




Certified Symbolic Transducer with Applications in String Solving

Shuanglong Kan   

Barkhausen Institut, Germany

Abstract

Finite Automata (FAs) are fundamental components in the domains of programming languages. For instance, regular expressions, which are pivotal in languages such as JavaScript and Python, are frequently implemented using FAs. Finite Transducers (FTs) extend the capabilities of FAs by enabling the transformation of input strings into output strings, thereby providing a more expressive framework for operations that encompass both recognition and transformation. Despite the various formalizations of FAs in proof assistants such as Coq and Isabelle/HOL, these formalizations often fall short in terms of applicability to real-world scenarios. A significant limitation of classical FAs and FTs is that transition labels are typically defined as a single character from a finite alphabet. However, in practical applications, the alphabet of an FA or FT can be enormously large or even *infinite*. The classical approach to formalizing transitions can result in transition explosion, leading to critical performance bottlenecks of FA and FT operations.

A more pragmatic approach involves the formalization of symbolic FAs [12] and FTs [34], where transition labels are symbolic and potentially infinite. While the formalization of symbolic FAs has been explored in the work of CertiStr [17], the formalization of symbolic FTs in interactive proof assistants remains largely unexplored due to the increased complexity challenges. In this paper, we aim to formalize symbolic FTs within the Isabelle/HOL framework. This formalization is refinement-based and is designed to be extensible with various symbolic representations of transition labels. To assess its performance, we applied the formalized symbolic FTs to an SMT string solver for modeling replacement operations. The experimental results indicate that the formalized symbolic transducer can efficiently and effectively solve string constraints with replacement operations.

2012 ACM Subject Classification Replace `ccsdsc` macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.23

Funding *Shuanglong Kan*: (Optional) author-specific funding acknowledgements

Acknowledgements I want to thank ...

1 Introduction

Finite Automata (FA) and Finite Transducers (FT) are fundamental constructs in the theory of formal languages, with extensive applications in programming languages and software engineering. For example, recent advancements in string solvers, as demonstrated by [8], have illustrated the relationship between regular expressions in modern programming languages and various forms of FAs and FTs. Additionally, FAs and FTs find significant industrial applications, such as in the verification of AWS access control policies [3].

Even though there are various formalizations of FAs and FTs in interactive proof assistants such as Isabelle [2] and Coq [1], these are predominantly based on classical definitions. However, these traditional approaches present certain limitations when applied to practical scenarios. One significant drawback is that transition labels are typically non-symbolic and finite. A conventional transition is represented as $q \xrightarrow{a} q'$, where a is a character from a finite alphabet. This simplistic representation can lead to a phenomenon known as transition explosion. For example, if the alphabet Σ encompasses the entire Unicode range, which



© Jane Open Access and Joan R. Public;
licensed under Creative Commons License CC-BY 4.0

16th Conference on Interactive Theorem Proving (ITP 2025).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is common in modern programming languages, it consists of 0x110000 distinct characters. Defining a transition from state q to q' that accepts any character in Σ would require splitting into 0x110000 individual transitions, rendering the FA and FT operations highly inefficient. Furthermore, in practical applications, it is often necessary to consider infinite alphabets, such as the set of all integers.

Symbolic Finite Automata (SFA) and Symbolic Finite Transducers (SFT) [12, 34] represent advanced extensions of classic FAs and FTs, enhancing their applicability in practical scenarios. These symbolic models utilize transition labels defined by first-order predicates over boolean algebras, allowing for more expressive representations. For example, a transition label might be specified as an interval ('a' - 'z'), encompassing all characters from 'a' to 'z', or as an arithmetic condition ($x \bmod 2 = 0$), denoting all even numbers. This symbolic approach not only provides a more succinct representation but also supports infinite alphabets, thereby extending the expressive capabilities of FAs and FTs.

However, the formalization of transition labels in SFAs and SFTs presents significant challenges within interactive proof assistants. Two fundamental considerations arise: first, the representation of transition labels must be sufficiently expressive to accommodate diverse predicate representations of boolean algebras; second, the formalization framework must be designed with extensibility in mind to facilitate the incorporation of new boolean algebras while minimizing redundant proof efforts.

Prior work of CertiStr [17] has successfully formalized SFAs within Isabelle/HOL, demonstrating both the efficiency and effectiveness of SFAs in practice. However, the formalization of SFTs remains an open challenge, primarily due to their inherent complexity in two aspects: the formalization of transition labels and the specification of transition output functions. SFTs constitute a significantly more expressive and powerful theoretical framework compared to SFAs, as evidenced by their capability to model complex string transformations such as replacement operations. Furthermore, numerous studies have demonstrated the broad applicability of SFTs across diverse domains. The work of [34] showcases SFTs in security-critical applications, particularly for cross-site scripting (XSS) prevention. [15] extends this security focus by applying SFTs to web application sanitizer analysis. In system verification, [36] employs SFTs for runtime behavior monitoring, while [16] demonstrates their utility in automated program transformation through systematic inversion techniques.

In this work, we present a comprehensive formalization of SFTs. To address the extensibility challenges inherent in supporting diverse transition label theories, we adopt a refinement-based approach. At the abstraction level, transition labels are formalized through the fundamental mathematical concept of *sets*. This abstraction facilitates subsequent refinement to various representations of Boolean algebras, such as intervals and arithmetic predicates, while maintaining theoretical consistency.

The key operation of our formalization is the product operation between an SFT and an input regular language. Specifically, given an SFA \mathcal{A} representing a regular language and an SFT \mathcal{T} , we define the product operation $\mathcal{T} \times \mathcal{A}$ that characterizes the output language generated by \mathcal{T} when processing inputs from the language recognized by \mathcal{A} .

In the refinement level, we implemented transition labels using an interval-based representation to exemplify the refinement process. The formalization of interval algebra provides efficient set-theoretic operations—including membership checking, intersection, and difference computations—facilitating the refinement of transition labels from abstract sets to concrete intervals. Furthermore, leveraging the data refinement framework [20] in Isabelle/HOL, we store states and transitions using sophisticated data structures such as hashmaps and red-black trees, ensuring efficient automata manipulation.

To evaluate the effectiveness and efficiency of our formalization, we developed a string solver compliant with SMT-LIB language [4], focusing particularly on replacement operations. We evaluated our implementation using a set of benchmarks from the SMT-LIB repository [26]. The experimental results demonstrate that our formalization achieves computational efficiency in constraint-solving scenarios.

In summary, our formalization makes the following contributions:

1. We present the first formalization of symbolic finite transducers in Isabelle/HOL proof assistant.
2. We leverage the refinement framework to create an extensible SFT formalization that is capable of accommodates various boolean algebras, ensuring adaptability to future transition label developments.
3. We develop an interval algebra with efficient set-theoretic operations, enabling refinement of transition labels from abstract sets to concrete intervals.
4. We demonstrate the practical utility of our formalization through its application to string constraint solving, specifically in modeling replacement operations with verified correctness guarantees.

The remainder of this paper is organized as follows: Section 2 introduces the formalization of symbolic finite transducers. Section 3 presents the product operation at the abstract level. Section 4 details the algorithmic refinement of the product operation. Section 5 demonstrates the application to string constraint solving. Section 6 discusses related work. Section 7 concludes with future directions.

2 Formalization of SFTs

We begin by presenting a mathematical definition of SFTs [34], abstracting from the specific implementation details of Isabelle/HOL.

2.1 The Mathematical Definition of SFTs

Let \mathcal{U} be a multi-sorted carrier set or background universe, which is equipped with functions and relations over the elements. We use τ as a sort and \mathcal{U}^τ denotes the sub-universe of elements of type τ . We have a special type \mathbb{B} with $\mathcal{U}^{\mathbb{B}} = \{\top, \perp\}$, which corresponds to the boolean type.

A lambda term is defined as $\lambda x. t$ of type $\tau_1 \rightarrow \tau_2$. When τ_2 is \mathbb{B} , this lambda term is a predicate. Let ϕ be a predicate. We write $a \in \llbracket \phi \rrbracket$ if $\phi a = \top$. For non-predicate lambda terms, we view them as functions that generate output elements of type τ_2 given input terms of type τ_1 . With these notations and the above definitions, we can define SFTs as follows.

► **Definition 1** (Symbolic Finite Transducer). *A Symbolic Finite Transducer over $\tau_1 \rightarrow \tau_2$ is a quadruple $\mathcal{T} = (\mathcal{Q}, \Delta, \mathcal{I}, \mathcal{F})$, where*

- \mathcal{Q} is a finite set of states,
- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states,
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of accepting states,
- Δ is the set of transition relations. Each element in Δ is of the form (q, ϕ, f, q') or written as $q \xrightarrow{\phi, f} q'$, where q and q' are states in \mathcal{Q} . ϕ is a predicate of type $\tau_1 \rightarrow \mathbb{B}$. f is a lambda term of type $\tau_1 \rightarrow \tau_2$. f is called an output function.

For each transition $q \xrightarrow{\phi, f} q'$, if there exists an element $a \in \llbracket \phi \rrbracket$, where a is called an input, then the application $(f a)$ is the output.

SFTs accept an input word and generate an output word. This can be defined by *runs* of SFTs. An SFT run σ is a sequence $(q_0, \phi_0, f_0, q_1), (q_1, \phi_1, f_1, q_2), \dots, (q_{n-1}, \phi_{n-1}, f_{n-1}, q_n)$ such that $q_0 \in \mathcal{I}$ and $(q_i, \phi_i, f_i, q_{i+1}), 0 \leq i \leq n-1$ is a transition in Δ . σ is an accepting run when q_n is an accepting state.

For a word $w = a_0, \dots, a_{n-1}$, it is accepted by σ if and only if $a_i \in \llbracket \phi_i \rrbracket$ for $0 \leq i \leq n-1$. When w is accepted, run σ generates an output sequence $w' = f_0 a_0, \dots, f_{n-1} a_{n-1}$. We define:

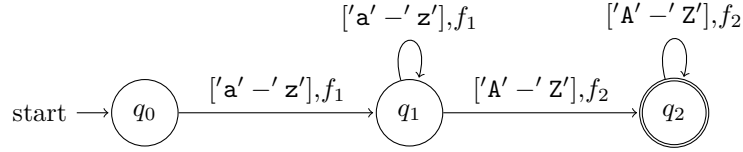
- $(a_0, (f_0 a_0)), \dots, (a_{n-1}, (f_{n-1} a_{n-1}))$ a *trace*,
- a_0, \dots, a_{n-1} the *input* of the trace and $(f_0 a_0), \dots, (f_{n-1} a_{n-1})$ the *output* of the trace.

If a_0, \dots, a_{n-1} is accepted by an accepting run in \mathcal{T} , we say that the trace is an accepting trace of \mathcal{T} . For a trace π , we denote its input as $in(\pi)$ and its output as $out(\pi)$. Given an SFT \mathcal{T} and a word w , we define the *product* operation of \mathcal{T} and w (denoted as $\mathcal{T} \times \{w\}$) as the set of outputs generated by \mathcal{T} with input w . More precisely,

$$\mathcal{T} \times \{w\} = \{w' \mid \exists \pi. \pi \text{ is an accepting trace of } \mathcal{T} \wedge in(\pi) = w \wedge out(\pi) = w'\}.$$

To make the operation *product* more general, we extend the operation to an SFT and a set of input words represented by a regular language, which can be denoted by an SFA \mathcal{A} . More precisely,

$$\mathcal{T} \times \mathcal{A} = \{w' \mid \exists w. w \in \mathcal{L}(\mathcal{A}) \wedge w' \in \mathcal{T} \times \{w\}\}, \text{ where } \mathcal{L}(\mathcal{A}) \text{ denotes the language of } \mathcal{A}.$$



■ **Figure 1** An example of SFT

Figure 1 illustrates an SFT that accepts words matching the regular expression $/['a' -' z'] + ['A' -' Z'] + /$. The transition labels utilize intervals of the form $[i-j]$, which are interpreted as predicates $\lambda x. i \leq x \leq j$. The output functions f_1 and f_2 perform case transformations: $f_1 = \lambda x. \text{toUpper}(x)$ converts lowercase letters to uppercase, while $f_2 = \lambda x. \text{toLower}(x)$ performs the inverse operation. For example, given the input string "bigSMALL", this SFT produces the output "BIGsmall".

2.2 The Isabelle/HOL Formalization of SFTs

As we have discussed the diversity of transition labels in Section 1, we now present an extensible formalization of SFTs that accommodates this variety. Our approach leverages the refinement framework (Refine_Monadic [19]) in Isabelle/HOL to achieve the flexibility and extensibility of transition labels modeling.

Figure 2 presents our formalization of SFTs in Isabelle/HOL. While the elements \mathcal{Q}_t , \mathcal{I}_t , and \mathcal{F}_t directly correspond to their counterparts (\mathcal{Q} , \mathcal{I} , and \mathcal{F}) in Definition 1, the transition relations are not exactly the same. The transition relations are formalized through LTTs (Labeled Transducer Transition System), where each transition is represented as a triple $'q \times ('a \text{ set option } \times' i) \times' q$. This representation reflects several key design decisions aimed at enhancing the abstraction and flexibility of our SFT formalization.

```

1 record ('q, 'a, 'i, 'b) NFT =
2   Qt :: "'q set"
3   Δt :: "('q, 'a, 'i) LTTS"
4   It :: "'q set"
5   Ft :: "'q set"
6   Mt :: "'i ⇒ ('a, 'b) Tlabel"
7 type_synonym ('q, 'a, 'i) LTTS = "('q × ('a set option × 'i) × 'q) set"
8 type_synonym ('a, 'b) Tlabel = "'a option ⇒ 'b set option"

```

■ **Figure 2** The formalization of *SFTs* in Isabelle/HOL

171 Firstly, *'a set option* is the input type of the transition, it accepts a set of elements of
 172 type *'a* or *None* corresponding to empty string ε . Accepting a set of *'a* elements aims to
 173 express the same but more abstract semantics of the input labels in Definition 1, in which
 174 an input label is a predicate. A predicate's semantics as introduced before represents a set
 175 of elements that make the predicate true. But predicates have various different forms. For
 176 instance, the interval $[1 - 9]$ represents the set $\{e \mid 1 \leq e \leq 9\}$. The predicate $\lambda b. b[7] = 1$,
 177 where b is a bit vector of length 8, denotes the set of bit vectors that have a 1 in the 7th
 178 position. All these different forms of labels are abstracted as sets in our formalization. The
 179 value *None* represents the empty string ε in our formalization, indicating a transition that
 180 consumes no input but may still produce output elements. This design choice facilitates the
 181 modeling of real-world applications in SFTs, as we will demonstrate in our application to
 182 string solvers.

183 The second element of type *'i* serves as an index into the output function space. We use
 184 indices instead of functions themselves to enable the reuse of the same output functions for
 185 different transitions. The mapping M_t associates each index with a specific output function.
 186 These output functions, formalized by *Tlabel*, map a single input element to a set of possible
 187 output elements rather than to a single element. This design enables non-deterministic
 188 output behavior, where the transducer may select any element from the output set randomly
 189 or according to specified criteria. Additionally, output functions can produce the empty
 190 string ε by returning *None*, providing further flexibility in transition behavior.

191 3 The Product Operation of SFTs

192 In this section, we formalize the product operation between an SFT \mathcal{T} and an SFA \mathcal{A} ,
 193 denoted as $\mathcal{T} \times \mathcal{A}$ in Section 2. An additional consideration in this operation is the presence
 194 of ε -transitions in our SFT formalization, which implies that the resulting automata may
 195 also contain ε -transitions. Since CertiStr [17] does not include a formalization of SFAs
 196 with ε -transitions, we extend the framework in CertiStr with a formalization of symbolic
 197 SFAs with ε -transitions (denoted as ε SFA) and provide a verified conversion to standard
 198 SFAs. In this paper, we present only the definitions of ε SFAs and SFAs, while the complete
 199 formalization, including correctness proofs, is available in our Isabelle development.

200 Figure 3 presents the formalization of both SFAs and ε SFAs using Isabelle/HOL record
 201 types *NFA* and *eNFA*, respectively. The ε SFA formalization extends the standard SFA structure
 202 by introducing Δ'_e , which captures ε -transitions as pairs of states, while maintaining the
 203 same labeled transition relation Δ as in standard SFAs.

204 Having established these foundational definitions, we can now formalize the product
 205 operation.

```

1 record ('q, 'a) NFA =
2     Q :: "'q set"
3     Δ :: "('q, 'a) LTS"
4     I :: "'q set"
5     F :: "'q set"
6
7 record ('q, 'a) eNFA =
8     Qe :: "'q set"
9     Δe :: "('q, 'a) LTS"
10    Δ'e :: "('q *' q) set"
11    Ie :: "'q set"
12    Fe :: "'q set"
13
14 type_synonym ('q, 'a) LTS = "'q ×' a set ×' q"
    
```

■ **Figure 3** The formalization of ε SFAs and SFAs

```

1 definition productT :: "('q, 'a, 'i, 'b)" NFT ⇒ ('q, 'a) NFA ⇒
2     (('a, 'b) Tlabel ⇒' a set ⇒' b set option) ⇒
3     ('q ×' q, 'b) eNFA where
4     "productT T A F = (
5         Qe = Qt T × Q A,
6         Δe = {((p, p'), the (((Mt T) f) None), (q, p')) | p, p', q, f. p' ∈ Q A ∧
7             (p, (None, f), q) ∈ Δt T ∧ ∃S. (Mt T) f None = Some S} ∪
8             {((p, p'), the (F ((Mt T) f) (σ1 ∩ σ2)), (q, q')) | p, p', q, σ1, σ2, q', f.
9             (p, (Some σ1, f), q) ∈ Δt T ∧ (p', σ2, q') ∈ Δ A ∧ σ1 ∩ σ2 ≠ ∅ ∧
10             ∃S. F ((Mt T) f) (σ1 ∩ σ2) = Some S},
11         Δ'e = {((p, p'), (q, p')) | p, p', q, f. p' ∈ Q A ∧
12             (p, (None, f), q) ∈ Δt T ∧ (Mt T) f None = None} ∪
13             {((p, p'), (q, q')) | p, p', q, σ1, σ2, q', f.
14             (p, (Some σ1, f), q) ∈ Δt T ∧ (p', σ2, q') ∈ Δ A ∧ σ1 ∩ σ2 ≠ ∅ ∧
15             ∃x ∈ (σ1 ∩ σ2). ((Mt T) f) (Some x) = None},
16         Ie = It T × I A
17         Fe = Ft T × F A )"
    
```

■ **Figure 4** The formalization of product operation

Figure 4 depicts the abstract level formalization for the product of an SFT and an SFA. The parameters \mathcal{T} and \mathcal{A} are an SFT and an SFA, respectively. The result of the product operation is an ε SFA. But we need to explain the role of parameter F . The output function f for each transition in Δ_t is of type $'a \text{ option} \Rightarrow 'b \text{ set option}$, which applies to a *single* element of type $'a$ or ε . F extends f to apply f to a set of elements. More precisely, let f be an output function, the semantics of F is defined as follows:

$$F f A = \text{Some } \left(\bigcup_{a \in A} (\text{if } f a = \text{Some } S \text{ then } S \text{ else } \emptyset) \right)$$

214 The transition relations Δ_e and Δ'_e are determined by considering two distinct cases
 215 based on the nature of transitions in the SFT: ε -transitions and non- ε -transitions. In both
 216 cases, the transition labels in the resulting ε SFA are derived from the composition of the
 217 SFT's output function and the SFA's input labels. Let us consider Δ_e first.

- 218 1. When $(p, (\text{None}, f), q)$ is a transition in Δ_t , i.e. the input character is ε , and $f \neq \text{None}$
 219 None . Consequently, the SFA \mathcal{A} remains in its current state, and the product transition
 220 produces the output " $((\mathcal{M}_t \mathcal{T}) f) \text{None}$ ". Remember that f is just an index, $(\mathcal{M}_t \mathcal{T}) f$
 221 is the output function.
- 222 2. When $(p, (\text{Some } \sigma_1, f), q)$ is a transition in Δ_t , synchronization is possible only with SFA
 223 transitions that share characters with σ_1 , i.e., $\sigma_1 \cap \sigma_2 \neq \emptyset$, where σ_2 represents the input
 224 label of the corresponding SFA transition. The resulting output is $F((\mathcal{M}_t \mathcal{T}) f) (\sigma_1 \cap \sigma_2)$.

225 The transitions in Δ'_e follow a similar pattern with analogous cases for ε and non- ε
 226 transitions.

227 To establish the correctness specification of the product operation, we begin by formalizing
 228 the concept of SFT *traces* as introduced in Definition 1. In our formalization, traces are
 229 represented by the type $('a \text{ option} \times 'b \text{ option}) \text{ list}$, as shown in Figure 5. Given a trace
 230 π , we define two key projection functions: (1) `inputE`, which corresponds to $\text{in}(\pi)$ and
 231 extracts the input sequence (2) `outputE`, which corresponds to $\text{out}(\pi)$ and extracts the
 232 output sequence.

233 The definition `outputL` generalizes `outputE` to characterize the set of all possible outputs
 234 that an SFT \mathcal{T} can generate when processing inputs from the language accepted by the SFA
 235 \mathcal{A} . The reachability of a trace π between states q and q' in an SFT \mathcal{T} is verified by the
 236 predicate `LTTS_reachable` $\mathcal{T} \ q \ \pi \ q'$.

```

1 fun inputE :: ('a option × 'b option) list ⇒ 'a list where
2   "inputE [] = []" |
3   "inputE ((Some a, _) # l) = a # (inputE l)" |
4   "inputE ((None, _) # l) = (inputE l)"
5
6 fun outputE :: "('a option × 'b option) list ⇒ 'b list" where
7   "outputE [] = []" |
8   "outputE ((_, Some a) # l) = a # (outputE l)" |
9   "outputE ((_, None) # l) = (outputE l)"
10
11 definition outputL :: "('q, 'a, 'i, 'b) NFT ⇒ ('q, 'a) NFA ⇒ 'b list set"
12   where
13     "outputL  $\mathcal{T} \ \mathcal{A} = \{\text{outputE } \pi \mid \pi \ q \ q'. \ q \in \mathcal{I}_t \ \mathcal{T} \wedge q' \in \mathcal{F}_t \ \mathcal{T} \wedge$ 
14       LTTS_reachable  $\mathcal{T} \ q \ \pi \ q' \wedge \text{inputE } \pi \in \mathcal{L} \ \mathcal{A}\}$ "

```

■ **Figure 5** The formalization of traces in SFTs

237 Figure 6 presents Lemma `productT_correct`, which establishes the correctness of the
 238 product operation. The lemma's assumptions, `F_ok1` and `F_ok2`, specify the essential
 239 properties of function `F`. The assumptions `NFT_wf` \mathcal{T} and `NFA` \mathcal{A} ensure that the SFT \mathcal{T}
 240 and the SFA \mathcal{A} are well-formed. The conclusion, marked by `shows`, demonstrates that
 241 the language of the constructed ε SFA (\mathcal{L}_e denotes the language of ε SFA) from $\mathcal{T} \times \mathcal{A}$
 242 coincides with the mathematical semantics defined by `outputL`, thereby establishing semantic
 243 preservation of the product construction.

```

1 lemma productT_correct:
2   fixes  $\mathcal{T}$   $\mathcal{A}$  F
3   assumes F_ok1: " $\forall f s. (\forall e \in s. f \text{ (Some } e) = \text{None}) \longleftrightarrow F f s = \text{None}$ "
4     and F_ok2: " $\forall f s. F f s \neq \text{None} \longrightarrow F f s =$ 
5       Some  $(\cup \{S \mid e \in S. e \in s \wedge f \text{ (Some } e) = \text{Some } S\})$ "
6     and wfTA: "NFT_wf  $\mathcal{T} \wedge \text{NFA } \mathcal{A}$ "
7   shows " $\mathcal{L}_e(\text{productT } \mathcal{T} \mathcal{A} F) = \text{outputL } \mathcal{T} \mathcal{A}$ "

```

■ **Figure 6** The correctness lemma of the product operation

244 4 Algorithm Level Refinement

245 Having established the abstract definition of the SFT product operation in Section 3, we
 246 now present its algorithmic refinement. This section introduces an efficient implementation
 247 of the product construction and refines the abstract representation of transition labels to a
 248 concrete interval algebra, enabling practical computation.

249 4.1 Intervals

250 An interval is defined as a pair (i, j) (represented as $[i - j]$ as well in the paper) representing the
 251 set $\{e \mid i \leq e \leq j\}$. To achieve greater expressiveness, our formalization extends this notion to
 252 interval lists of the form $[(i_1, j_1), \dots, (i_n, j_n)]$, which denote the set $\bigcup_{1 \leq k \leq n} \{e \mid i_k \leq e \leq j_k\}$.
 253 This generalization offers two key advantages: it enables more compact representation of
 254 transitions in SFAs and SFTs through merging, and it allows for efficient handling of interval
 255 operations without unnecessary splitting. For example, the set difference between intervals
 256 $(1, 5)$ and $(3, 4)$ can be directly represented as the interval list $[(1, 2), (5, 5)]$, which is also an
 257 interval.

258 Throughout the following discussion, we use the term "interval" to refer to interval lists.
 259 Our formalization provides a collection of interval operations through the following interface:

- 260 1. **semIs** i : Defines the semantic interpretation of an interval as a set. For an interval (i, j) ,
 261 **semIs** $(i, j) = \{e \mid i \leq e \leq j\}$.
- 262 2. **emptyIs** i : Tests whether an interval represents an empty set, i.e., whether **semIs** $i = \emptyset$.
- 263 3. **nemptyIs** i : Tests whether an interval represents a non-empty set, i.e., whether **semIs** $i \neq$
 264 \emptyset .
- 265 4. **intersectIs** $i_1 i_2$: Computes the intersection of two intervals, yielding an interval i such
 266 that **semIs** $i = \text{semIs } i_1 \cap \text{semIs } i_2$.
- 267 5. **diffIs** $i_1 i_2$: Computes the set difference of two intervals, yielding an interval i such
 268 that **semIs** $i = \text{semIs } i_1 \setminus \text{semIs } i_2$.

269 To facilitate formal reasoning and optimize performance, we introduce a canonical form
 270 for interval. An interval $[(i_1, j_1), \dots, (i_n, j_n)]$ is in canonical form if it satisfies two key
 271 properties:

- 272 1. Each (i_k, j_k) is well-formed: $i_k \leq j_k$ for all $k \in \{1, \dots, n\}$
- 273 2. Intervals are ordered and non-overlapping: $j_k < i_{k+1}$ for all $k \in \{1, \dots, n-1\}$

274 We prove that all interval operations preserve canonical form when applied to canonically-
 275 formed inputs. This invariant serves two purposes: it simplifies formal proofs by eliminating
 276 the need to reason about malformed or overlapping intervals, and it enables more efficient
 277 implementations of interval operations by reducing the number of cases to consider.

4.2 Algorithmic Implementation of the Product Operation

In this subsection, we present the algorithmic implementation of the product operation between an SFT and an SFA¹. Figure 7 illustrates the core algorithm `productT_impl`, which is implemented using the *Refine_Monadic* framework. Note that in the implementation, there are some refined operations: `nfa_states`, `nfa_trans`, `nfa_initial`, `nfa_accepting`, and `nft_tranfun`. These are corresponding to the states, transitions, initial states, accepting states, and output function mapping of the SFT.

The operation `prods_imp`, shown in Figure 8, computes the Cartesian product of two state sets. This function employs the `FOREACH` construct, a higher-order iteration operator analogous to OCaml's `Set.fold`. Specifically, given a set S , a function f of type $'a \Rightarrow 'b \Rightarrow 'b$, and an initial accumulator I of type $'b$, the expression `FOREACH S f I` systematically applies f to each element in S , accumulating results in a principled manner.

```

1 definition productT_impl where
2   "productT_impl  $\mathcal{T}$   $\mathcal{A}$  F fe = do {
3     Q  $\leftarrow$  prods_imp (nft_states  $\mathcal{T}$ ) (nfa_states  $\mathcal{A}$ );
4     (D1, D2)  $\leftarrow$  trans_comp_imp (nft_tranfun  $\mathcal{T}$ ) F fe
5       (nft_trans  $\mathcal{T}$ ) (nfa_trans  $\mathcal{A}$ ) (nfa_states  $\mathcal{A}$ );
6     I  $\leftarrow$  prods_imp (nft_initial  $\mathcal{T}$ ) (nfa_initial  $\mathcal{A}$ );
7     F  $\leftarrow$  prods_imp (nft_accepting  $\mathcal{T}$ ) (nfa_accepting  $\mathcal{A}$ );
8     RETURN (Q, D1, D2, I, F)
9   }"
```

Figure 7 The computation of SFT product

```

1 definition prods_imp where
2   "prods_imp Q1 Q2 =
3     FOREACH {q. q  $\in$  Q1} ( $\lambda$  q Q. do {
4       S  $\leftarrow$  FOREACH {q. q  $\in$  Q2}
5         ( $\lambda$  q' Q'. RETURN ({(q,q')}  $\cup$  Q'))  $\emptyset$ ;
6       RETURN (Q  $\cup$  S)
7     })  $\emptyset$ "
```

Figure 8 The computation of Cartesian product of two state sets

A central algorithmic challenge in our implementation lies in the computation of transition sets D1 (corresponding to Δ_e) and D2 (corresponding to Δ'_e) in Figure 7. As shown in Figure 9, we implement this computation through the function `trans_comp_imp`, which computes the synchronization of transitions between the SFT and SFA. This function decomposes the synchronization process into two distinct cases, each handled by a specialized function:

1. `subtrans_comp_ε`: Processes ε -transitions in the SFT, where transitions consume no input but may produce output

¹ For clarity of presentation, we show a simplified version of the Isabelle/HOL implementation while preserving the essential algorithmic structure.

```

1 definition trans_comp_imp where
2   "trans_comp_imp M F fe T1 T2 Q =
3     FOREACH {t. t ∈ T1}
4       (λ (q, (α, f), q') (D1, D2).
5         (if (α = None) then
6           (subtrans_comp_ε M q f q' F fe T2 D1 D2)
7         else
8           (subtrans_comp M q (the α) f q' F fe T2 D1 D2)
9         )) (∅, ∅)"

```

■ **Figure 9** The computation of `trans_comp_imp`

- 297 2. `subtrans_comp`: Processes standard transitions in the SFT, where both input consump-
 298 tion and output generation may occur

```

1 definition subtrans_comp where
2   "subtrans_comp M q α f q' F fe T D1 D2 =
3     FOREACH
4       {t. t ∈ T} (λ (q1, α', q1') (D1, D2).
5         (if (notEmptyIs (intersectIs α α')) then do {
6           D1 ← (if (F (M f) (intersectIs α α')) ≠ None)
7             then
8               let αi = the (F (M f) (intersectIs α α')) in
9               RETURN {(q,q1), αi, (q',q1'))} ∪ D1
10          else RETURN D1;
11         D2 ← (if fe (M f) (intersectIs α α') then
12           RETURN {(q,q1), (q',q1'))} ∪ D2 else RETURN D2);
13         RETURN (D1, D2)
14       }
15     else (RETURN (D1, D2)))) (D1, D2)"

```

■ **Figure 10** The computation of `subtrans_comp`

299 We now present the implementation of `subtrans_comp` in detail, as shown in Figure
 300 10 (the implementation of `subtrans_comp_ε` follows analogous principles). For a transtion
 301 $(q, (\text{Some } \alpha, f), q')$ in the SFT, this function traverses all transitions in the SFA, represented
 302 by the set T . For each inpution $(q_1, \alpha', q'_1) \in T$, the function performs two key operations
 303 when the intersection of input labels is non-empty ($\alpha \cap \alpha' \neq \emptyset$, verified using `notEmptyIs`):

- 304 1. Computes non- ε -transitions ($D1$): When the output function applied to the intersection
 305 $\alpha \cap \alpha'$ yields a non-empty set, a new transition is added to $D1$ with the computed output
 306 label.
 307 2. Generates ε -transitions ($D2$): When there exists at least one input in the intersection
 308 $\alpha \cap \alpha'$ that produces an empty string (verified by checking if $M f$ maps any element to
 309 `None`. The checking is implemented by `fe`), a corresponding ε -transition is added to $D2$.

310 The correctness of the product computation is established through a refinement proof,
 311 demonstrating that `productT_imp` (Figure 7) correctly implements the abstract specification

```

1  lemma productT_imp_correct:
2  assumes finite_TT: "finite ( $\Delta_t \mathcal{T}$ )"
3      and finite_TA: "finite ( $\Delta \mathcal{A}$ )"
4      and finite_Q: "finite ( $\mathcal{Q} \mathcal{A}$ )"
5      and finite_TQ: "finite ( $\mathcal{Q}_t \mathcal{T}$ )"
6      and finite_I: "finite ( $\mathcal{I} \mathcal{A}$ )"
7      and finite_TI: "finite ( $\mathcal{I}_t \mathcal{T}$ )"
8      and finite_F: "finite ( $\mathcal{F} \mathcal{A}$ )"
9      and finite_TF: "finite ( $\mathcal{F}_t \mathcal{T}$ )"
10 shows "productT_imp  $\mathcal{T} \mathcal{A} F$  fe  $\leq$  SPEC ( $\lambda A. A = \text{productT } \mathcal{T} \mathcal{A} F$ )"

```

■ **Figure 11** The refinement relation between `productT_imp` and `productT`

312 `productT` (Figure 4). This refinement relationship is formally specified in Figure 11, where
 313 we leverage the *Refine_Monadic* framework’s data refinement.

314 The refinement is expressed through the relation $\mathbf{C} \leq \text{SPEC } \mathbf{A}$, which asserts that the
 315 concrete implementation \mathbf{C} is an element of the abstract specification \mathbf{A} . More precisely, the
 316 concrete implementation must produce an ε SFA that is structurally equivalent or isomorphic
 317 to the one produced by the abstract algorithm `productT`.

318 To establish this equivalence, we must prove that the ε SFAs produced by `productT_imp`
 319 and `productT` are isomorphic in all essential components: the set of states, transition relations,
 320 ε -transition relations, initial and accepting state sets.

321 The implementation of `productT_imp` follows *Refine_Monadic* framework’s interfaces for
 322 sets to store states and transition relations. The *Refine_Monadic* framework provides a way
 323 to automatically refine these interfaces to more efficient data structure, such as red-black
 324 trees or hashmaps. In our formalization, we refine the sets of storing states and transitions
 325 to red-black trees.

326 5 An Application to String Solving

327 In this section, we demonstrate the practical application of our formalized SFTs to string
 328 constraint solving, specifically focusing on replacement operations. While CertiStr [17]
 329 provides a verified framework for string constraint solving using SFAs, its capabilities are
 330 inherently limited by the expressiveness of SFAs. In particular, SFAs cannot directly model
 331 string transformation operations such as replacement.

332 Our string solver is based on CertiStr [17] by extending it with the modeling of replacement
 333 operations. CertiStr exploits forward-propagation to solve the string constraints. Our SFT
 334 modeling for replacement operations can seamlessly be integrated into CertiStr’s forward-
 335 propagation framework.

336 5.1 Modeling the Replacement Operation

337 The string replacement operation, denoted as `replace(str, pattern, replacement)`, is a
 338 fundamental string transformation that takes three parameters:

- 339 ■ **str**: The input string to be transformed.
- 340 ■ **pattern**: A regular expression defining the matching criteria.
- 341 ■ **replacement**: The string to be substituted for the matched substring.

23:12 Certified Symbolic Transducer with Applications in String Solving

The semantics of the replacement operation can be formally characterized by two distinct cases:

1. When there exists at least one substring s' in str such that s' matches pattern , then s' is replaced with replacement .
2. When no substring of str matches pattern , the operation returns str unchanged.

The first case can be modeled by SFTs. We illustrate this case by an example. Given a replacement operation $\text{replace}(s, /[0-9]+/, \text{"NUM"})$, which means replacing a occurrence of the substring that matches the regular expression $/[0-9]+/$ with the string "NUM" . We model this replacement operation by an SFT with the following 3 steps:

1. First, construct an SFA that recognizes the regular expression pattern $/[0-9]+/$. Transform this SFA into an SFT by augmenting each transition with an output function $f = \lambda x. \text{None}$ that produces the empty string.
2. Second, construct an SFA that accepts the replacement string "NUM" . Since a constant string can be viewed as a specialized regular expression, we can construct its SFA representation. Convert this SFA into an SFT by adding ε -transitions and appropriate output functions that emit the characters of "NUM" in sequence.
3. Finally, compose the two SFTs through concatenation and augment the resulting transducer with self-loop transitions at the initial and final states. These additional transitions, labeled with Σ (the set of all characters in the alphabet) and the identity function $\text{id} = \lambda x. x$, enable the SFT to process arbitrary prefixes and suffixes of the input string while preserving the replacement behavior on matched substrings.

Step 1. Figure 12 illustrates the construction of the pattern-matching component. The left side shows the SFA that recognizes the regular expression $/[0-9]+/$, while the right side presents its transformation into an SFT. This transformation is achieved by augmenting each transition with the output function $f = \lambda x. \text{None}$, which consistently produces the empty string, effectively "consuming" the matched digits without generating output.

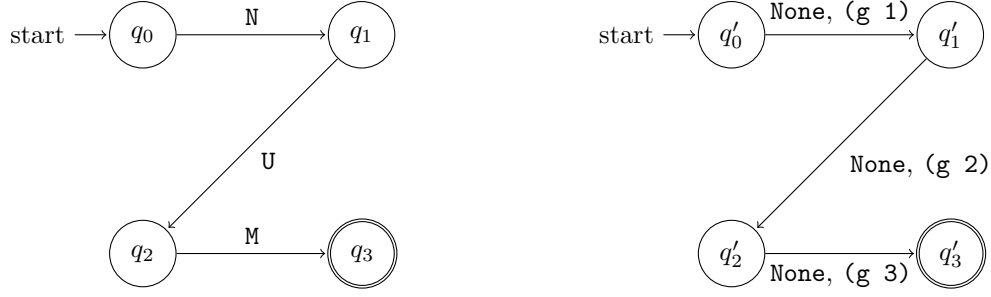


■ **Figure 12** Corresponding SFA and SFT for $/[0-9]+/$

Step 2. Figure 13 depicts the automata for the replacement string "NUM" . The left side shows the SFA that accepts this constant string, while the right side presents its SFT transformation. The transformation employs an indexed output function g that emits characters of the replacement string sequentially:

$$g = \lambda i \ x. \text{match } i \text{ with } \begin{cases} 1 \mapsto [(78, 78)] & (\text{ASCII for 'N'}) \\ 2 \mapsto [(85, 85)] & (\text{ASCII for 'U'}) \\ 3 \mapsto [(77, 77)] & (\text{ASCII for 'M'}) \\ _ \mapsto \text{None} \end{cases}$$

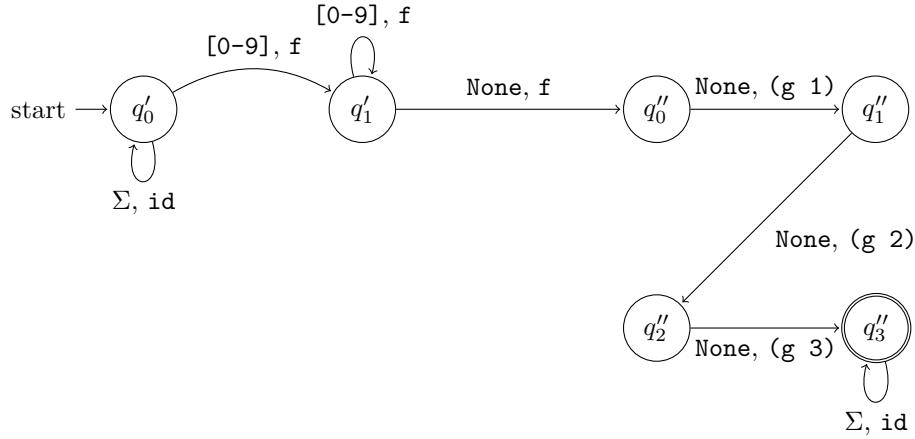
This function maps transition indices to their corresponding character outputs, using ASCII codes to represent the string "NUM" character by character.



■ **Figure 13** Corresponding SFA and SFT for "NUM"

375 *Step 3.* Figure 14 presents the complete SFT construction, obtained by composing the
 376 pattern-matching SFT (Fig. 12) with the replacement-generating SFT (Fig. 13). The
 377 composition process involves two key modifications:

- 378 1. Connect the two SFTs by adding ε -transitions (labeled with "None, f") from each accepting
 379 state of the pattern-matching SFT to the initial state of the replacement-generating SFT
- 380 2. Augment the resulting transducer with self-loop transitions at both ends, labeled with
 381 " Σ , id", where Σ represents the full alphabet. For the string solver, Σ is the set of all
 382 unicode characters. These transitions enable the SFT to process arbitrary input prefixes
 383 and suffixes while preserving the matched substring for replacement



■ **Figure 14** The SFT for the replacement operation `replace(s,/[0-9]+/,NUM)`

384 Having constructed the SFT, we can now compute the forward image of the replacement
 385 operation. Consider the string constraint $s' = \text{replace}(s, /[0-9]+/, \text{"NUM"})$, where we aim
 386 to characterize the possible values of s' . Let \mathcal{T} denote the constructed SFT modeling the
 387 replacement operation, and let \mathcal{A} be the SFA representing the domain of possible values for
 388 the input string s . The forward image of this replacement operation is given by the product
 389 $\mathcal{T} \times \mathcal{A}$, which precisely captures the set of all possible output strings that can be produced
 390 by applying the replacement operation to any input string accepted by \mathcal{A} . Our string solver
 391 uses this forward image to do forward propagation for further string solving.

392 *Comparison with SMT-LIB Semantics.* It is important to note that our formalization
 393 differs from the standard SMT-LIB semantics for the replacement operation. In SMT-LIB, the
 394 `str.replace` operation is defined to replace only the *first* occurrence of a substring matching

the given pattern. In contrast, our semantics allows for a more general interpretation where any matching substring may be replaced. But it still replaces only one occurrence of them.

5.2 Experiments

We have implemented CertiStrR, an extension of CertiStr [17], to support string replacement operations. While the core solving algorithm maintains the certification guarantees of CertiStrR, the frontend components are implemented using established non-certified OCaml libraries: dolmen [24] for SMT-LIB parsing and ocaml-re-nfa [35] for regular expression to NFA conversion.

CertiStrR implements two kinds of SMT-LIB's replacement operations: `str.replace s p r`: a string-based replacement where pattern `p` is a constant string. `str.replace_re s p r`: a regular expression-based replacement where pattern `p` is a regular expression.

Listing 1 demonstrates the regular expression variant through an example that replaces numeric sequences with the string "NUM". The constraints are satisfiable because the input string `a = "2024,2025"` contains a substring "2025" that matches the regular expression `re.+ (re.range "0" "9")` (equivalent to `/[0-9]+/`), and replacing this match with "NUM" yields the expected output string `b = "2024,NUM"`.

As discussed in the previous subsection, our replacement semantics differs from the SMT-LIB standard semantics. Run the SMT solver CVC5 [30] with the code in Listing 1, the result is UNSAT because CVC5 only matches the first occurrence of the pattern.

```
(set-logic QF_S)
(declare-fun a () String)
(declare-fun b () String)
(assert (= a "2024,2025"))
(assert (= b "2024,NUM"))
(assert (= b (str.replace_re a (re.+ (re.range "0" "9")) "
NUM")))
(check-sat)
```

■ Listing 1 Example SMT-LIB Code

We evaluate CertiStrR using benchmarks from SMT-LIB 2024 [26], focusing on the QF_S and QF_SLIA logic fragments. The benchmarks are divided into two categories based on the replacement operations: (1) string-based replacement (`str.replace`) and (2) regular expression-based replacement (`str.replace_re`). Due to CertiStrR's current front-end limitations in supporting the full SMT-LIB language, we preprocess certain operations. For instance, conjunctive assertions of the form `(assert (and c1 c2))` are decomposed into separate assertions `(assert c1)` and `(assert c2)` when `c1` and `c2` are string constraints supported by CertiStrR.

	SAT	UNSAT	Inconclusive	Time	Number of Tests
<code>replace_str</code>	142	173	8	0.27	323
<code>replace_re</code>	84	4	10	0.36	98

■ Table 1 Experimental results

The experimental evaluation was conducted on a laptop with an Apple M4 processor and 24 GB of memory, with a time limit of one minute per test. The results show average

execution times of 0.27 seconds for `str.replace_str` and 0.36 seconds for `str.replace_re` operations. Test outcomes were classified into three categories: SAT (satisfiable), UNSAT (unsatisfiable), and Inconclusive. An "Inconclusive" result indicates that the solver cannot determine satisfiability, not due to timeout (all tests are finished in one minute) but rather due to inherent limitations of the forward-propagation algorithm inherited from CertiStr [17]. Specifically, when the string constraints do not satisfy the tree property defined in [17], the forward-propagation algorithm may be unable to reach a definitive conclusion, even when the variable domains remain non-empty after propagation.

Our performance analysis revealed that while the SFT-based replacement operation modeling is efficient, the primary computational bottleneck stems from automata accumulation during forward-propagation. Consider the following string constraints:

$$x = x_1 ++ x_2; x = \text{replace}(x_3, p, r); x = x_4; x = \text{replace_re}(x_5, p_1, r_1);$$

where `++` denotes string concatenation. The variable x appears multiple times on the left-hand side of the equations, causing the forward-propagation algorithm to accumulate automata representations for all constraints: $x_1 ++ x_2$, $\text{replace}(x_3, p, r)$, x_4 , and $\text{replace_re}(x_5, p_1, r_1)$. Each accumulation step requires computing the product of the current automaton with the previous result. Given that the product operation has a worst-case complexity of $O(n^2)$, where n represents the automaton size, this repeated accumulation can lead to state explosion.

5.3 Effort of Certified Development

We discuss the effort of certified SFT development in this subsection. Table 2 provides an overview. The abstract-level development means all formalizations in Section 2 and 3, including SFTs and ε SFAs. The implementation-level development means all formalizations in Section 4 including the refinement of the product operation. The last row corresponds to the effort of the interval formalization. The most difficult part is the correctness proof of the product operation at the abstract level (Figure 6). Interval formalization complexity dramatic increase in CertiStrR compared to the interval formalization in CertiStr [17] due to the extension of intervals to a list.

	Definitions	Lemmas	Proofs (lines of code)
Abstract-level	17	21	3274
Implementation-level	51	43	2700
Interval	15	29	1500

Table 2 Overview of the effort of certified development

6 Related Work

Symbolic Automata and Transducers. Symbolic Automata and Transducers [12, 34, 11, 10, 27] represent a significant advancement in automata theory, offering improved efficiency in operations and enhanced expressiveness through algebraic theories that support infinite alphabets. This symbolic framework has been progressively extended to accommodate more complex structures: symbolic tree transducers [32] handle hierarchical data structures, while symbolic pushdown automata [9] manage nested word structures. Recent developments in 2024 have further expanded the scope of symbolic techniques to include Büchi automata and

omega-regular languages [31], enabling verification of infinite-state systems and analysis of non-terminating computations.

Applications of Symbolic Transducers. The efficiency, scalability, and expressive power of symbolic automata and transducers have led to their widespread adoption in numerous real-world applications. These applications span diverse domains, including: constraint solving for program analysis [33], security-critical sanitizer analysis for web applications [15], runtime verification of system behaviors [36], and automated program inversion for software transformation [16]. Each application leverages the symbolic approach’s ability to handle complex patterns and infinite alphabets efficiently.

Formalization of Symbolic Automata and Transducers. While classical automata theories have been extensively formalized in interactive theorem provers [29, 18, 7, 13], with some work on transducer formalization [21], the symbolic variants of automata and transducers remain largely unexplored in formal verification. To our knowledge, CertiStr [17] represents the only existing work on symbolic automata formalization in a proof assistant. Our work advances this frontier by extending the formal treatment to both SFTs and ε SFAs.

String Solving. A significant application of our certified transducer framework is in string constraint solving, a field that has seen intensive research development over the past decade. While our work provides formal verification guarantees, there exists a rich ecosystem of non-certified string solvers, each with distinct capabilities: Kaluza [23] specializes in JavaScript analysis, CVC5 [30], Z3-str3 [5] builds on the Z3 framework, S3P [28], Ostrich [8], and SLOTH [14]. As more and more bugs have been uncovered in existing string solvers [6] and SMT solvers [22] We believe that our work will benefit the community by providing a formal foundation for string solvers development.

Certified SMT Solvers. Beyond string theories, certification efforts in SMT solving have extended to other domains. For example, the work by Shi et al. [25], who developed a certified SMT solver for quantifier-free bit-vector theory, demonstrating the broader applicability of interactive theorem proving in certified SMT solver development.

7 Conclusion

In this paper, we have presented the first formalization of symbolic finite transducers in the proof assistant Isabelle/HOL. Our formalization provides flexible interfaces that facilitate diverse applications through two key features: support for ε -transitions in both inputs and outputs, and extensibility to various boolean algebras via the refinement framework.

To demonstrate the practical utility of our formalization, we developed CertiStrR, an extension of CertiStr [17]. This implementation adds support for string replacement operations and has been evaluated on benchmarks from SMT-LIB 2024 [26]. The experimental results confirm both the efficiency and effectiveness of our approach. While we focused on string solving applications, our extensible framework for transition labels is broadly applicable to other domains, including program verification.

Future work includes extending our symbolic formalization in two key directions: (1) Enriching the boolean algebra framework to support more complex theories, such as arithmetic constraints. (2) Incorporating prioritized transitions into SFTs, which would enable precise modeling of SMT-LIB’s standard semantics for replacement operations, particularly the first-match behavior. These extensions will further enhance the expressiveness and practical applicability of our formalization while maintaining its formal verification guarantees.

References

- 1 Coq Homepage. <https://coq.inria.fr/>.
- 2 Isabelle Proof Assistant, 2013. <https://isabelle.in.tum.de/>.
- 3 John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Sørensen, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018. doi:10.23919/FMCAD.2018.8602994.
- 4 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.
- 5 Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017. doi:10.23919/FMCAD.2017.8102241.
- 6 Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 45–51. Springer, 2018. doi:10.1007/978-3-319-96142-2_6.
- 7 Julian Brunner. Transition Systems and Automata Isabelle Library. Arch. Formal Proofs, 2017. https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html.
- 8 Taolue Chen, Alejandro Flores-Lamas, Matthew Hague, Zhilei Han, Denghang Hu, Shuanglong Kan, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.*, 6(POPL):1–31, 2022. doi:10.1145/3498707.
- 9 Loris D’Antoni and Rajeev Alur. Symbolic visibly pushdown automata. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014. doi:10.1007/978-3-319-08867-9_14.
- 10 Loris D’Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. In Sam Staton, editor, *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018*, volume 341 of *Electronic Notes in Theoretical Computer Science*, pages 79–99. Elsevier, 2018. URL: <https://doi.org/10.1016/j.entcs.2018.03.017>, doi:10.1016/J.ENTCS.2018.03.017.
- 11 Loris D’Antoni and Margus Veanes. Minimization of symbolic automata. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*, pages 541–554. ACM, 2014. doi:10.1145/2535838.2535849.
- 12 Loris D’Antoni and Margus Veanes. The power of symbolic automata and transducers. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 47–67. Springer, 2017. doi:10.1007/978-3-319-63387-9_3.
- 13 Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. A constructive theory of regular languages in coq. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2013. doi:10.1007/978-3-319-03545-1_6.

- 565 14 Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String
566 constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*,
567 2(POPL):4:1–4:32, 2018. doi:10.1145/3158092.
- 568 15 Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes.
569 Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San*
570 *Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL:
571 http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf.
- 572 16 Qinheping Hu and Loris D’Antoni. Automatic program inversion using symbolic transducers.
573 In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN*
574 *Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona,*
575 *Spain, June 18-23, 2017*, pages 376–389. ACM, 2017. doi:10.1145/3062341.3062345.
- 576 17 Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. Certistr: a
577 certified string solver. In Andrei Popescu and Steve Zdancewic, editors, *CPP ’22: 11th ACM*
578 *SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA,*
579 *January 17 - 18, 2022*, pages 210–224. ACM, 2022. doi:10.1145/3497775.3503691.
- 580 18 P. Lammich. The CAVA automata library. *Arch. Formal Proofs*, 2014, 2014.
- 581 19 Peter Lammich. Refinement for monadic programs. *Archive of Formal Proofs*, January 2012.
582 https://isa-afp.org/entries/Refine_Monadic.html, Formal proof development.
- 583 20 Peter Lammich. Automatic Data Refinement. In Sandrine Blazy, Christine Paulin-Mohring,
584 and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference,*
585 *ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in*
586 *Computer Science*, pages 84–99. Springer, 2013. doi:10.1007/978-3-642-39634-2_9.
- 587 21 Alexander Lochmann, Bertram Felgenhauer, Christian Sternagel, René Thiemann, and Thomas
588 Sternagel. Regular tree relations. *Arch. Formal Proofs*, 2021, 2021. URL: https://www.isa-afp.org/entries/Regular_Tree_Relations.html.
- 590 22 Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholtz, and Fuyuan Zhang.
591 Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In Prem Devanbu,
592 Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE ’20: 28th ACM Joint*
593 *European Software Engineering Conference and Symposium on the Foundations of Software*
594 *Engineering, Virtual Event, USA, November 8-13, 2020*, pages 701–712. ACM, 2020. doi:
595 10.1145/3368089.3409763.
- 596 23 Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn
597 Song. A symbolic execution framework for JavaScript. In *S&P*, pages 513–528, 2010. doi:
598 10.1109/SP.2010.38.
- 599 24 Gabriel Scherer and contributors. Dolmen: A library and binary for parsing, typechecking,
600 and evaluating languages used in automated deduction. <https://github.com/Gbury/dolmen>,
601 2023. Accessed: 2023-10-15.
- 602 25 Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang.
603 Coqqfbv: A scalable certified SMT quantifier-free bit-vector solver. In Alexandra Silva and
604 K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference,*
605 *CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes*
606 *in Computer Science*, pages 149–171. Springer, 2021. doi:10.1007/978-3-030-81688-9_7.
- 607 26 SMT-LIB. Smt-lib benchmarks. <https://smt-lib.org/benchmarks>. Accessed: 2023-10-17.
- 608 27 Hellis Tamm and Margus Veanes. Theoretical aspects of symbolic automata. In A Min Tjoa,
609 Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jiri Wiedermann, editors, *SOFSEM*
610 *2018: Theory and Practice of Computer Science - 44th International Conference on Current*
611 *Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 - February*
612 *2, 2018, Proceedings*, volume 10706 of *Lecture Notes in Computer Science*, pages 428–441.
613 Springer, 2018. doi:10.1007/978-3-319-73117-9_30.
- 614 28 Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vul-
615 nerability detection in web applications. In Gail-Joon Ahn, Moti Yung, and Ninghui Li,

- 616 editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communica-*
 617 *tions Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1232–1243. ACM, 2014.
 618 doi:10.1145/2660267.2660372.
- 619 29 Thomas Tuerk. A Formalisation of Finite Automata in Isabelle / HOL. <https://www.thomas-tuerk.de/assets/talks/cava.pdf>, 2012.
- 620 30 Stanford University and University of Iowa. cvc5: An efficient open-source automatic theorem
 621 prover for smt problems. <https://cvc5.github.io/>, 2023. Accessed: 2023-10-15.
- 622 31 Margus Veanes, Thomas Ball, Gabriel Ebner, and Ekaterina Zhuchko. Symbolic automata:
 623 Omega-regularity modulo theories. *Proc. ACM Program. Lang.*, 9(POPL):33–66, 2025. doi:
 624 10.1145/3704838.
- 625 32 Margus Veanes and Nikolaj S. Bjørner. Symbolic tree transducers. In Edmund M. Clarke,
 626 Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics -*
 627 *8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June*
 628 *27-July 1, 2011, Revised Selected Papers*, volume 7162 of *Lecture Notes in Computer Science*,
 629 pages 377–393. Springer, 2011. doi:10.1007/978-3-642-29709-0_32.
- 630 33 Margus Veanes, Nikolaj S. Bjørner, and Leonardo Mendonça de Moura. Symbolic automata
 631 constraint solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Pro-*
 632 *gramming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17,*
 633 *Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in*
 634 *Computer Science*, pages 640–654. Springer, 2010. doi:10.1007/978-3-642-16242-8_45.
- 635 34 Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj S. Bjørner.
 636 Symbolic finite state transducers: algorithms and applications. In John Field and Michael
 637 Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of*
 638 *Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*,
 639 pages 137–150. ACM, 2012. doi:10.1145/2103656.2103674.
- 640 35 Jeremy Yallop and contributors. ocaml-re-nfa: Ocaml code to construct an nfa from a regular
 641 expression. <https://github.com/yallop/ocaml-re-nfa>, 2023. Accessed: 2023-10-15.
- 642 36 Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable
 643 runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating*
 644 *Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages
 645 701–718. USENIX Association, 2020. URL: [https://www.usenix.org/conference/osdi20/](https://www.usenix.org/conference/osdi20/presentation/yaseen)
 646 [presentation/yaseen](https://www.usenix.org/conference/osdi20/presentation/yaseen).