Modeling Imperative String Operations with Transducers

Pieter Hooimeijer
University of Virginia
pieter@cs.virginia.edu

David Molnar
Microsoft Research
dmolnar@microsoft.com

Prateek Saxena
UC Berkeley
prateeks@cs.berkeley.edu

Margus Veanes
Microsoft Research
margus@microsoft.com

Abstract

We present a domain-specific imperative language, Bek, that directly models low-level string manipulation code featuring boolean state, search operations, and substring substitutions. We show constructively that Bek is reversible through a semantics-preserving translation to *symbolic* finite state transducers, a novel representation for transducers that annotates transitions with logical formulae. Symbolic finite state transducers give us a new way to marry the classic theory of finite state transducers with the recent progress in *satisfiability modulo theories* (SMT) solvers. We exhibit an efficient well-founded encoding from symbolic finite state transducers into the higher-order theory of algebraic datatypes. We evaluate the practical utility of Bek as a constraint language in the domain of web application sanitization code. We demonstrate that our approach can address real-world queries regarding, for example, the idempotence and relative strictness of popular sanitization functions.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Validation; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; F.3.1 [*Logics and Meanings*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Algorithms, Languages, Theory, Verification

1. Introduction

A large fraction of security vulnerabilities (including SQL injection, cross-site scripting, and buffer overflows) arise due to errors in string-manipulating code. Developers frequently use low-level string operations, like concatenation and substitution, to manipulate data that must follow a particular high-level structure, like HTML or SQL. This leads to problems if the code fails to adhere to that intended structure, causing the output to have unintended consequences.

The growing rate of security vulnerabilities, especially in web applications, has sparked interest in techniques for vulnerability discovery in existing applications. A number of approaches have been proposed to address the problems associated with string manipulating code. Several static analyses aim to address cross-site scripting or SQL injection by explicitly modeling sets of values that strings can take at runtime [5, 16, 22, 23]. These approaches use analysis-specific models of strings that are based on finite automata or context-free grammars. More recently, there has been significant interest in constraint solving tools that model strings [2, 10, 12, 14, 18, 20, 21]. String constraint solvers allow any client analysis to express constraints (e.g., path predicates) that include common string manipulation functions.

Neither existing static approaches nor string constraint solvers are currently well-equiped to model low-level imperative string manipulation code. In this paper, we aim to address this niche. Our approach is based on the insight that many security-critical string functions can be modeled precisely using finite state transducers over a symbolic alphabet. We present Bek, a domain-specific language for modeling string transformations. The language is designed to be (a) sufficiently expressive to model real-world code, and (b) sufficiently restricted to allow precise analysis using transducers. Bek can model real-world sanitization functions, such as those in the .NET System.Web library, without approximation. We provide a translation from Bek expressions to the theory of algebraic datatypes, allowing Bek expressions to be used directly when specifying constraints for an SMT solver, in combination with other theories.

Key to enabling the analysis of Bek expressions is a new theory of *symbolic finite state transducers*, an extension of standard form finite transducers that we introduce formally in this paper. We develop a theory of symbolic transducers, showing its integration with other theories in SMT solvers that support E-matching [6]. We describe a tractable encoding of symbolic finite transducers into the theory of algebraic data types and prove its well-foundedness (i.e., given sufficient resources, any given query yields a finite-length proof) using model-theoretic insights. We introduce the concept of *join composition*, which enables us to preserve the key property of *reversibility* (i.e, given an output, produce corresponding inputs) that is crucial for checking sanitizer

correctness. Finally, we present a translation of Bek expressions into symbolic finite transducers.

To evaluate our approach, we show that several popular sanitization procedures can be ported to Bek with little effort. Each such port matches the behavior of the original procedure without any need for conservative over-approximation. We use a prototype implementation of Bek that uses the Z3 SMT solver [24] as the back end interpreter. During our experiments, we were able to generate witnesses for a previously-known vulnerability. We demonstrate that the prototype can resolve queries that are of practical interest to both users and developers of sanitization routines, such as "do two sanitizers exhibit deviant behaviors on certain inputs," "do multiple applications of a sanitizer introduce errors," or "given an possibility of attack output, what is the maximal set of corresponding inputs that demonstrate the attack?"

The primary contributions of this paper are:

- We formally describe a domain-specific language, Bek, for string manipulation. We describe a syntax-driven translation from Bek expressions to symbolic finite state transducers.
- We formalize symbolic finite state transducers and their reduction to the theory of algebraic datatypes, including the intersection and composition operations.
- We show that Bek can encode real-world string manipulating code used to sanitize untrusted inputs in Web applications. Within this domain, we demonstrate several applications that are of direct practical interest.

The rest of this paper is structured as follows. We provide a motivating example in Section 2. Sections 3–5 describe our approach. Section 3 describes Bek in terms of its large step operational semantics. Section 4 provides language-theoretic definitions and presents the translation from Bek expressions to transducers. In Section 5, we formalize a theory of *symbolic* finite transducers based on the theory of algebraic datatypes. We present our case studies in Section 6. Finally, we discuss closely related work in Section 7 and conclude in Section 8.

2. Motivating Example

We put our work in context using a code fragment from version 2.6.0 of `wu-ftpd`, a file transfer server, written in C, that has a known format string vulnerability. Figure 1 shows a fragment of code for handling `SITE EXEC` commands from `wu-ftpd`. The `SITE EXEC` portion of the file transfer protocol allows remote users to execute certain commands on the local server. The `cmd` string holds untrusted data provided by such a remote user; an example benign value is `"/usr/bin/ls -l *.c"`. This code is an indicative example of string processing in the wild: it tries to accomplish several tasks at once, it relies on character-level imperative updates to manipulate its input, and control flow depends on string values.

The variable `PATH` points to a directory containing executable files that remote users are allowed to invoke (e.g., `"/home/ftp/bin"`). To prevent the remote user from invoking other executables via pathname trickery (e.g., `cmd == "../../../bin/dangerous"`), lines 5–15 sanitize the command string by skipping past all slash-delimited path elements. However, skipping past all slashes does not have the desired effect: `"/bin/echo '10/5=2'"` should become `"/echo '10/5=2'"` and not `"5=2'"`; slashes should only be

```
void site_exec(char *cmd)
{
  char buf[MAXPATHLEN], *slash, *t;
  /* sanitize the command string */
  char *sp = (char*) strchr(cmd, ' ');          5
  if (sp == 0) {
    while((slash = strchr(cmd, '/')) != 0)
      cmd = slash + 1;
  }
  else {                                        10
    while(sp &&
          (slash = (char*) strchr(cmd, '/')) &&
           (slash < sp))
      cmd = slash + 1;
  }                                             15
  for (t = cmd; *t && !isspace(*t); t++)
    if (isupper(*t)) *t = tolower(*t);
  /* build the command */
  int pathlen = strlen(PATH);
  int cmdlen  = strlen(cmd);                    20
  if (pathlen + cmdlen +2 > sizeof(buf))
    return;
  sprintf(buf, "%s/%s", PATH, cmd);
  /* ... execute buf, store results ... */
  fprintf(remote_socket, cmd);                  25
}
```

Figure 1. Source code using hand-written sanitization and checks to avoid a buffer overrun (successfully, line 21) and a format string vulnerability (unsuccessfully, line 25), and enforce path-related policies (successfully).

removed from the command, not from the arguments. The `strchr` invocation on line 5 is used to check if any spaces are present (line 6). If so, a more complicated version of the slash-skipping logic is used (lines 10–15) that only advances `cmd` past slashes before the first space. Lines 18–22 build the command that will be executed (e.g., completing the transformation from `"/usr/bin/ls -l *.c"` to `"/home/ftp/bin/ls -l *.c"`) by using `sprintf` to concatenate the trusted directory, a slash, and the suffix of the user command. The check on line 21 prevents a buffer overrun on the local stack-allocated variable `buf` by explicitly adding together the two string lengths, one byte for the slash, and one byte for C's null termination, and comparing the result against the size of `buf`.

More tellingly, while the code correctly avoids buffers overruns and implements its path-based security policy, it is vulnerable to a format string attack. Since the user's command is passed as the format string to `fprintf` (line 25), if it contains sequences such as `%d` or `%s` they will be interpreted by `printf`'s formatting logic. This typically results in random output, but careful use of the uncommon `%n` directive, which instructs `printf` to store the number of characters written so far through an integer pointer on the stack, can allow an adversary to take control of the system. An exploit for just such an attack against exactly this code was made publicly available [4].

3. Modeling Low-Level String Operations

In this section, we give a high-level description of a small imperative language, Bek, of low-level string operations. Our goal is two-fold. First, it should be possible to model Bek expressions in a way that allows for their analysis using existing constraint solvers. Second, we want Bek to be sufficiently expressive to closely model real-world code (such as the `wu-ftpd` example of Section 2). This section defines

Bool Constants	$B \in \{\mathbf{T}, \mathbf{F}\}$	Bool Variables	$b, \ldots$
Char Constants	$d \in \Sigma$	Char Variables	$c, \ldots$
Int Constants	$n \in \mathbb{N}$	String Variables	t
String Literals	$const \in \Sigma^*$		

$$
\begin{array}{lrcl}
\text{Expressions} & strexpr & ::= & \textbf{iter}[cseq\ \textbf{in}\ strexpr]\ (init)\ \{clist\} \\
& & \mid & (strexpr)\ \textbf{from}\ posexpr \\
& & \mid & (strexpr)\ \textbf{upto}\ posexpr \\
& & \mid & strexpr \cdot const \mid const \cdot strexpr \\
& & \mid & t \\
& cseq & ::= & c \mid c,\ cseq \\
& init & ::= & b\ :=\ B,\ init \mid \epsilon \\
& clist & ::= & case\ clist \mid case \\
& case & ::= & \textbf{case}(bexpr)\ \{cstmt\} \\
& cstmt & ::= & cstmt\ cstmt \mid \textbf{pass}; \\
& & \mid & b\ :=\ boolexpr; \\
& & \mid & \textbf{yield}(chexpr); \\
\text{Positions} & posexpr & ::= & pterm \\
& & \mid & (pterm) \diamond n \quad \diamond \in \{+, -\} \\
& pterm & ::= & \textbf{last}\ const \\
& & \mid & \textbf{first}\ const \\
\text{Booleans} & bexpr & ::= & bexpr \vee bexpr \mid bexpr \wedge bexpr \\
& & \mid & \neg(bexpr) \mid chexpr = chexpr \\
& & \mid & bexpr = bexpr \mid B \mid b \\
\text{Characters} & chexpr & ::= & c \mid d \mid \$
\end{array}
$$

Figure 2. Concrete syntax for Bek. Well-formed Bek expressions are functions of type `string -> string`; the language provides basic constructs to filter and transform the single input string t.

$$
\frac{\langle \varnothing, init\rangle \Downarrow E_B \qquad t \text{ is fresh} \qquad \langle \varnothing, se\rangle \Downarrow \langle E', r'\rangle \qquad E^{(2)} = E_B[t \mapsto r'] \qquad \langle E^{(2)}, \textbf{iter}[c_1, \ldots\ \textbf{in}\ t]\ ()\ \{clist\}\rangle \Downarrow \langle E^{(3)}, r\rangle}{\langle E, \textbf{iter}[c_1 \ldots\ \textbf{in}\ se]\ (init)\ \{clist\}\rangle \Downarrow \langle \varnothing, r\rangle}\ \textsc{Itr}
$$

$$
\frac{E(s) = []}{\langle E, \textbf{iter}[c_1 \ldots\ \textbf{in}\ s]\ ()\ \{clist\}\rangle \Downarrow \langle \varnothing, []\rangle}
$$

$$
\frac{E(s) = w_1 :: w' \qquad t \text{ is fresh} \qquad E_C = E[c_1 \mapsto E(s)(1)] \ldots \qquad \langle E_C, clist\rangle \Downarrow \langle E^{(2)}, r\rangle \qquad \langle E^{(2)}[t \mapsto w'], \textbf{iter}[c_1, \ldots\ \textbf{in}\ t]\ ()\ \{clist\}\rangle \Downarrow \langle E', r'\rangle}{\langle E, \textbf{iter}[c_1, \ldots\ \textbf{in}\ s]\ ()\ \{clist\}\rangle \Downarrow \langle E', r \cdot r'\rangle}
$$

$$
\frac{\langle E, case\rangle \Downarrow \langle E', r\rangle}{\langle E, case\ clist\rangle \Downarrow \langle E', r\rangle} \qquad \frac{\langle E, case\rangle \Downarrow \mathsf{Skip} \qquad \langle E, clist\rangle \Downarrow \langle E', r\rangle}{\langle E, case\ clist\rangle \Downarrow \langle E', r\rangle}\ \textsc{Cases}
$$

$$
\frac{\langle E, be\rangle \Downarrow \mathbf{T} \qquad \langle E, cst\rangle \Downarrow \langle E', r\rangle}{\langle E, \textbf{case}(be)\{cst\}\rangle \Downarrow \langle E', r\rangle} \qquad \frac{\langle E, be\rangle \Downarrow \mathbf{F}}{\langle E, \textbf{case}(be)\{cst\}\rangle \Downarrow \mathsf{Skip}}
$$

Figure 3. Selected operational semantics for the **iter** construct, which provides a sliding window over the value of a string expression. Boolean state (declared using *init* in Itr) is available across iterations, but local to the **iter** block for which it is declared. For each iteration, only the body of the topmost matching **case** is evaluated (Cases). Case statements may update the boolean state, and **yield** zero or more characters (not shown).

Bek, presents its forward operational semantics, and provides examples. In the sections that follow, we demonstrate that Bek can be integrated into existing constraint solvers.

Figure 3 describes the language syntax. We define a single string variable, t, to represent an input string, and a number of expressions that can take either t or another expression as their input. The **from** and **upto** constructs represent search operations that truncate their input starting at (or ending with) the occurrence of a constant search string. Without the integer argument, the results of both **from** and **upto** include the matched search constant.

Example 1. The following expression searches for the last occurrence of `foo` in its input, returning everything following the match (if any).

$$(t)\ \textbf{from}\ (\textbf{last}\ \texttt{foo}) - 1;$$

If applied to the string `foofoo`, the output would be `ofoo`. If we replaced **last** with **first**, the result would also be `ofoo`, since there is no earlier occurrence of `foo` that has one preceeding character in the string. ⊠

The **iter** construct is designed to model loops that traverse strings while making imperative updates. Given a string expression (*strexpr*), a sequence of character binders (*cseq*), and an optional initial boolean state (*init*), an **iter**-block provides a sliding window over its input. For the ith (0-based) iteration, the character binders $c_1, \ldots, c_n$ are bound to characters w_i through w_{i+n-1} in the input. If some w_j do not exist (i.e., we have reached the end of the input), then the corresponding character binder is assigned the special symbol \$. The case statements inside the block can **yield** zero or more characters, and update the boolean state (affecting future iterations).

Example 2. The following expression represents a basic sanitizer that escapes single and double quotes (but only if they are not escaped already). An iter block declares a single-character window (c_1) and a single boolean state variable b_1, which is initially false.

```
iter[c1 in t] (b1 = F) {
    case(¬(b1) ∧ (c1 = ' ∨ c1 = ")) {
        b1 := F; yield(\); yield(c1);
    } case(c1 = \) {
        b1 := ¬(b1); yield(c1);
    } case(T) {
        b1 := F; yield(c1);
    }
}
```

The boolean variable b_1 is used to track whether the previous character seen was an unescaped slash. For example, in the input `\\"` the double quote is not considered escaped, and the transformed output is `\\\"`. If we apply the expression to `\\\"` again, the output is the same. An interesting question is whether this holds for any output string. In other words, we may be interested in whether a given Bek expression is *idempotent*.

If implemented wrongly, double applications of such sanitization functions has resulted in duplicate escaping which have opened real systems to command injection of script-injection attacks in the past. Checking *idempotence* of certain functions is practically useful. The transducer translation presented in Section 4 can be used to prove such properties about Bek expressions (including idempotence). ⊠

Figure 3 presents the operational semantics for the **iter** construct. We define the evaluation relation:

$$\Downarrow \subseteq (\mathit{context} \times \mathit{strexpr}) \times (\mathit{context} \times \mathit{string})$$

where a *context* E maps variables to values. The **iter** judgments update the environment to carry boolean state across iterations and to update the character binders for each iteration. Each iteration consumes the first character w_1 of the current remaining string. The case block conditions are checked in sequence; the first case to match is executed. If none of the case conditions match, we assume an implicit case (not shown) that outputs the empty string and makes no change to the state. We write $E(s)(n)$ for the nth character in the value of string variable s. If $n \geq \mathsf{len}(E(s))$, then $E(s)(n) = \$$. We define \$ to be a special character symbol that is uncomparable to in-domain characters.

Any well-formed derivation under these inference rules starts with the base case: $\langle E, t\rangle \Downarrow \langle \varnothing, E(t)\rangle$, where E is assumed as the initial assignment to t. The out state is used only by the evaluation rules for **iter**. We elide the judgments for the search operations **from** and **upto** and the concatenation-with-a-constant operations; they are defined directly in terms of their input string, yielding only the corresponding output string. Note that, in Figure 3, we ignore any state E' produced by the evaluation of nested string expression *se* (ITR judgment), and we always emit the empty mapping. In other words, the execution of an **iter** block is free of external side-effects. It follows that all toplevel *strexpr* judgments are side-effect free.

4. Translation to Finite Transducers

We now turn to the translation of BEK expressions to finite state transducers. For a given BEK expression P, we write $M[\![P]\!]$ for the corresponding finite transducer. We use this construction to show that BEK programs are *reversible*: given a BEK expression P and an output string y, we can compute the maximal set $R = \{x \mid P(x) = y\}$, and R is regular for any such computation. The presentation is structured as follows. In Section 4.1, we provide transducer-related definitions. Section 4.2 exhibits the high-level translation from BEK to finite transducers. Finally, in Section 5, we extend the definitions of Section 4.1 to a formal encoding of *symbolic* finite transducers. This allows for an implementation that integrates BEK-program-induced constraints directly with other constraints.

4.1 Definitions

We work in the context of a fixed *multi-sorted* universe of values, where each sort σ is (corresponds to) a sub-universe. The basic sorts that we need are the Boolean sort BOOL, with the values **t** and **f**, and the sort BV^n of *n-bit-vectors*, for $n \geq 1$. We also need the sort $\text{TUPLE}\langle\sigma_0, \ldots, \sigma_{n-1}\rangle$, for $n \geq 1$, of *n-tuples* of elements of sorts σ_i for $i < n$. The 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 matches with the standard lexicographic order of 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 $\varepsilon : \text{LIST}\langle\sigma\rangle$ and for all $e : \sigma$ and $l : \text{LIST}\langle\sigma\rangle$, $[e \,|\, l] : \text{LIST}\langle\sigma\rangle$. The accessors are $hd : \text{LIST}\langle\sigma\rangle \to \sigma$ and $tl : \text{LIST}\langle\sigma\rangle \to \text{LIST}\langle\sigma\rangle$ with their usual meaning. We also adopt the convention that $[a, b, c]$ stands for the list $[a \,|\, [b \,|\, [c \,|\, \varepsilon]]]$ and we write $l_1 \cdot l_2$ for the concatenation of l_1 with l_2. When convenient, we will use length-bounded lists in the context of finite sets (such as the alphabet of an automaton).

Words are represented by lists. Typically, characters have sort BV^n for some fixed $n > 0$, e.g., if words represent strings of ASCII characters, in which case constant characters are written as `'a'` assuming for example ASCII encoding. In general however, characters may have compound sorts such as $\text{TUPLE}\langle\text{BV}^7, \text{BV}^7, \text{BOOL}\rangle$, although *finite*, e.g., unbounded lists will not be considered as characters.

We assume basic familiarity with classical automata theory [13]. In this paper we use 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 use the following formal definition of a finite transducer. The definition is known as the *standard form* of a finite transducer [17, p. 69].

Definition 1. A *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 $Q \times (\Sigma \cup \{\epsilon\})$ to $2^{Q \times (\Gamma \cup \{\epsilon\})}$.

We indicate a component of a finite transducer A by using A as a subscript. Instead of $(q, b) \in \delta_A(p, a)$ we often use the more intuitive notation $p \xrightarrow{a/b}_A q$, or $p \xrightarrow{a/b} q$ when A is clear from the context. Given words v and w we let $v \cdot w$ be the concatenated word. Note that $v \cdot \epsilon = \epsilon \cdot v = v$.

Given $q_i \xrightarrow{a_i/b_i}_A q_{i+1}$ for $i < n$ we write $q_0 \xrightarrow{v/w}_A q_n$ where $v = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1}$ and $w = b_0 \cdot b_1 \cdot \ldots \cdot b_{n-1}$. A induces the binary relation $[\![A]\!] \subseteq \Sigma_A^* \times \Gamma_A^*$ as follows for which we use infix notation

$$v[\![A]\!]w \stackrel{\text{def}}{=} \exists q \in F_A \, (q_A^0 \xrightarrow{v/w} q)$$

Given two binary relations R_1 and R_2, we write $R_1 \circ R_2$ for the binary relation $\{(x, y) \mid \exists z \, (R_1(x, z) \wedge R_2(z, y))\}$. A useful composition of finite transducers A and B is the *join composition of A and B*, that is a finite transducer $A \circ B$ such that $[\![A \circ B]\!] = [\![A]\!] \circ [\![B]\!]$.

Definition 2. Let A and B be finite transducers. 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 $\delta_{A \circ B}$ is defined as follows

$$(p, q) \xrightarrow{a/c}_{A \circ B} (p', q') \stackrel{\text{def}}{=} \begin{array}{l} \exists b \, (b \neq \epsilon \wedge p \xrightarrow{a/b}_A p' \wedge q \xrightarrow{b/c}_B q') \\ \vee (p \xrightarrow{a/\epsilon}_A p' \wedge c = \epsilon \wedge q = q') \\ \vee (q \xrightarrow{\epsilon/c}_B q' \wedge a = \epsilon \wedge p = p') \end{array}$$

The first case (disjunct) in the definition of $\delta_{A \circ B}$ means that some character b is output in state p of A while input in the state q of B, thus consuming b in the composed transition that inputs a and outputs c (note that a or c may be ϵ). The second case means that A outputs nothing while inputting a, thus B stays in the same state. The third case means that B inputs nothing while outputting c, thus A stays in the same state. The following property is well-known.

Proposition 1. *Let A and B be finite transducers. Then $[\![A \circ B]\!] = [\![A]\!] \circ [\![B]\!]$.*

Similarly to parallel composition of finite automata, the join composition of finite transducers can be done incrementally using depth first search, avoiding the introduction of states that cannot be reached from the initial state, called *unreachable states*. Moreover, all states in a finite transducer from which no final state can be reached, called *dead states*, can be elmininated through backwards reachability. Both optimizations may significantly decrease the size of the resulting composite transducer while preserving equivalence in terms of the denoted relation.

4.2 Translating Bek Expressions

The evaluation order for Bek programs is relatively straightforward; each string expression depends either on the input variable t or on another string expression. There are no side effects, with the exception of the boolean state available in the **iter** construct, and that that boolean state is limited in scope to the **iter** block in which it is defined. This informs our approach: we define the translation function $M[\![\cdot]\!]$ recursively, using the composition operator $\circ$ on transducers to model nested string expressions. This leads to a single $M[\![\cdot]\!]$ for each type of *strexpr*; we give the high-level definition in Figure 4.

The Slide function is central to the translations for the **first**, **upto**, and **iter** constructs. For a given finite sort σ, Slide_σ takes an integer parameter and produces a transducer:

$$(Q, q^0, F, \sigma, \text{TUPLE}\langle\underbrace{\sigma \cup \{\$\}, ..., \sigma \cup \{\$\}}_{n}\rangle, \delta)$$

so that any input of sort σ is split into partially overlapping n-tuples.

Example 3. We consider a toy example to illustrate how the Slide operation can be implemented using concrete transducers. Figure 5 shows the full transducer for $\mathsf{Slide}_{\{\mathsf{a},\mathsf{b}\}}(2)$ (where $\{\mathsf{a}, \mathsf{b}\}$ can be modeled using sort BV^1). Given an input sequence [abba], transducer output is

$$\begin{array}{l}[\langle \mathsf{a},\mathsf{b}\rangle \\ \quad \langle \mathsf{b},\ \mathsf{b}\rangle \\ \quad\quad \langle \mathsf{b},\mathsf{a}\rangle \\ \quad\quad\quad \langle \mathsf{a},\ \$\rangle]\end{array}$$

Given a search request (t) **from** (**first** b) -1 applied to this string, we can output the first a when we see the first pair $\langle \mathsf{a}, \mathsf{b}\rangle$. Searches that involve **last** are handled analogously, but there we rely on the nondeterminism of the transducer (i.e., once we see a match we must not see it again). ⊠

Intuitively, we use this conversion to provide look-ahead for the search operations **first** and **upto**, and to provide the sliding window for **iter** blocks. For the search operation translations (e.g., the definition of FF in Figure 4), we assume implicit special handling of the \$ symbol, so that that symbol never appears in the output of such an operation. Similarly, we ignore **yield** statements if the character value is \$. We omit a full algorithmic description of Slide. In practice, a concrete instantiation of Slide-transducers is infeasible because the state space grows exponentially with n; we discuss our symbolic representation in Section 5.

The Iter function converts **iter** blocks into a corresponding finite transducer. Figure 6 describes a collecting semantics that defines this transducer. We introduce a judgment of the form:

$$F, P \vdash expr : F', P'$$

which states that, given an initial transducer F and a possibly-open boolean term P, the given expression *expr* yields the updated transducer F' and new term P'. The Itr judgment relates the collecting semantics to the output of the function Iter. To construct the transducer, we proceed as follows. We start with an initial transducer that has one state for each possible boolean assignment in the Bek expression (e.g., 2^4 states if *init* declares 4 distinct variables). We assume a mapping from concrete boolean states b to transducer states q_b. The transducer's start state is the state q_b such that $b = \mathsf{Red}(init)$, where Red reduces boolean Bek expressions to possibly-open propositional terms. We compose this automaton on the left with a Slide transducer to produce a sliding window of the appropriate width.

We process the **case** blocks in syntactic order (Cases). Recall that the semantics for case blocks require that we execute the first matching case (exclusively). We write $F \cup G$ to denote the transducer F extended with the set of

$$M[\![t]\!] = \mathsf{Ident}$$

$$M[\![strexpr \cdot w]\!] = M[\![strexpr]\!] \circ (\mathsf{Ident} \cdot \mathsf{Const}(w))$$

$$M[\![w \cdot strexpr]\!] = M[\![strexpr]\!] \circ (\mathsf{Const}(w) \cdot \mathsf{Ident})$$

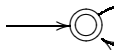
$$M[\![(se)\ \textbf{from}\ (\textbf{first}\ w) \diamond n]\!] = M[\![se]\!] \circ \mathsf{FF}(w, 0 \diamond n)$$

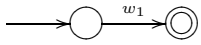
$$M[\![(se)\ \textbf{from}\ (\textbf{last}\ w) \diamond n]\!] = M[\![se]\!] \circ \mathsf{FL}(w, 0 \diamond n)$$

$$M[\![(se)\ \textbf{upto}\ (\textbf{first}\ w) \diamond n]\!] = M[\![se]\!] \circ \mathsf{UF}(w, 0 \diamond n)$$

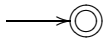
$$M[\![(se)\ \textbf{upto}\ (\textbf{last}\ w) \diamond n]\!] = M[\![se]\!] \circ \mathsf{UL}(w, 0 \diamond n)$$

$$M[\![\textbf{iter}[cs\ \textbf{in}\ se]\ (init)\ \{cl\}]\!] = M[\![se]\!] \circ \mathsf{Iter}(cs, init, cl)$$

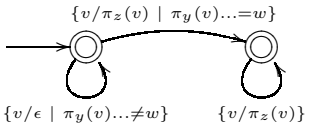
$\mathsf{Ident} =$ 

$\mathsf{Const}(w_1 :: w') =$ 

$\cdot\ \mathsf{Const}(w')$

$\mathsf{Const}([]) =$ 

$\mathsf{FF}(w, n) = \mathsf{Slide}_{\text{BV}^7}(x)$

$\circ$ 

$$\text{such that } x = \begin{cases} \mathsf{len}(w) - n & \text{if } n \leq 0 \\ \mathsf{max}(\mathsf{len}(w), n+1) & \text{if } n > 0 \end{cases}$$

$$y = \begin{cases} -n & \text{if } n \leq 0 \\ 0 & \text{if } n > 0 \end{cases} \quad z = \begin{cases} 0 & \text{if } n \leq 0 \\ n & \text{if } n > 0 \end{cases}$$

Figure 4. The definition of $M[\![\cdot]\!]$, the translation of Bek expressions to corresponding finite transducers. The functions FL, UF, UL are symmetric with FF. Slide, described in the text, returns a sliding window representation of its input to accomodate multi-character search and replacement. The integers x, y, and z represent the width of the window, the relative position of the "needle" in the window, and the relative positioning of the desired output, respectively.

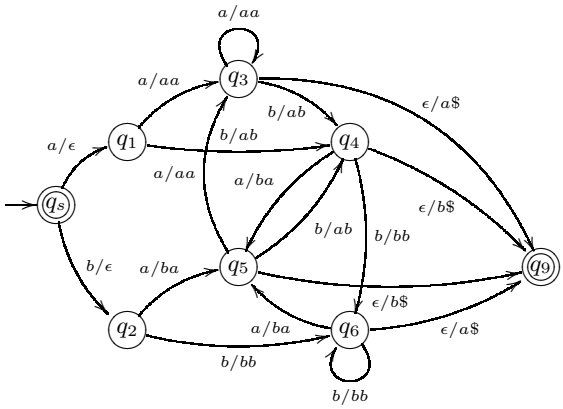


Figure 5. Finite transducer for $\mathsf{Slide}_{\{a,b\}}(2)$. For a length n and a finite sort σ, $\mathsf{Slide}_\sigma(n)$ transforms its input to an n-tuple output sort $\textsc{tuple}\langle\sigma \cup \{\$\}, \ldots, \sigma \cup \{\$\}\rangle$. The outputs represent a sliding window of the inputs. Note that these transducers grow very rapidly in the size of σ and n; we discuss how to avoid concrete instantiation in Section 5. For clarity, the diagram elides edges from q_1 and q_2 to q_9.

$$\frac{F, P \;\vdash\; case_1 \;:\; F^{(2)}, P^{(2)} \qquad F^{(2)}, P^{(2)} \;\vdash\; case_2 \;:\; F', P'}{F, P \;\vdash\; case_1\; case_2 \;:\; F', P'}\ \textsc{Cases}$$

$$\frac{\begin{array}{rl} F' & = F \cup \{q_{b_1} \xrightarrow{bc/cc} q_{b_2} \mid \quad bc = \mathsf{Red}(be \wedge \neg P)(b_1) \\ & \qquad\qquad\quad \wedge\ cc = \mathsf{Yields}(cst)(bc) \\ & \qquad\qquad\quad \wedge\ b_2 = \mathsf{Symex}(cst)(bc)\} \\ P' & = P \vee be \end{array}}{F, P \;\vdash\; \mathbf{case}(be)\{cst\} \;:\; F', P'}$$

$$\frac{\begin{array}{rl} F^{(2)} & = \mathsf{Slide}_{\mathrm{BV}^7}(\mathsf{len}(cs)) \\ & \quad \circ\ (Q, q_{\mathsf{Red}(init)}, Q, \mathrm{BV}^7, \mathrm{BV}^7, \delta) \\ P' & = false \\ F^{(2)}, P' & \vdash\ cases\ : F', P \\ F & = \mathsf{UnList}(F') \end{array}}{\vdash \mathsf{Iter}(cs, init, cases) = F}\ \textsc{Itr}$$

Figure 6. Collecting semantics for constructing transducers that model **iter** blocks. We represent the boolean states of the Bek expression using transducer states; we write q_b for the states in which boolean expression b is satisfiable. We write $\mathsf{Red}(b)(b')$ for the partial application of b as an open propositional term to b. Yields produces a list sort of character constraints. Symex processes case statements and converts them to an open propositional term.

transitions G. We use P to hold the disjunction of the case conditions we have already seen, and for each following case, we require that that disjunction be false.

We define the edges that must be added in terms of logical conditions. In particalar, for the current case block, we add edges $q_{b_1} \xrightarrow{bc/cc} q_{b_2}$ for each q_{b_1} given the following constraints:

1. bc defines a feasible character condition. In other words, there exists at least one character so that, starting at in boolean state b_1, the case condition be is true.
2. cc corresponds to the list of yields in the current case. Each c_i in the character binder is replaced with the appropriate projection $\pi_i(v)$, where v refers to the current input vector. We write Yield to indicate the extraction of list constraints from the case body.
3. b_2 is the result of executing the boolean assignments in the current case, given initial boolean state b_1 and the case condition be. We write Symex for the conversion of a sequence of boolean assignments to an open propositional term.

Finally, having added the appropriate edges for each case block, we must convert the output alphabet from Yield's list sort back to individual characters. Note that the maximum length of these lists is bounded by the maximum number of **yield** statements per case. The UnList operation is similar to Slide (e.g., Figure 5). As with Slide, we avoid instantiating the UnList transducers directly, instead relying on axiomatic definition in the theorem prover.

The presentation of the Iter function would be significantly more complex if we discussed it strictly in terms of concrete character transitions. In the following section, we formalize the notion of *symbolic* finite transducers. This concept yields several direct benefits. First, we can avoid instantiating prohibitively large transducers like those for Slide and UnList by using specialized axioms instead. Second, the symbolic encoding allows us to use the logical definition of Figure 6 directly without much further work.

5. Symbolic Finite Transducers

In this section we develop the theory of *symbolic* finite transducers. The theory lends itself to efficient symbolic analysis using state-of-the-art satisfiability modulo theories (SMT) solvers, and can be integrated through E-matching [6] with other theories supported by such solvers.

We first develop a mathematical theory of (symbolic) finite transducers and prove it to be well-defined for the class of *well-founded* finite transducers. The theory relies on the *combination* of the *theory of algebraic data types* [15], in particular lists, with the *theory of uninterpreted function symbols* that builds on the notion of *model expansion* from model theory [8].

We then discuss how algorithms can be built on top of the symbolic representation of finite transducers with a particular emphasis on *symbolic join composition* that is used in the translation of Bek to finite transducers, as discussed above.

Finally, we map that theory to a background theory of an SMT solver in terms of universally quantified transducer axioms. We discuss how such axioms work in general and how they are implemented using the SMT solver Z3, that is currently the only SMT solver that provides built-in support for the given combination of theories.

5.1 Symbolic finite transducer theory

In the following, let $A = (Q, q^0, F, \Sigma, \Gamma, \delta)$ be a fixed finite transducer. We assume that all input characters have the same sort $\textsc{sort}(\Sigma)$ and all output characters have the same

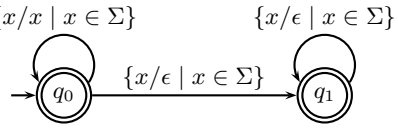


Figure 7. Finite transducer realizing the *prefix* operation of words in Σ^* where $\Sigma = \Gamma$.

sort $\text{SORT}(\Gamma)$. We use the following definitions to combine all possible input/output pairs of characters between any fixed pair (p, q) of states in Q. These definitions play an important role in a symbolic representation of transitions and in the defintion of the theory of A that is introduced below. Let $\delta^{(p,\_,\_,q)}(x,y)$, $\delta^{(p,\epsilon,\_,q)}(y)$, $\delta^{(p,\_,\epsilon,q)}(x)$, $\delta^{(p,\epsilon,\epsilon,q)}$ be predicates, where $x : \text{SORT}(\Sigma)$ and $y : \text{SORT}(\Gamma)$ are free variables, such that, where Σ and Γ are viewed as unary predicates:

$$\begin{aligned}
\delta^{(p,_,_,q)}(a,b) &\Leftrightarrow \Sigma(a) \wedge \Gamma(b) \wedge p \xrightarrow{a/b} q \\
\delta^{(p,_,\epsilon,q)}(a) &\Leftrightarrow \Sigma(a) \wedge p \xrightarrow{a/\epsilon} q \\
\delta^{(p,\epsilon,_,q)}(b) &\Leftrightarrow \Gamma(b) \wedge p \xrightarrow{\epsilon/b} q \\
\delta^{(p,\epsilon,\epsilon,q)} &\Leftrightarrow p \xrightarrow{\epsilon/\epsilon} q
\end{aligned}$$

Note that the predicates can always be represented as explicit disjunctions by combining individual characters, but this would often defeat the purpose of getting a more succinct and more efficient representation for analysis by using built-in functions and implicit symbolic representations.

Definition 3. We say that A is *symbolic* if δ is represented by predicates of the above form.

Example 4. Consider the finite transducer in Figure 7. The predicate $\delta^{(q_0,\_,\_,q_0)}(x,y)$ can be defined as $x = y$. Both $\delta^{(q_0,\_,\epsilon,q_1)}(x)$ and $\delta^{(q_1,\_,\epsilon,q_1)}(x)$ can be defined as $\mathbf{t}$. ⊠

We adapt the notion of IDs and step relations from [13] to finite transducers. An ID is an Instantaneous Description of a possible state of a finite transducer together with an input word and output word starting from that state. The formal definition is as follows.

Definition 4. An *ID* of A is a triple (v, q, w) where $v \in \Sigma^*$, $q \in Q$, and $w \in \Gamma^*$. The *step relation* of A is the binary relation $\vdash_A$ over IDs induced by δ.

$$\begin{aligned}
([a\,|\,v], p, [b\,|\,w]) \vdash_A (v, q, w) &\Leftrightarrow \delta^{(p,_,_,q)}(a,b) \\
([a\,|\,v], p, w) \vdash_A (v, q, w) &\Leftrightarrow \delta^{(p,_,\epsilon,q)}(a) \\
(v, p, [b\,|\,w]) \vdash_A (v, q, w) &\Leftrightarrow \delta^{(p,\epsilon,_,q)}(b) \\
(v, p, w) \vdash_A (v, q, w) &\Leftrightarrow \delta^{(p,\epsilon,\epsilon,q)}
\end{aligned}$$

The following proposition is an immediate consequence of the definitions.

Proposition 2. $v[A]w \Leftrightarrow \exists q \in F\,((v, q^0, w) \vdash_A^* (\epsilon, q, \epsilon))$.

The overall idea behind the theory $Th(A)$ introduced next is to precisely characterize $[\![A]\!]$. The definition is essentially an axiomatic formalization of $\vdash_A$.

Definition 5. Let A be as above. For each $p \in Q$, let

$$Acc_p : \text{LIST}\langle\text{SORT}(\Sigma)\rangle \times \text{LIST}\langle\text{SORT}(\Gamma)\rangle \to \text{BOOL}$$

be a predicate symbol of $Th(A)$ called the *acceptor for* p. $Th(A)$ contains the following axiom for each Acc_p:

$$\begin{aligned}
&Acc_p(v : \text{LIST}\langle\text{SORT}(\Sigma)\rangle, w : \text{LIST}\langle\text{SORT}(\Gamma)\rangle) \Leftrightarrow \\
&\quad (v = \varepsilon \wedge w = \varepsilon \wedge p \in F) \vee \\
&\quad \bigvee_{q \in Q}((v \neq \varepsilon \wedge w \neq \varepsilon \wedge \\
&\qquad \delta^{(p,_,_,q)}(hd(v), hd(w)) \wedge Acc_q(tl(v), tl(w))) \\
&\qquad \vee(v \neq \varepsilon \wedge \delta^{(p,_,\epsilon,q)}(hd(v)) \wedge Acc_q(tl(v), w)) \\
&\qquad \vee(w \neq \varepsilon \wedge \delta^{(p,\epsilon,_,q)}(hd(w)) \wedge Acc_q(v, tl(w))) \\
&\qquad \vee(\delta^{(p,\epsilon,\epsilon,q)} \wedge Acc_q(v, w)))
\end{aligned}$$

The *acceptor for* A, denoted by Acc_A, is the acceptor for q^0.

Note that the acceptor axioms above are written in a very general form and have not been simplifed. Obviously, false disjuncts can simply be eliminated, e.g., when $p \notin F$, or when there is no transition from p to q of a certain kind, as illustrated in the following example. The example also illustrates another simplification that can be used to eliminate some reqursive cases.

Example 5. Consider the transducer, say *Prefix*, in Figure 7. Assume that both the input and the output alphabets contain all characters of sort $\text{SORT}(\Sigma)$ (e.g. BV^7). Then $Th(\textit{Prefix})$ contains the following two axioms.

$$\begin{aligned}
Acc_{q_0}(v, w) &\Leftrightarrow (v = \varepsilon \wedge w = \varepsilon) \vee \\
&\qquad (v \neq \varepsilon \wedge w \neq \varepsilon \wedge hd(v) = hd(w) \wedge \\
&\qquad\quad Acc_{q_0}(tl(v), tl(w))) \vee \\
&\qquad (v \neq \varepsilon \wedge Acc_{q_1}(tl(v), w)) \\
Acc_{q_1}(v, w) &\Leftrightarrow (v = \varepsilon \wedge w = \varepsilon) \vee \\
&\qquad (v \neq \varepsilon \wedge Acc_{q_1}(tl(v), w))
\end{aligned}$$

The second axiom is equivalent to $Acc_{q_1}(v, w) \Leftrightarrow w = \varepsilon$. ⊠

The final simplification in Example 5, say *sink-simplification*, can always be applied to acceptor axioms for final states q when Σ contains *all* characters of $\text{SORT}(\Sigma)$, $\delta(q, x) = \{(q, \epsilon)\}$ for all $x \in \Sigma$ and $\delta(q, \epsilon) = \emptyset$, in which case

$$Acc_q(v, w) \Leftrightarrow w = \varepsilon$$

Thus, any input $v : \text{LIST}\langle\text{SORT}(\Sigma)\rangle$ is accepted, i.e., the input characters do not have to be individually restricted to Σ since this is imposed by the sort, while the output must be the empty word (list). Symmetrical simplification rule can be applied for output sink states.

We write $\mathbf{sat}(\varphi)$ for satisfiablity of formula φ (modulo the built-in theories), i.e., $\mathbf{sat}(\varphi)$ means that there exists a model M that provides an interpretation for all the uninterpreted function symbols in φ such that $M \models \varphi$. Note that the uninterpreted function symbols in $Th(A)$ are the acceptors. Also, given a theory T, we write $\bigwedge T$ for $\bigwedge_{\varphi \in T} \varphi$.

The *correctness* criterion that we need $Th(A)$ to fulfill is $\mathbf{sat}(\bigwedge Th(A) \wedge Acc_A(v, w))$ if an only if $v[\![A]\!]w$. To this end, we need to consider finite transducers whose step relation is well-founded.

Theorem 1. *If $\vdash_A$ is well-founded then $v[\![A]\!]w$ if and only if* $\mathbf{sat}(\bigwedge Th(A) \wedge Acc_A(v, w))$.

Proof. Assume $\vdash_A$ is well-founded. Thus, since Q is finite, there exists a well-ordering $\succ_Q$ over Q such that

$$p \succ_Q q \Rightarrow \neg((\epsilon, q, \epsilon) \vdash_A^+ (\epsilon, p, \epsilon)).$$

Define the lexicographic order $\succ$ over $\Sigma^* \times \Gamma^* \times Q$ as :

$$\begin{array}{rcl}(v,w,q) \succ (v',w',q') & \stackrel{\text{def}}{=} & |v| > |v'| \vee \\ & & (|v| = |v'| \wedge |w| > |w'|) \vee \\ & & (|v| = |v'| \wedge |w| = |w'| \wedge q \succ_Q q')\end{array}$$

The following statement follows by induction over $\succ$ using Definition 5. For all $p \in Q$, $v \in \Sigma^*$, and $w \in \Gamma^*$:

$$\exists q \in F\,((v,p,w) \vdash_A^* (\epsilon,q,\epsilon)) \Leftrightarrow \mathbf{sat}(\textstyle\bigwedge Th(A) \wedge Acc_p(v,w))$$

Finally, let $p = q^0$ and use Proposition 2. □

The following proposition provides a useful condition over the structure of A that is equivalent to $\vdash_A$ being well-founded; the proposition reflects the role of $\succ_Q$ in the proof of Theorem 1. An *ϵ-loop* is a nonemty path of *ϵ-moves* $p \xrightarrow{\epsilon/\epsilon} q$ that starts and ends in the same state.

Proposition 3. *$\vdash_A$ is well-founded $\Leftrightarrow$ A is ϵ-loop-free.*

The practical significance of the proposition is that there is an efficient algorithm that given A in *symbolic* form constructs an equivalent ϵ-loop-free finite transducer from A in *symolic* form (provided that *disjunction* over predicates is supported efficiently).

While full ϵ-move elimination may cause quadratic increase in the number of symbolic transitions (by eliminating *sharing*), ϵ-loop elimination does not increase the number of symbolic transitions. For symbolic analysis, full ϵ-move elimination may reduce the performance considerably, similar to the case of symbolic finite automata [21].

The following definition provides the key idea behind the ϵ-loop elimination algorithm. Recall the definition of *ϵ-closure*, denoted here by $\epsilon(q)$, as the closure of $\{q\}$, for $q \in Q$, by ϵ-moves [13] (where stated for finite automata, but is similar for finite transducers). Similarly, define $\ni(q)$ as the closure of $\{q\}$ by ϵ-moves in reverse. Let $\widetilde{q} \stackrel{\text{def}}{=} \epsilon(q) \cap \ni(q)$ (note that $\{q\} \subseteq \widetilde{q}$) and lift the notion to sets: $\widetilde{P} \stackrel{\text{def}}{=} \{\widetilde{p} \mid p \in P\}$.

Definition 6. Let $\widetilde{A} \stackrel{\text{def}}{=} (\widetilde{Q}, \widetilde{q^0}, \widetilde{F}, \Sigma, \Gamma, \widetilde{\delta})$ where

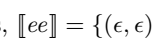
$$\begin{array}{rcl}\widetilde{\delta}^{(\widetilde{p},\_,\_,\widetilde{q})} & \stackrel{\text{def}}{=} & \bigvee_{p \in \widetilde{p}, q \in \widetilde{q}} \delta^{(p,\_,\_,q)} \\ \widetilde{\delta}^{(\widetilde{p},\_,\epsilon,\widetilde{q})} & \stackrel{\text{def}}{=} & \bigvee_{p \in \widetilde{p}, q \in \widetilde{q}} \delta^{(p,\_,\epsilon,q)} \\ \widetilde{\delta}^{(\widetilde{p},\epsilon,\_,\widetilde{q})} & \stackrel{\text{def}}{=} & \bigvee_{p \in \widetilde{p}, q \in \widetilde{q}} \delta^{(p,\epsilon,\_,q)} \\ \widetilde{\delta}^{(\widetilde{p},\epsilon,\epsilon,\widetilde{q})} & \stackrel{\text{def}}{=} & \bigvee_{p \in \widetilde{p}, q \in \widetilde{q}, \widetilde{p} \neq \widetilde{q}} \delta^{(p,\epsilon,\epsilon,q)}\end{array}$$

Note that if A is already ϵ-loop-free (such as the transducer in Figure 7) then A and $\widetilde{A}$ are isomorphic. The algorithm for constructing $\widetilde{A}$ can be implemented as a graph algorithm that collapses ϵ-loops into single nodes and joins the labels with disjunction. The algorithm is linear in the number of nodes plus edges (symbolic transitions), which may be an order of magnitude smaller than the number of concrete transitions. The actual complexity depends on the complexity of computing disjunctions (that is, without simplifications, an $O(1)$ operation in most SMT solvers).

The following theorem follows from Definition 6 and by using techniques similar to the proof of equivalence between nondeterministic finite automata and nondeterministic finite automata with epsilon-moves (see [13]).

Theorem 2. *$\widetilde{A}$ is ϵ-loop-free and $[\![A]\!] = [\![\widetilde{A}]\!]$.*

Theorem 1 fails if we omit the condition that $\vdash_A$ is well-founded as shown by the following example.

Example 6. Consider the transducer *ee*:  . Thus, $[\![ee]\!] = \{(\epsilon,\epsilon)\}$ and $Th(ee)$ is

$$\{Acc_{ee}(v,w) \Leftrightarrow (v = \epsilon \wedge w = \epsilon) \vee Acc_{ee}(v,w)\}.$$

For example let M be a model such that $M \models Acc_{ee}(v,w)$ for all v and w, then $M \models Th(ee)$, but $v[\![A]\!]w$ does not hold for all v and w. ⊠

The following theorem follows from Theorem 1, Proposition 3, and Theorem 2, and outlines the algorithm in a nut-shell for creating a soft-theory plug-in for A for an SMT solver, that in our case is Z3 [7].

Theorem 3. $v[\![A]\!]w \Leftrightarrow \mathbf{sat}(\bigwedge Th(\widetilde{A}) \wedge Acc_{\widetilde{A}}(v,w))$

When asserting $Th(\widetilde{A})$ as a soft theory to an SMT solver, the first assumption is that the solver actually supports lists as a built-in algebraic data type, which, unlike the acceptors, cannot be defined through uninterpreted functions, since *the theory of algebraic data types is not first-order definable* [15]. Note that the proof of Theorem 1 would fail without this assumption, where $\succ$ is defined in terms of *lengths* of words, which is well-defined since the notion of *counting* the elements of a list is well-defined.

5.2 Symbolic finite transducer algorithms

The built-in theory integration of state-of-the-art SMT solvers can be exploited, to some degree, for directly encoding finite transducer algorithms symbolically. One particular algorithm that we need is join composition of finite transducers. The following propostion shows a direct encoding of join composition.

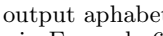
Proposition 4. *Assume* $\textsc{sort}(\Gamma_A) = \textsc{sort}(\Sigma_B)$. *Then* $\mathbf{sat}(\bigwedge Th(\widetilde{A}) \cup Th(\widetilde{B}) \wedge \exists z\,(Acc_{\widetilde{A}}(v,z) \wedge Acc_{\widetilde{B}}(z,w)))$ *if and only if* $v[\![A \circ B]\!]w$.

Proof. The folowing statements are equivalent:

1. $\mathbf{sat}(\bigwedge Th(\widetilde{A}) \cup Th(\widetilde{B}) \wedge \exists z\,(Acc_{\widetilde{A}}(v,z) \wedge Acc_{\widetilde{B}}(z,w)))$
2. $\exists z$ s.t. $\mathbf{sat}(\bigwedge Th(\widetilde{A}) \wedge Acc_{\widetilde{A}}(v,z))$ and $\mathbf{sat}(\bigwedge Th(\widetilde{B}) \wedge Acc_{\widetilde{B}}(z,w))$
3. $\exists z$ s.t. $v[\![A]\!]z$ and $z[\![B]\!]w$.

The equivalence between 1 and 2 holds by disjointness of the uninterpreted function symbols (acceptors) of the theories. The equivalenve between 2 and 3 follows from Theorem 3. Finally, use Proposition 1. □

While absence of ϵ-moves is preserved for example by *parallel composition* of finite automata, this is not the case for join composition of finite transducers.

Example 7. Consider the transducers $e\_$:  and $\_e$:  that have no ϵ-moves, and where the input and output aphabets are, say BOOL. Then $e\_ \circ \_e = ee$ with *ee* as in Example 6. It is therefore interesting to note that $Th(e\_) \cup Th(\_e)$ is well-defined by Proposition 4. Note that, with sink-simplification, as explained after Example 5, the axioms for $Th(e\_)$ and $Th(\_e)$ are $Acc_{e\_}(\_,w) \Leftrightarrow w = \varepsilon$ and $Acc_{\_e}(v,\_) \Leftrightarrow v = \varepsilon$, respectively. ⊠

In general, we can take acceptors for regular [21] and context free [20] languages an combine them with finite transducer acceptors and use SMT to solve them. For example,

suppose L is a regular language with a theory $Th(L)$ defining the acceptor Acc_L such that $Acc_L(v)$ iff $v \in L$, and A is a finite transducer then $\{w \mid \exists v\,(Acc_L(v) \wedge Acc_A(v,w))\}$ is the relational image of L under A.

While such direct encodings have certain advantages, such as generality, they cannot easily cope with unsatisfiable solutions when the acceptors are recursive and accept infinite languages. For example, a symbolic join composition algorithm that first constructs $A \circ B$ may discover that $A \circ B$ is empty, while the direct use of $Th(A) \cup Th(B)$ does not terminate. There are many nonobvious algorithmic tradeoffs that arise with the symbolic algorithms for finite transducers, similar to the case with finite automata [9], that are a subject of ongoing research.

5.3 Implementation with SMT solvers

The general idea behind the encoding of $Th(A)$ of a well-founded finite transucer A as a theory of an SMT solver, is similar to the encoding of *language acceptors* [20]. We use particular kinds of axioms, all of which are equations of the form

$$\forall \bar{x}(t_{\text{lhs}} \Leftrightarrow t_{\text{rhs}}) \tag{1}$$

where $FV(t_{\text{lhs}}) = \bar{x}$ and $FV(t_{\text{rhs}}) \subseteq \bar{x}$. The left-hand-side t_{lhs} of (1) is called the *head* of (1) and the right-hand-side t_{rhs} of (1) is called the *body* of (1) While SMT solvers support various kinds of patterns in general for *triggering* axioms, here we assume that the pattern of an equational axiom is always its head. The acceptor symbols are declared to the SMT solver as *uninterpreted Boolean function symbols* with the given sorts. In our encoding of $Th(A)$, each acceptor axiom in $Th(A)$ is represented by two axioms, one for the case $Acc_p(\varepsilon, w)$ and one for the case $Acc_p([x\,|\,v], w)$, motivated by our application domain where proofs are triggered by *input* words.

Such axioms are asserted as equations that are expanded during proof search. Expanding the formula up front is problematic since the equational axioms are in general mutually recursive and a naive a priori exhaustive expansion would in most cases not terminate, while straight-forward depth-bounded expansions are impractical as the size of the expansion is easily exponential in depth. Well-foundedness of A guarantees termination of the expansion process during proof search.

We use some features that are specific to Z3, including the integrated combination of decision procedures for algebraic data-types, integer linear arithmetic, bit-vectors and quantifier instantiation. We also make use of incremental features so that we can manipulate logical contexts while exploring different combinations of constraints. Working within a context enables *incremental* use of the solver. A context includes declarations for a set of symbols, assertions for a set of formulas, and the status of the last satisfiability check (if any). There is a *current context* and a backtrack stack of previous contexts. Contexts can be saved through *pushing* and restored through *popping*. This feature is used for implementing the satisfiability checks performed during symbolic join composition of finite transducers.

6. Implementation and Case Study

Our implementation contains roughly 5000 lines of C# code implementing the basic transducer algorithms and Z3 integration, and 1000 lines of F# code for translation from Bek. The work builds, in part, on the earlier Rex project where basic automata axioms were introduced [21]. Rex provides a lot of the boilerplate for the current project, in particular the language acceptors for regular expressions, that are built directly using the internal .NET parser for regexes.

6.1 Sliding Window Axioms

One of the algorithmic difficulties we encountered, when dealing with creating transducers from Bek, is related to maintaining a sliding window of characters (providing a look-ahead in the input string) or for outputting multiple characters in an iter block, which in some cases may lead to an explosion of the transducer state space for general purpose Bek programs. For example, assuming a look-ahead of 4 characters in the output string and a relatively small alphabet size of 20 characters already led to a transducer with over 200000 states.

A technique that enabled us to scale the approach on a collection of micro-benchmarks, was to use additional axioms and combine them with the acceptor axioms. In particular, when outputting a list of strings (that are represented as lists of bounded length) we use an axiom for *folding* such lists back to lists of singleton characters that are then fed to another transducer acceptor or an automaton acceptor. For example, for upper bound 3, the following axiom is used

$$\begin{aligned} &\mathit{fold}(x{:}\textsc{list}\langle\textsc{list}\langle\sigma\rangle\rangle, y{:}\textsc{list}\langle\sigma\rangle) \Leftrightarrow (x = \varepsilon \wedge y = \varepsilon) \vee \\ &\quad (x \neq \varepsilon \wedge \mathit{hd}(x) \neq \varepsilon \wedge \mathit{tl}(\mathit{hd}(x)) = \varepsilon \wedge \\ &\quad y \neq \varepsilon \wedge \mathit{hd}(\mathit{hd}(x)) = \mathit{hd}(y) \wedge \mathit{fold}(\mathit{tl}(x), \mathit{tl}(y))) \vee \\ &\quad \mathit{hd}(x)\ \textit{has exactly 2 characters case}\ \vee \\ &\quad \mathit{hd}(x)\ \textit{has exactly 3 characters case} \end{aligned}$$

E.g., $\mathit{fold}([[a,b],[c,d,e],[f]],[a,b,c,d,e,f])$. Using axioms of this kind, we connect several acceptors in a chain (avoiding the state space explosion), as in φ:

$$\exists x\, y\, z\,(Acc_A(x,y) \wedge \mathit{fold}(y,z) \wedge Acc_B(z)),$$

where A is a transducer generated from a sanitizer, and B is an acceptor for a regex pattern of disallowed output strings. Then φ is satisfiable iff the sanitizer has a bug, i.e., when there exists an input x that may produce an unwanted output. Moreover, the actual model generation with Z3 yields concrete witnesses for the existential variables and if no model is found then the sanitizer is correct with respect to B.

6.2 Macrobenchmarks

We then applied our framework to the analysis of code from real Web programs. A basic sanitization function in the Web context is "HTMLEncode," which takes a string and "escapes" characters such as angle brackets. This sanitization function has been re-implemented multiple times for different Web programs and libraries. Do all these implementations compute the same function? If not, is the set of characters escaped by one a superset of the characters escaped by another? These questions are important because failing to escape some characters can directly lead to a cross-site scripting attack by an adversary who can use the unescaped character to change a web browser's behavior.

To answer these questions, we translated five different implementations of the HTMLEncode function to the Bek language by hand. Our implementations were from the System.Web DLL version 2.0.0.0, an internal Microsoft product, and three versions of the AntiXSS library distributed by Microsoft. We used the free Reflector decompilation tool to extract C# code from the compiled binary and then translated the resulting code to Bek. The translation process took

rougly a day of work, the majority of which was spent understanding the original implementations.

The original sanitizer implementations themselves were 28, 63, 64, 111, and 147 lines of C# code. We discovered that all five implementations of HTMLEncode could be easily represented as a simple Bek iteration over single characters of the input string. Each iteration had 256 cases, one for each potential character value. We used metaprograms in Perl to output the C# constructor code needed by our implementation to create parse trees for Bek programs; these metaprograms ranged from 66 to 126 lines. Our results show that Bek is expressive enough to handle this crucial Web sanitizer and that the translation effort does not incur undue programmer time or overhead. We then focused on characters common in cross-site scripting attacks designed to foil sanitization. We can easily check whether such characters can be legal outputs of a sanitizer simply by transforming its Bek program to a symbolic finite state transducer, asserting that the output of the transducer is equal to the character in question, and then using our framework to solve for an input that yields the character.

Our framework discovered that the *single quote* character is a legal output of the System.Web HTMLEncode implementation. This is a security bug in the System.Web implementation, because the single quote character can be used in some HTML contexts to close string literals and open the way for a browser to treat subsequent strings as Javascript. The System.Web implementation, which also happens to be the most difficult to understand C# implementation of HTMLEncode, simply fails to transform single quotes under any circumstances. While this was difficult for us to determine by visual inspection of the C# code, our framework was able to solve for an example input exhibiting the problem in less than a second. We discovered that the problem was previously known independent of our work [3].

Our framework also showed that there are no strings of five characters or less that result in single quotes in a legal output from the other four sanitizer implementations. The "five characters or less" is a restriction we specified on the search by specifying a recursion depth of five for our fold axioms. We noticed that these four implementations of HTMLEncode have the property that they do not drop characters from the input on any path. Therefore, our framework's results are sufficient to show that no legal output of these sanitizers can contain single quotes.

We encountered performance problems, however, when scaling our framework to check for longer substrings in the output, such as strings found on the XSS Cheat Sheet web site. These longer substrings require specifying a deeper recursion depth for our fold axioms, because otherwise the search will not consider strings long enough to yield the desired substring. Unfortunately, this deeper recursion depth also yields a larger search space. We were not able to synthesize new inputs exhibiting potential cross-site scripting attacks for any of our Bek implementations, with a maximum of 5 minutes for each invocation of the underlying constraint solver.

In the case of the sanitizers we considered, it would be possible to directly encode all five implementations as regular transducers. This would allow bypassing the need for special Z3 axioms. Of course, some sanitizers, such as the `wu-ftpd` sanitizer we showed in Section 2, can not be captured this way. Exploring the performance and expressibility tradeoffs of a larger scale practical study is future work.

7. Related Work

Saner combines dynamic and static analysis to validate sanitization functions in web applications[1]. Saner creates finite state transducers for an over-approximation of the strings accepted by the sanitizer using static analysis of existing PHP code. In contrast, our work focuses on a simple language that is expressive enough to capture existing sanitizers or write new ones by hand, but then compile to symbolic finite state transducers that precisely capture the sanitization function. Saner also treats the issue of inputs that may be tainted by an adversary, which is not in scope for our work. Our work also focuses on efficient ways to compose sanitizers and combine the theory of finite state transducers with SMT solvers, which is not treated by Saner.

Minamide constructs a string analyzer for PHP code, then uses this string analyzer to obtain context free grammars that are over-approximations of the HTML output by a server[16]. He shows how these grammars can be used to find pages with invalid HTML. Our work treats issues of composition and state explosion for finite state transducers by leveraging recent progress in SMT solvers, which aids us in reasoning precisely about the transducers created by transformation of Bek programs.

Wasserman and Su also perform static analysis of PHP code to construct a grammar capturing an over-approximation of string values. Their application is to SQL injection attacks, while our framework allows us to ask questions about any sanitizer [22]. Follow-on work combines this work with dynamic test input generation to find attacks on full PHP web applications [23]. Our focus is specifically on sanitizers instead of on full applications; we emphasize precision of the analysis over scaling to large amounts of code.

Christensen et al.'s Java String Analyzer is a static analysis package for deriving finite automata that characterize an over-approximation of possible values for string variables in Java[5]. The focus of their work is on analyzing legacy Java code and on speed of analysis. In contrast, we focus on precision of the analysis and on constructing a specific language to capture sanitizers, as well as on the integration with SMT solvers.

Our work is complementary to previous efforts in extending SMT solvers to understand the theory of strings. HAMPI [14] and Kaluza [19] extend the STP solver to handle equations over strings and equations with multiple variables. Rex extends the Z3 solver to handle regular expression constraints [21], while Hooimeijer et al. show how to solve subset constraints on regular languages [11]. We in contrast show how to combine any of these solvers with finite automata whose edges can take symbolic values in the theories understood by the solver.

8. Conclusions

We have shown that combining SMT solvers with finite state transducers is a powerful way to reason precisely about the sanitization functions used for enforcing security guarantees. We introduced a special purpose language for capturing such sanitizers and showed that it is expressive enough to model real implementations of sanitizers. Future work points to a rich interplay between our style of symbolic finite transduce translation to logical formulas, improvement in the ability of SMT solvers to reason about string constraints, and the expressivity of language required to capture the special cases of transducers seen in practical sanitization examples.

References

[1] 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 *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[2] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS'09*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.

[3] N. Calinoiu. Httputility.htmlattributeencode security and rendering issues, 2005. Bug ID 103332, Microsoft Connect Visual Studio Feedback.

[4] CERT. CERT advisory CA-2001-33 multiple vulnerabilities in WU-FTPD, 2001.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, pages 1–18, 2003.

[6] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE-21: Proceedings of the 21st International Conference on Automated Deduction*, volume 4603 of *LNAI*, pages 183–198. Springer, 2007.

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

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

[9] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. Technical Report MSR-TR-2010-90, Microsoft Research, July 2010.

[10] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, pages 188–198, 2009.

[11] 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.

[12] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *ASE 2010: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010.

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

[14] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *ISSTA '09*, pages 105–116. ACM, 2009.

[15] A. Mal'cev. *The Metamathematics of Algebraic Systems*. North-Holland, 1971.

[16] 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.

[17] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1. Springer, 1997.

[18] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.

[19] 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.

[20] M. Veanes, N. Bjørner, and L. de Moura. Solving extended regular constraints symbolically. Technical Report MSR-TR-2009-177, Microsoft Research, 2009.

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

[22] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 32–41. ACM, 2007.

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

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

Appendix: Symbolic Join Composition

We start by providing some supporting definitions. We then describe the algorithm that, given two symbolic finite transducers A and B constructs a transducer AB such that $[\![AB]\!] = [\![A]\!] \circ [\![B]\!]$. Finally, we provide a correctness argument for the algorithm.

Definition 7. An *input/output condition with sort* σ/γ is a formula $IO(x{:}\sigma, y{:}\gamma)$.

Definition 8. Let $IO_1(x{:}\sigma, y{:}\rho)$ and $IO_2(y{:}\rho, z{:}\gamma)$ be input/output conditions. The *join of* IO_1 *with* IO_2, $IO_1 \circ IO_2$, is the input/output condition $\exists y\,(IO_1 \wedge IO_2)$ with sort σ/γ.

Definition 9. Let $C(x{:}\sigma)$ be a predicate and let $IO(x{:}\sigma, y{:}\gamma)$ be an input/output conditon. Then $C \circ IO$ is the predicate $\exists x\,(C(x) \wedge IO(x, y))$.

Definition 10. Let $C(y{:}\gamma)$ be a predicate and let $IO(x{:}\sigma, y{:}\gamma)$ be an input/output conditon. Then $IO \circ C$ is the predicate $\exists y\,(IO(x, y) \wedge C(y))$.

Definition 11. Given an input/output condition $IO(x, y)$ we let $[\![IO]\!]$ denote the binary relation $\{(x^M, y^M) \mid M \models IO\}$. Given a predicate C with a single free variable x we let $[\![C]\!]$ denote the set (unary relation) $\{x^M \mid M \models C\}$.

We assume that $\mathbf{sat}(\varphi)$ is a Boolean value that is $\mathbf{t}$ iff φ is satisfiable, meaning that there exists a model M that assigns values to all the uninterpreted function symbols in φ such that $M \models \varphi$, where each free variable of φ is treated as an uninterpreted constant symbol.

The intuition behind the following definition is that it symbolically represents δ_A for a given start state p and a given end state q, i.e., the transition label associated with a pair of states (p, q) is the four-tuple:

$$\langle \delta_A^{(p,\_,\_,q)}, \delta_A^{(p,\_,\epsilon,q)}, \delta_A^{(p,\epsilon,\_,q)}, \delta_A^{(p,\epsilon,\epsilon,q)} \rangle$$

Definition 12. A *transition label* ℓ *with sort* σ/γ, is a four-tuple $\langle IO(x{:}\sigma, y{:}\gamma), I(x{:}\sigma), O(y{:}\gamma), e \rangle$ of predicates,

- $IO(\ell) \stackrel{\text{def}}{=} IO$ is the *input/output condition of* ℓ;
- $I(\ell) \stackrel{\text{def}}{=} I$ is the *input condition of* ℓ;
- $O(\ell) \stackrel{\text{def}}{=} O$ is the *output condition of* ℓ;
- $\epsilon(\ell) \stackrel{\text{def}}{=} e$, $e \in \{\mathbf{t}, \mathbf{f}\}$, is the *$\epsilon$-condition of* ℓ;

ℓ is *feasible* if one of $\mathbf{sat}(IO)$, $\mathbf{sat}(I)$, $\mathbf{sat}(O)$, or e is $\mathbf{t}$.

The following operations are used in the join algorithm.

Definition 13. Let ℓ_1 and ℓ_2 be transition labels.

$$\begin{aligned} \ell_1 &= \langle IO_1(x, y), I_1(x), O_1(y), e_1 \rangle \\ \ell_2 &= \langle IO_2(y, z), I_2(y), O_2(z), e_2 \rangle \end{aligned}$$

Then $\ell = \ell_1 \circ \ell_2$, the *join of* ℓ_1 *with* ℓ_2, is the transition label

$$\langle IO(x, z), I(x), O(z), \mathbf{sat}(O_1 \wedge I_2) \rangle$$

where

$$IO = \begin{cases} \mathbf{f}, & \text{if } \mathbf{sat}(IO_1 \circ IO_2) = \mathbf{f}; \\ IO_1 \circ IO_2\ , & \text{otherwise.} \end{cases}$$

$$I = \begin{cases} \mathbf{f}, & \text{if } \mathbf{sat}(IO_1 \circ I_2) = \mathbf{f}; \\ IO_1 \circ I_2\ , & \text{otherwise.} \end{cases}$$

$$O = \begin{cases} \mathbf{f}, & \text{if } \mathbf{sat}(O_1 \circ IO_2) = \mathbf{f}; \\ O_1 \circ IO_2, & \text{otherwise.} \end{cases}$$

Note that the ϵ-conditions of ℓ_1 and ℓ_2 do not affect the definition of ℓ. However, the ϵ-condition of ℓ is determined by whether the output condition of ℓ_1 is consistent with the input condition of ℓ_2 or not.

Definition 14. Let ℓ_1 and ℓ_2 be transition labels

$$\begin{aligned} \ell_1 &= \langle IO_1(x,y), I_1(x), O_1(y), e_1 \rangle \\ \ell_2 &= \langle IO_2(x,y), I_2(x), O_2(y), e_2 \rangle \end{aligned}$$

Then $\ell = \ell_1 + \ell_2$, the *sum of* ℓ_1 *and* ℓ_2, is the transition label $\langle IO_1 \vee IO_2, I_1 \vee I_2, O_1 \vee O_2, \mathbf{sat}(e_1 \vee e_2) \rangle$.

Recall that a formula φ is *existential* if φ is equivalent to a formula $\exists \bar{x} \psi$ where ψ is quantifier free. We say that a transition label ℓ is *existential* if all conditions of ℓ are existential.

Proposition 5. *Sums and joins of labels are existential if the arguments are existential.*

Definition 15. We say that A is *clean* if all transition lables in A are feasible.

Proposition 5 is central for efficient implementation of $\mathbf{sat}(\varphi)$ during the construction of join of transition labels using an SMT solver, in order to maintain *cleanness* of the resulting finite transducer of the join algorithm.

Each symbolic transducer A is assumed to be represented so that $\Delta_A(p)$ yields the set of all pairs $\langle \delta_A^{(p,q)}, q \rangle$ from the state $p \in Q_A$, where

$$\delta_A^{(p,q)} = \langle \delta_A^{(p,\_,\_,q)}, \delta_A^{(p,\_,\epsilon,q)}, \delta_A^{(p,\epsilon,\_,q)}, \delta_A^{(p,\epsilon,\epsilon,q)} \rangle.$$

The join algorithm is described as a DFS algorithm. Symbolic transitions of AB are maintained using a dictionary Δ from $Q_{AB} \times Q_{AB}$ to transition labels. *Updating* Δ *with* $k \mapsto \ell$ stands for either adding the new entry $k \mapsto \ell$ to Δ, if k is not a key in Δ, or updating $\Delta(k)$ to $\Delta(k) + \ell$, otherwise.

Input: The input to the algorithm is a pair of clean finite transducers A and B such that $\textsc{sort}(\Gamma_A) = \textsc{sort}(\Sigma_B)$.

Output: The output of the algorithm is a clean finite transducer AB such that $[\![AB]\!] = [\![A]\!] \circ [\![B]\!]$.

Initialize: Let $S = (\langle q_A^0, q_B^0 \rangle)$ be a stack of AB states.

Let $Q = \{\langle q_A^0, q_B^0 \rangle\}$ be a set of AB states.

Let $\Delta = \emptyset$ be a dictionary from AB states to labels.

$\mathbf{s}(q) \stackrel{\text{def}}{=}$ If $q \notin Q$ then add q to Q and push q to S;

Explore: While S is nonempty:

Pick a state: Pop $p = \langle p_1, p_2 \rangle$ from S.

For each: $\langle \ell_1, q_1 \rangle \in \Delta_A(p_1)$ and $\langle \ell_2, q_2 \rangle \in \Delta_B(p_2)$:

If feasible$(\ell_1 \circ \ell_2)$**:** let $q = \langle q_1, q_2 \rangle$; $\mathbf{s}(q)$;

Update Δ with $(p,q) \mapsto \ell_1 \circ \ell_2$;

If $\epsilon(\ell_1) = \mathbf{t}$ **or** $\mathbf{sat}(I(\ell_1))$**:** let $q = \langle q_1, p_2 \rangle$; $\mathbf{s}(q)$;

Update Δ with $(p,q) \mapsto \langle \mathbf{f}, I(\ell_1), \mathbf{f}, \epsilon(\ell_1) \rangle$;

If $\epsilon(\ell_2) = \mathbf{t}$ **or** $\mathbf{sat}(O(\ell_2))$**:** let $q = \langle p_1, q_2 \rangle$; $\mathbf{s}(q)$;

Update Δ with $(p,q) \mapsto \langle \mathbf{f}, \mathbf{f}, O(\ell_2), \epsilon(\ell_2) \rangle$;

Finish: Let $AB = (Q, \langle q_A^0, q_B^0 \rangle, Q \cap F_A \times F_B, \Sigma_A, \Gamma_B, \Delta)$.

[Cleanup:] Remove dead states from AB. A *dead state* is a noninitial state from which no final state is reachable.

The final cleanup step is optional with respect to the intended semantics but may reduce the size of AB considerably. Note that the algorithm does not require A or B to be ϵ-loop-free. Also, $\widetilde{AB}$ construction is need for $Th(\widetilde{AB})$. The ϵ-loop elimination algorithm uses the sum operation over labels and does not require satisfiability checking of labels, since label-sums cannot produce infeasible transition labels from feasible ones.

Correctness argument Termination of the algoritm follows from the standard argument of DFS, i.e., in this particular case, an element is pushed to the stack S only if it is not in Q, i.e., has not already been in S, and the number of elements is bounded by the size of $Q_A \times Q_B$. Obviously, termination is contingent upon termination of satisfiability checking, that is being assumed. The partial correctness of the algorithm, AB is clean and $[\![AB]\!] = [\![A]\!] \circ [\![B]\!]$, follows from the following arguments. First, *cleanness* is implied by that in each update of Δ with $(p,q) \mapsto \ell$, ℓ is feasible and feasibility is preserved by sums. Second, S represents a frontier of reachable states, the algorithm computes the closure Q of all reachable states. The order of picking elements from S is irrelevant. Once a state p is chosen, all possible combinations of outgoing transitions from p are added. When a transition $(p,q) \mapsto \ell$ is added, it follows from Definition 2 that ℓ represents $\delta_{A \circ B}^{(p,q)}$, where the additional input-only and output-only moves correspond to the separate cases in Definition 2. It follows that $[\![AB]\!] = [\![A \circ B]\!]$ and thus $[\![AB]\!] = [\![A]\!] \circ [\![B]\!]$ by Proposition 1.