags}(\texttt{"<a<a>"}) = \epsilon$. The problem is that there is a missing case in *GetTags*: line 9 of the C# code should be `q = (c == '<' ? 1 : 0);`. The corresponding SFT *GetTags2* that handles the missing case is as follows:



$\ell = \text{‘>’}/[\text{‘>’}]$
$\ell \neq \text{‘<’}/\epsilon$
$\ell = \text{‘<’}/\epsilon$
$\ell \neq \text{‘<’}/[\text{‘<’}, \ell]$
q_2
q_0
$\ell = \text{‘<’}/\epsilon$
q_1
$\ell \neq \text{‘<’}/\epsilon$
q_2'
$\ell = \text{‘<’}/\epsilon$
$\ell \neq \text{‘<’} \wedge \ell \neq \text{‘>’}/\epsilon$

By repeating the checks, idempotence still holds and (4) holds for *GetTags2*. The checks assume that the SFTs are indeed single-valued. Suppose for example that the check $\ell \neq \text{‘>’}$ is missing in the condition of the transition from q_2' to q_0 in *GetTags2*, e.g., as a result of a translation error from the C# code, say the faulty SFT is *GetTags3*. Then $\mathit{GetTags3} \not\overset{1}{=} \mathit{GetTags3}$. ⊠

5.4 Model generation

The algorithm for compatibility $A \overset{1}{=} B$ plays a central role in most of the decision problems. However, it is not constructive, e.g., it does not provide a concrete witness v such that $\mathbf{T}_A(v) \neq \mathbf{T}_B(v)$ when A and B are single-valued and $A \not\overset{1}{=} B$. Here we briefly discuss a solution to this problem, called *model generation*. The technique is a generalization of the symbolic language acceptors for SFAs [41], the generalization to SFTs is explained in more detail in [5].

State-of-the-art SMT solvers such as Z3 support an integrated combination of decision procedures for the theory of uninterpreted functions symbols, algebraic data types, linear arithmetic, bitvector arithmetic, and the theory of arrays. An SFT $A^{\iota/o}$ is associated with a collection of uninterpreted relation symbols

$$\mathit{Acc}_q : \textsc{list}\langle\iota\rangle \times \textsc{list}\langle o\rangle \to \textsc{bool} \quad (q \in Q_A)$$

For each state $p \in Q_A$ the following auxiliary (mutually recursive) axioms are asserted to the solver. For example, suppose $p \xrightarrow{\varphi_i[\ell]/[t_i[\ell]]}_A p_i$ for $i \in \{1, 2\}$, then

$$\begin{array}{l}(\forall\, y : \textsc{list}\langle\iota\rangle, z : \textsc{list}\langle o\rangle) \\ \quad \mathit{Acc}_q(y, z) \quad \Leftrightarrow \quad \bigvee_{i\in\{1,2\}}(y \neq \epsilon \wedge \varphi_i[\mathit{hd}(y)] \wedge \\ \qquad\qquad\qquad\qquad\qquad z \neq \epsilon \wedge \mathit{hd}(z) = t_i[\mathit{hd}(y)] \wedge \\ \qquad\qquad\qquad\qquad\qquad \mathit{Acc}_{p_i}(\mathit{tl}(y), \mathit{tl}(z)))\end{array}$$

The auxiliary axioms can be guaranteed to be *well-founded* by first eliminating epsilon moves. Provided that $A \not\overset{1}{=} B$ has been established, satisfiability of the following assertion will generate a witness for some $y = [\ell_1, \ldots, \ell_n]$, $n \geq 0$, where $\ell_i : \iota$, for $i \leq n$, are uninterpreted constants

$$\mathit{Acc}_{q_A^0}(y, z) \wedge \mathit{Acc}_{q_B^0}(y, w) \wedge z \neq w \tag{5}$$

Essentially, the axioms are used as a semidecision procedure to search for a witness. We have implemented this technique to support witness generation in combination with the implementation of the algebra of SFTs in the context of the Bek project [3].

Example 11. Consider the SFTs *GetTags* and *GetTags2* in Example 10. We know that $\mathit{GetTags} \not\overset{1}{=} \mathit{GetTags2}$. Model generation for (5) generates a witness $y = [\text{‘<’}, 0, \text{‘<’}, 0, \text{‘>’}]$, that is also a shortest witness of $\mathit{GetTags} \not\overset{1}{=} \mathit{GetTags2}$. Similarly, model generation for $\mathit{GetTags3} \not\overset{1}{=} \mathit{GetTags3}$ generates a witness $[\text{‘<’}, 0, \text{‘>’}]$ that is also a shortest witness. ⊠

[1]Note that if the characters are represented as integers, then a deterministic version of *GetTags* does not exists, and if the characters are represented as 16-bit bitvectors then it blows up by a factor of 2^{16}.

[2]Note that the regex P implicitly allows an arbitrary prefix and suffix of any characters.

Our implementation contains roughly $5,000$ lines of C# code implementing the basic transducer algorithms and Z3 [9] integration. We build on an existing symbolic finite automata analysis library; if this is also included then the total size is roughly $10,000$ lines of C# code. Our technical report discusses performance in the context of the Bek project [18].

6. RELATED WORK

General equivalence of finite state transducers is undecidable [16], and already so for very restricted fragments [22]. Equivalence of decidability of single-valued GSMs was shown in [36], and extended to the finite-valued case (there exists k such that, for all v, $|\mathbf{T}_A(v)| \leq k$) in [8, 45]. The decidability of equivalence of the finite-valued case does not follow from the single-valued case. Corresponding decidability result of equivalence of finite-valued SFTs is shown in [5]. Unlike for the single-valued case that has a practical algorithm (Figure 3), the finite-valued case is substantially harder, the compatibility algorithm does not generalize to this case because the satisfiability checks cannot be made locally: Lemma 1 does not show violation of partial-equivalence in the finite-valued case.

In recent years there has been considerable interest in automata over infinite languages [37], starting with the work on *finite memory automata* [23], also called *register automata*. Finite words over an infinite alphabet are often called *data words* in the literature. Other automata models over data words are *pebble automata* [29] and *data automata* [6]. Several characterizations of logics with respect to different models of data word automata are studied in [4]. This line of work focuses on fundamental questions about definability, decidability, complexity, and expressiveness on classes of automata on one hand and fragments of logic on the other hand. A different line of work on automata with infinite alphabets introduces *lattice automata* [15] that are finite state automata whose transitions are labeled by elements of an atomic lattice with motivation coming from verification of symbolic communicating machines.

Finite state automata with arbitrary predicates over labels, called *predicate-augmented finite state recognizers*, or *symbolic finite automata* (SFAs) in the current paper, were first studied in the context of natural language processing [30]. While the work [30] views symbolic automata as a "fairly trivial" extension, the fundamental algorithmic questions are far from trivial. For example, it is shown in [19] that symbolic complementation by a combinatorial optimization problem called *minterm generation* leads to significant speedups compared to state-of-the-art automata algorithm implementations.

The work in [30] introduces a different symbolic extension to finite state transducers called *predicate-augmented finite state transducers*. This extension is not expressive enough for describing SFTs. Besides identities, it is not possible to establish functional dependencies from input to output that are needed for example to encode transformations such as *EncodeHtml*. *Streaming transducers* [1] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence, e.g., full linear arithmetic is not allowed.

SFTs are used in the Bek project [3, 18] (Bek is also available online as a web service[3]) and use the SMT solver Z3 [47, 9] for solving label constraints that arise during composition and equivalence checking algorithms, as well as for witness search by model generation using auxiliary SFT axioms. Finite state transducers have been used for dynamic and static analysis to validate sanitization functions in web applications in [2], by an over-approximation of the strings accepted by the sanitizer using static analysis of existing PHP code. Other security analysis of PHP code, e.g., SQL injection attacks, use string analyzers to obtain over-approximations (in form of context free grammars) of the HTML output by a server [28, 43, 44].

Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. HAMPI [24] and Kaluza [34] extend the STP solver to handle equations over strings and equations with multiple variables. The work in [20] shows how to solve subset constraints on regular languages. In contrast, we show how to combine any of these solvers with SFTs whose edges can take symbolic values in the theories understood by the solver.

Axiomatization of SFAs using a background of lists was initially introduced in [41] and is used to provide integrated support for *regex*-constraints in parameterized unit testing of .NET code [33, 39], and to provide analysis support for *like*-expressions in SQL query analysis [42]. Axiomatization of SFAs was extended to symbolic PDAs in [40].

Top-down tree transducers [14] provide another extension of finite state transducers: a finite state transducer is a top-down tree transducer over a *monadic* ranked alphabet. Similar to finite state transducers, decidability of equivalence of top-down tree transducers is known for the single-valued case [11, 13], including a specialized method for the *deterministic* case [7], and also for the finite-valued case [38]. Although several extensions of top-down tree transducers have been studied, e.g., [14, 27, 12, 32, 26, 25, 31], as far as we know, the label alphabet is always fixed and finite, none of the extensions have considered a symbolic representation of the transducers where the labels are factored out as a separate theory.

7. CONCLUSION

We introduced a symbolic extension of the theory of classical finite transducers, where transitions are represented by terms modulo a given background theory. It enables a new powerful algorithmic foundation to address many different software analysis problems in combination with state-of-the-art constraint solving techniques. The core algorithms we presented are composition and equivalence checking of single-valued symbolic finite transducers, and we showed how to decide whether arbitrary symbolic transducers have the single-valuedness property. These algorithms make it possible to work with symbolic transducers, just as traditionally done with finite state transducers, as first class citizens in designing new analyses and program transformation techniques by leveraging the continuous advances and improvements in constraint solvers and satisfiability modulo theories solvers. We demonstrated how our work directly applies to analysis of web string sanitizers and we expect more applications to follow.

[3]http://www.rise4fun.com/Bek

8. REFERENCES

[1] R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11*, pages 599–610. ACM, 2011.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Oakland Security and Privacy*, 2008.

[3] Bek. http://research.microsoft.com/bek.

[4] M. Benedikt, C. Ley, and G. Puppis. Automata vs. logics on data words. In *CSL*, volume 6247 of *LNCS*, pages 110–124. Springer, 2010.

[5] N. Bjørner and M. Veanes. Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research, January 2011.

[6] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE, 06.

[7] B. Courcelle and P. Franchi-Zannettacchi. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17:163–191, 1982.

[8] K. Culic and J. Karhumäki. The equivalence of finite-valued transducers (on HDTOL languages) is decidable. *Theoretical Computer Science*, 47:71–84, 1986.

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

[10] A. J. Demers, C. Keleman, and B. Reusch. On some decidable properties of finite state translations. *Acta Informatica*, 17:349–364, 1982.

[11] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R. V. Book, editor, *Formal Language Theory*, pages 241–286. Academic Press, New York, 1980.

[12] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39:2003, 2003.

[13] Z. Esik. Decidability results concerning tree transducers. *Acta Cybernetica*, 5:1–20, 1980.

[14] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers.* EATCS. Springer, 1998.

[15] T. L. Gall and B. Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *SAS 2007*, volume 4634 of *LNCS*, pages 52–68, 2007.

[16] T. Griffiths. The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. *J. ACM*, 15:409–413, 1968.

[17] W. Hodges. *Model theory.* Cambridge Univ. Press, 1995.

[18] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Bek: Modeling imperative string operations with symbolic transducers. Technical Report MSR-TR-2010-154, Microsoft Research, November 2010.

[19] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI'11*, LNCS. Springer, 2011.

[20] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, New York, NY, USA, 2009. ACM.

[21] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[22] O. Ibarra. The unsolvability of the equivalence problem for Efree NGSM's with unary input (output) alphabet and applications. *SIAM Journal on Computing*, 4:524–532, 1978.

[23] M. Kaminski and N. Francez. Finite-memory automata. In *31st Annual Symposium on Foundations of Computer Science (FOCS 1990)*, volume 2, pages 683–688. IEEE, 1990.

[24] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *ISSTA*, 2009.

[25] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'10, pages 495–508. ACM, 2010.

[26] A. Maletti, J. Graehl, M. Hopkins, and K. Knight. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39:410–430, June 2009.

[27] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proc. 19th ACM Symposium on Principles of Database Systems (PODS'2000)*, pages 11–22. ACM, 2000.

[28] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

[29] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. CL*, 5:403–435, 2004.

[30] G. V. Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:2001, 2001.

[31] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*, pages 587–598. ACM, 2011.

[32] T. Perst and H. Seidl. Macro forest transducers. *Information Processing Letters*, 89(3):141–149, 2004.

[33] Pex. http://research.microsoft.com/projects/pex.

[34] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for javascript. In *IEEE Security and Privacy*, 2010.

[35] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: Preventing script injection attacks in legacy web applications with automatic sanitization. Technical Report MSR-TR-2010-128, Microsoft Research, August 2010.

[36] M. P. Schützenberger. Sur les relations rationnelles. In *GI Conference on Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 209–213, 1975.

[37] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In Z. Ésik, editor, *CSL*, volume 4207 of *LNCS*, pages 41–57, 2006.

[38] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Math. Systems Theory*, 27:285–346, 1994.

[39] N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *TAP'08*, volume 4966 of *LNCS*, pages 134–153, Prato, Italy, April 2008. Springer.

[40] M. Veanes, N. Bjørner, and L. de Moura. Symbolic automata constraint solving. In C. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS*, pages 640–654. Springer, 2010.

[41] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*. IEEE, 2010.

[42] M. Veanes, N. Tillmann, and J. de Halleux. Qex: Symbolic SQL query explorer. In *LPAR-16*, LNAI. Springer, 2010.

[43] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[44] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, 2008.

[45] A. Weber. Decomposing finite-valued transducers and deciding their equivalence. *SIAM Journal on Computing*, 22(1):175–202, February 1993.

[46] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 41–110. Springer, 1997.

[47] Z3. http://research.microsoft.com/projects/z3.