Certified Symbolic Finite Transducers: Formalization and Applications to String Manipulation

Anonymous Author(s)

Abstract

Finite Transducers (FTs) extend the capabilities of Finite Automata (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. FTs have wide applications, including program analysis, string constraint solving, and security-critical sanitizer analysis. However, there is currently no scalable formalization of FTs. This is primarily because transition labels in FTs are not formalized symbolically, resulting in transition explosion and significant performance bottlenecks in practical applications.

In this paper, we present a formalization of FTs in the symbolic setting using Isabelle/HOL, where transition labels can be instantiated with various Boolean algebras, such as intervals and arithmetic predicates. This symbolic approach enables not only efficient FT operations—such as computing the output language of an FT given a regular language—but also supports infinite alphabets. To evaluate the effectiveness of our formalization, we apply the formalized SFTs to two applications: (1) `HTMLdecode`, a sanitizer used for preventing XSS attacks, and (2) a string solver that models replacement operations with various semantics, including left-most and replace-all matching. Experimental results demonstrate that our approach achieves computational efficiency in constraint-solving scenarios.

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 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, 33] 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 ($'\mathtt{a}' - '\mathtt{z}'$), encompassing all characters from $'\mathtt{a}'$ to $'\mathtt{z}'$, or as an arithmetic condition ($x \mod 2 = 0$), 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

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [33] 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, [34] 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 examplify 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 [25]. 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 accommodating 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 **??** presents the product operation at the abstract level. Section 3.2 details the algorithmic refinement of the product operation. Section **??** demonstrates the application to string constraint solving. Section 5 discusses related work. Section 6 concludes with future directions.

2 Symbolic Finite Transducers

We begin by presenting a mathematical definition of SFTs [33], abstracting from the specific implementation details of Isabelle/HOL.

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 $\mathcal{U}^{\tau}$ denotes the sub-universe of elements of type τ. We have a special type $\mathbb{B}$ with $\mathcal{U}^{\mathbb{B}} = \{\top, \bot\}$, which corresponds to the boolean type.

A lambda term is defined as $\lambda x.\ t$ of type $\tau_1 \longrightarrow \tau_2$. When τ_2 is $\mathbb{B}$, this lambda term is a predicate. Let ϕ be a predicate. We write $a \in [\![\phi]\!]$ 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 2.1 (Symbolic Finite Transducer). A Symbolic Finite Transducer over $\tau_1 \longrightarrow \tau_2$ is a quadruple $\mathcal{T} = (Q, \Delta, \mathcal{I}, \mathcal{F})$, where

- Q is a finite set of states,
- $\mathcal{I} \subseteq Q$ is the set of initial states,
- $\mathcal{F} \subseteq Q$ 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 Q. ϕ is a predicate of type $\tau_1 \longrightarrow \mathbb{B}$. f is a lambda term of type $\tau_1 \longrightarrow \tau_2$. f is called an *output function.*

For each transition $q \xrightarrow{\phi,f} q'$, if there exists an element $a \in [\![\phi]\!]$, 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 $(q_0, \phi_0, f_0, q_1), (q_1, \phi_1, f_1, q_2), \ldots, (q_{n-1}, \phi_{n-1}, f_{n-1}, q_n)$

such that $q_0 \in \mathcal{I}$ and $(q_i, \phi_i, f_i, q_{i+1}), 0 \le i \le n-1$ 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 $a_i \in [\![\phi_i]\!]$ for $0 \le i \le n-1$. 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,

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

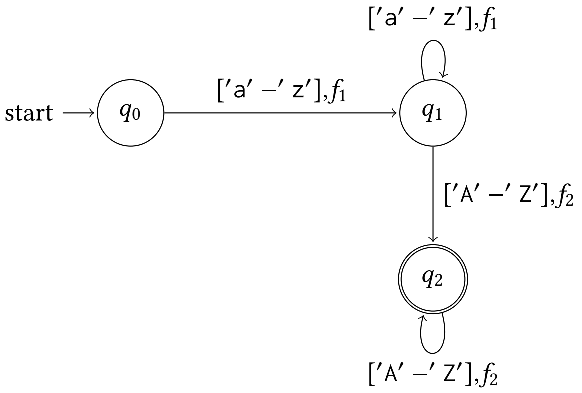


Figure 1. An example of SFT

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

3 The Isabelle/HOL Formalization of SFTs

We present the basic architecture of our SFT formalization in Isabelle/HOL, which consists of three distinct layers.

The **abstract layer** defines transition labels as **sets**, states are stored in sets as well. This abstract formalization, which abstracts away implementation details, enables straightforward correctness proofs for SFT operations.

The **implementation layer** refines the abstract transition labels to a locale where a boolean algebra is defined as type variable `'b` together with a collection of operations and assumptions. State sets are refined using Refine_Monadic [19], a framework in Isabelle/HOL for data refinement that automatically transforms general sets into efficient data structures such as hashmaps and red-black trees.

The **boolean algebra layer** refines the transition label locale to a concrete boolean algebra implementation. In this paper, we present an interval-based refinement.

This three-layer architecture separates the fixed components (abstract and implementation layers) of the SFT formalization from the variable components (transition labels). This separation enables us to extend the SFT formalization to accommodate new transition label theories without modifying the first two layer formalization.

3.1 The Abstract Layer of SFTs

We present the SFT formalization in abstract layer in Fig. 2. 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 2.1, 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 $'q \times ('a$ `set option` $\times' i) \times' q$. This representation reflects several key design decisions aimed at enhancing the abstraction and flexibility of our SFT formalization.

```
record ('q, 'a, 'i, 'b) NFT =
        Q_t  ::  "'q set"
        Δ_t  ::  "('q, 'a, 'i) LTTS"
        I_t  ::  "'q set"
        F_t  ::  "'q set"
        M_t  ::  "'i ⇒ ('a, 'b) Tlabel"
type_synonym ('q, 'a, 'i) LTTS  =  "
   ('q × ('a set option  ×' i) ×' q)  set"
type_synonym ('a, 'b) Tlabel  =  "
   'a option  ⇒  'b set option"
```

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

Firstly, $'a$ set option is the input type of the transition, it accepts a set of elements of type $'a$ or None corresponding to empty string ε. Accepting a set of $'a$ elements aims to express the same but more abstract semantics of the input labels in Definition 2.1, in which an input label is a predicate. A predicate's semantics as introduced before represents a set of elements that make the predicate true. But predicates have various different forms. For instance, the interval $[1-9]$ represents the set $\{e \mid 1 \leq e \leq 9\}$. The predicate $\lambda b.\ b[7] = 1$, where b is a bit vector of length 8, denotes the set of bit vectors that have a 1 in the 7th position. All these different forms of labels are abstracted as sets in our formalization. The value None represents the empty string ε in our formalization, indicating a transition that consumes no input but may still produce output elements. This design choice facilitates the modeling of real-world applications in SFTs, as we will demonstrate in our application to string solvers.

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

```
record ('q, 'a) NFA =
          𝒬 :: "'q set"
          Δ :: "('q, 'a) LTS"
          ℐ :: "'q set"
          ℱ :: "'q set"

record ('q, 'a) eNFA =
          𝒬ₑ :: "'q set"
          Δₑ :: "('q, 'a) LTS"
          Δ'ₑ :: "('q *' q) set"
          ℐₑ :: "'q set"
          ℱₑ :: "'q set"

type_synonym ('q,' a) LTS = "'q ×' a set ×' q"
```

Figure 3. The formalization of εSFAs and SFAs

```
definition productT :: "('q,' a,' i,' b)" NFT ⇒
    ('q,' a) NFA ⇒(('a,' b) Tlabel ⇒' a set
    ⇒' b set option) ⇒ ('q ×' q, 'b) eNFA where
"productT 𝒯 𝒜 F = ⦇
  𝒬e = 𝒬ₜ 𝒯 × 𝒬 𝒜,
  Δₑ = {((p, p'), the (((ℳₜ 𝒯) f) None), (q, p')) |
          p, p', q, f. p' ∈ 𝒬 𝒜 ∧ (p, (None, f), q) ∈ Δₜ 𝒯
           ∧ ∃S. (ℳₜ 𝒯) f None = Some S} ∪
         {((p, p'), the (F ((ℳₜ 𝒯) f) (σ₁∩σ₂)), (q, q')) |
          p, p', q, σ₁, σ₂, q', f. (p, (Some σ₁, f), q) ∈ Δₜ 𝒯 ∧
          (p', σ₂, q') ∈ Δ 𝒜 ∧ σ₁ ∩ σ₂ ≠ ∅ ∧
                ∃S. F ((ℳₜ 𝒯) f) (σ₁ ∩ σ₂) = Some S},
  Δ'ₑ = {((p, p'), (q, p')) | p, p', q, f. p' ∈ 𝒬 𝒜∧
          (p, (None, f), q) ∈ Δₜ 𝒯 ∧
          (ℳₜ 𝒯) f None = None} ∪
         {((p, p'), (q, q')) | p, p', q, σ₁, σ₂, q', f.
           (p, (Some σ₁, f), q) ∈ Δₜ 𝒯 ∧ (p', σ₂, q') ∈ Δ 𝒜
           ∧ σ₁ ∩ σ₂ ≠ ∅ ∧ ∃x ∈ (σ₁ ∩ σ₂).
            ((ℳₜ 𝒯) f) (Some x) = None},
  ℐₑ = ℐₜ 𝒯 × ℐ 𝒜
  ℱₑ = ℱₜ 𝒯 × ℱ 𝒜 ⦈"
```

Figure 4. The formalization of product operation

The product operation formalization. We formalize the product operation between an SFT $\mathcal{T}$ and an SFA $\mathcal{A}$, denoted as $\mathcal{T} \times \mathcal{A}$ in Section 2. The SFA is formalized as a record type `NFA in Figure 3`. An additional consideration in this operation is the presence of ε-transitions in our SFT formalization, which implies that the resulting automata may also contain ε-transitions. Therefore, we need to formalize εSFAs, which is shown in Fig. 3 as eNFA. The εSFA formalization extends the standard SFA structure by introducing Δ'_e, which captures ε-transitions as pairs of states, while maintaining the same labeled transition relation Δ as in standard SFAs. In this paper, we present only the definitions of εSFAs and SFAs, while the complete formalization, including correctness proofs, is available in our Isabelle development.

Having established these foundational definitions, we can now formalize the 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 εSFA. But we need to explain the role of parameter F. The output function f for each transition in Δ_t is of type "$'a$ option $\Rightarrow' b$ 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:

$$\mathsf{F}\ f\ A = \mathsf{Some}\ \Big(\bigcup_{a \in A} (\mathsf{if}\ f\ a = \mathsf{Some}\ S\ \mathsf{then}\ S\ \mathsf{else}\ \emptyset)\Big)$$

.

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

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

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

To establish the correctness specification of the product operation, we begin by formalizing the concept of SFT *traces* as introduced in Definition 2.1. In our formalization, traces are represented by the type $('a\ \mathrm{option} \times' b\ \mathrm{option})$ `list`, as shown in Figure 5. Given a trace π, we define two key projection functions: (1) `inputE`, which corresponds to $in(\pi)$ and extracts the input sequence (2) `outputE`, which corresponds to $out(\pi)$ and extracts the output sequence.

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

```
fun inputE ::
   "('a option × 'b option) list ⇒ 'a list"
where
  "inputE [] = []" |
  "inputE ((Some a, _) # l) =
                           a # (inputE l)" |
  "inputE ((None , _) # l) = (inputE l)"

fun outputE ::
    "('a option×' b option) list ⇒' b list"
where
  "outputE [] = []" |
  "outputE ((_,Some a) # l) =
                         a # (outputE l)" |
  "outputE ((_,None) # l) = (outputE l)"

definition outputL ::
   "('q,' a,' i,' b) NFT ⇒ ('q,' a) NFA ⇒' b list set"
 where
  "outputL 𝒯 𝒜 = {outputE π | π q q'.
    q ∈ ℐt 𝒯 ∧ q' ∈ ℱt 𝒯 ∧
    LTTS_reachable 𝒯 q π q'∧inputE π ∈ ℒ 𝒜}"
```

Figure 5. The formalization of traces in SFTs

```
lemma productT_correct:
  fixes 𝒯 𝒜 F
  assumes
   F_ok1: "∀ f s. (∀e ∈ s. f (Some e) = None) ⟷
         F f s = None"
   and F_ok2: "∀ f s. F f s ≠ None ⟶ F f s =
         Some (∪ {S | e S. e ∈ s ∧
         f (Some e) = Some S})"
      and wfTA: "NFT_wf 𝒯 ∧ NFA 𝒜"
 shows "ℒe (productT 𝒯 𝒜 F) = outputL
         𝒯 𝒜"
```

Figure 6. The correctness lemma of the product operation

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

3.2 Implementation Layer Refinement

Having established the abstract definition of the SFT product operation in Section 3.1, we now present its algorithmic refinement. This section introduces an efficient implementation of the product construction and refines the abstract representation of transition labels to a concrete interval algebra, enabling practical computation.

Boolean Algebra Locale. Figure 7 presents the simplified definition of a boolean algebra locale. The type variable 'b represents any boolean algebra, while the type variable 'a denotes the element type of the sets that the boolean algebra represents. The locale defines a collection of operations and assumptions that characterize boolean algebra behavior, including operations for set semantics, emptiness checks, non-emptiness checks, intersection, difference, and element membership. The function "sem" provides the semantic interpretation of a boolean algebra as a set, with all other operations' assumptions defined in terms of this set semantics.

The locale serves as an interface to boolean algebra implementations. Any concrete boolean algebra implementation must provide a concrete instantiation of this locale, which includes defining the type variable 'b and providing implementations for all operations along with proofs of the required assumptions.

```
locale bool_algebra =
 fixes sem :: " 'b ⇒ 'a::ord set"
 fixes empty:: "'b ⇒ bool"
 fixes nempty:: "'b ⇒ bool"
 fixes intersect:: "'b ⇒ 'b ⇒ 'b"
 fixes diff:: "'b ⇒ 'b ⇒ 'b"
 fixes elem:: "'a  ⇒ 'b ⇒ bool"
 assumes
  empty_sem: "empty s = (sem s = {})"
                                     and
  nempty_sem: "nempty_intervals s =
                     (¬ (empty s))" and
  inter_sem: "sem (intersect s1 s2) =
                (sem s1) ∩ (sem s2)" and
  diff_sem: "sem (diff f1 f2 s1 s2) =
                (sem s1) - (sem s2)" and
  elem_sem: "elem a s ≡ (a ∈ sem s)"
```

Figure 7. The boolean algebra locale

Implementation layer of the product operation. We now present the algorithmic implementation of the product operation between an SFT and an SFA[1] based on Refined_monadic framework.

Figure 8 illustrates the core algorithm `productT_impl`, which is implemented using the *Refine_Monadic* framework. Note that in the implementation, there are some refined operations: `nft_tranfun`, `nfa_states`, `nfa_trans`, `nfa_initial`, and `nfa_accepting`. 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 9, 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.

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 8. As shown in Figure 10, 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

```
definition productT_impl where
"productT_impl 𝒯 𝒜 F fe = do {
    Q ← prods_imp (nft_states 𝒯)
    (nfa_states 𝒜);
    (D1, D2) ← trans_comp_imp
     (nft_tranfun 𝒯) F fe (nft_trans 𝒯)
      (nfa_trans 𝒜) (nfa_states 𝒜);
    I ← prods_imp (nft_initial 𝒯)
      (nfa_initial 𝒜);
    F ← prods_imp (nft_accepting 𝒯)
      (nfa_accepting 𝒜);
    RETURN (Q, D1, D2, I, F)
  }"
```

Figure 8. The computation of SFT product

```
definition prods_imp where
"prods_imp Q1 Q2 =
   FOREACH {q. q ∈ Q1} (λ q Q. do {
    S ← FOREACH {q. q ∈ Q2}
      (λ q' Q'. RETURN ({(q,q')} ∪ Q')) ∅;
    RETURN (Q ∪ S)
   }) ∅"
```

Figure 9. The computation of Cartesian product of two state sets

```
definition trans_comp_imp where
"trans_comp_imp M F fe T1 T2 Q =
  FOREACH {t. t ∈ T1}
   (λ(q,(α,f),q') (D1,D2).
    (if (α=None) then
     (subtrans_comp_ε M q f q' F fe
                           T2 D1 D2)
     else
     (subtrans_comp M q (the α) f
            q' F fe T2 D1 D2))) (∅, ∅)"
```

Figure 10. The computation of `trans_comp_imp`

2. `subtrans_comp`: Processes standard transitions in the SFT, where both input consumption and output generation may occur

We now present the implementation of `subtrans_comp` in detail, as shown in Figure 11 (the implementation of `subtrans_comp_`ε follows analogous principles). For a transtion $(q, (\text{Some } \alpha, f), q')$ in the SFT, this function traverses all transitions in the SFA, represented by the set `T`. For each transition $(q_1, \alpha', q_1') \in$ `T`, the function performs two key operations when the intersection of input labels is non-empty ($\alpha \cap \alpha' \neq \emptyset$, verified using `nemptyIs`):

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

```
definition subtrans_comp where
"subtrans_comp M q α f q' F fe T D1 D2 =
 FOREACH {t.t∈T} (λ (q1,α',q1') (D1,D2).
  (if (nempty (intersect α α')) then
    do {
      D1 ←
       (if (F (M f) (intersect α α'))
          ≠ None) then
         let α_i = the (F (M f)
              (intersect α α')) in
         RETURN {((q,q1), α_i, (q',q1'))}
                ∪ D1
        else RETURN D1);
      D2 ←
       (if fe (M f) (intersect α α')
        then
         RETURN {((q,q1), (q',q1'))} ∪ D2
        else RETURN D2);
      RETURN (D1, D2) }
   else (RETURN (D1, D2))) (D1, D2)"
```

Figure 11. The computation of `subtrans_comp`

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

```
  lemma productT_imp_correct:
  assumes finite_TT: "finite (Δ_t 𝒯)"
      and finite_TA: "finite (Δ 𝒜)"
      and finite_Q: "finite (𝒬 𝒜)"
      and finite_TQ: "finite (𝒬_t 𝒯)"
      and finite_I: "finite (ℐ 𝒜)"
      and finite_TI: "finite (ℐ_t 𝒯)"
      and finite_F: "finite (ℱ 𝒜)"
      and finite_TF: "finite (ℱ_t 𝒯)"
  shows "productT_imp 𝒯 𝒜 F fe ≤
          SPEC (λA. A = productT 𝒯 𝒜 F)"
```

Figure 12. The refinement relation between `productT_imp` and `productT`

The correctness of the product computation is established through a refinement proof, demonstrating that `productT_imp` (Figure 8) correctly implements the abstract specification `productT` (Figure 4). This refinement relationship is formally specified in Figure 12, where we leverage the *Refine_Monadic* framework's data refinement.

The refinement is expressed through the relation **C** ≤ SPEC **A**, which asserts that the concrete implementation **C** is an element of the abstract specification **A**. More precisely, the concrete implementation must produce an εSFA that is structurally equivalent or isomorphic to the one produced by the abstract algorithm `productT`.

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

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

3.3 Interval Implementation

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

Throughout the following discussion, we use the term "interval" to refer to interval lists. Our formalization provides all operations in the boolean algebra locale. For instance, we implement `intersect` for intervals, which computes the intersection of two intervals i_1 and i_2, yielding an interval i such that $\mathsf{sem}\ i = \mathsf{sem}\ i_1 \cap \mathsf{sem}\ i_2$

To facilitate formal reasoning and optimize performance, in addition to the above locale operations, we introduce a canonical form for interval. An interval $[(i_1, j_1), \ldots, (i_n, j_n)]$ is in canonical form if it satisfies two key properties:

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

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

4 Applications and Evaluation

We evaluate our formalization and implementation of the SFT product operation in two key application domains: string sanitization and string replacement operations in string solving. These applications demonstrate the practical utility of our certified SFT product operation in real-world scenarios.

4.1 Sanitizer

An important application of our formalization lies in string sanitization for web applications. Sanitizers are crucial for preventing security vulnerabilities such as cross-site scripting (XSS) attacks by ensuring that user inputs are properly encoded before rendering in web browsers. Listing 1 presents a C# code snippet implementing an HTML encoding function—a fundamental sanitization operation. This function iterates through each character in the input string and encodes characters that are unsafe for HTML rendering.

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

Listing 1. C# Code for AntiXSS.EncodeHtml version 2.0.

Figure 13 illustrates the SFT representation of the HTML encoding sanitizer function. The transducer accepts any input string and produces an output where safe characters are preserved using the identity function id, while unsafe characters are encoded as HTML entities. The set of safe characters is defined as safe_chars = [' ',',',',','.','-','_'] ∪ ['0' − '9'] ∪ ['A' − 'Z'] ∪ ['a' − 'z'], which can be efficiently represented using the interval algebra we formalized. Unsafe characters are transformed using the encoding function encode(c) = “&#” + toString(ord(c)) + “ ”. When the alphabet is set to Unicode [0x0000, 0x10FFFF], the complement ¬safe_chars represents all Unicode characters outside the safe set, which can also be efficiently represented using our formalized interval operations.

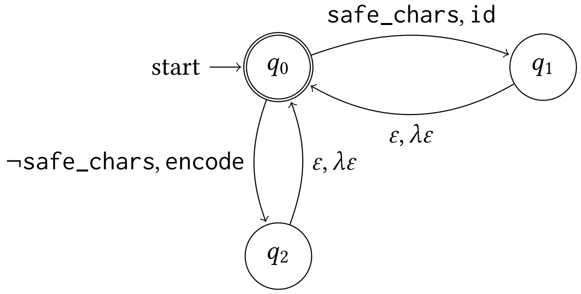

safe_chars, id
start
q_0
q_1
$\varepsilon, \lambda\varepsilon$
¬safe_chars, encode
$\varepsilon, \lambda\varepsilon$
q_2

Figure 13. SFT representation of the HTML encoding sanitizer function.

We evaluate the performance of our SFT product operation by computing the product of the HTML encoding sanitizer SFT with SFAs constructed from randomly generated Unicode user inputs. This experimental setup reflects realistic usage scenarios where diverse character sets and input patterns are encountered.

Figure 14 demonstrates the scalability characteristics of our SFT product implementation. The computation requires approximately 1 second for 100,000-character inputs and 10 seconds for 1,000,000-character inputs. These performance metrics confirm that our certified implementation maintains computational efficiency suitable for practical string processing applications at scale.

Beyond performance evaluation, our SFT formalization enables formal verification of string sanitizer correctness.

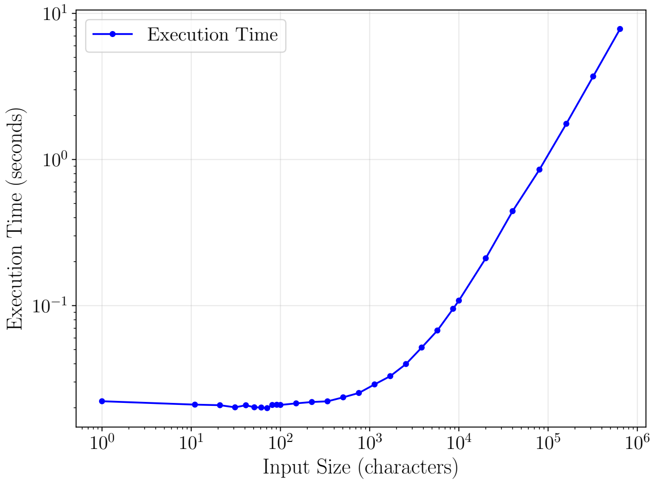


Figure 14. Experimental results for the sanitizer product

For example, this verification framework models user input as an SFA, the sanitizer as an SFT, and potential attacks as an SFA accepting unsafe strings. A sanitizer is verified correct if the product of the sanitizer SFT and user input SFA produces no outputs that intersect with the attack SFA's language. This correctness property can be checked by computing the intersection of the product SFT's output language with the attack SFA, where emptiness of this intersection guarantees sanitizer safety.

4.2 Modeling the Replacement Operation

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

- `str`: The input string to be transformed.
- `pattern`: A regular expression defining the matching criteria.
- `replacement`: The string to be substituted for the matched substring.

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 `replace` (s, `/[0-9]+/`, ”`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 $f = \lambda x.$ 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 `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 15 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` $= \lambda x.$ `None`, which consistently produces the empty string, effectively "consuming" the matched digits without generating output.

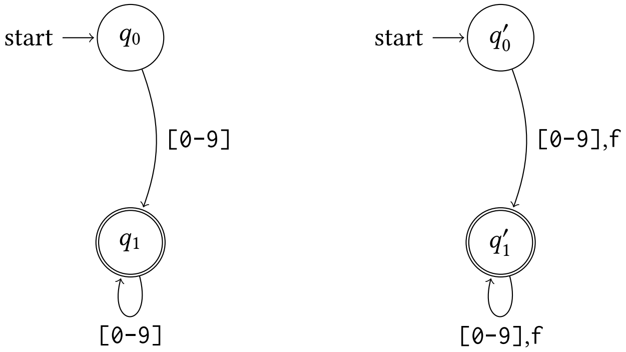


Figure 15. Corresponding SFA and SFT for `/[0-9]+/`

Step 2. Figure 16 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:

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

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

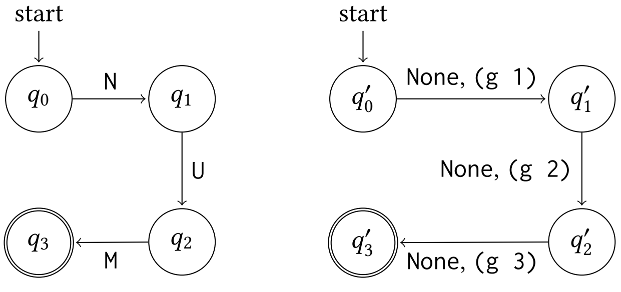


Figure 16. Corresponding SFA and SFT for "`NUM`"

Step 3. Figure 17 presents the complete SFT construction, obtained by composing the pattern-matching SFT (Fig. 15) with the replacement-generating SFT (Fig. 16). The composition process involves two key modifications:

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

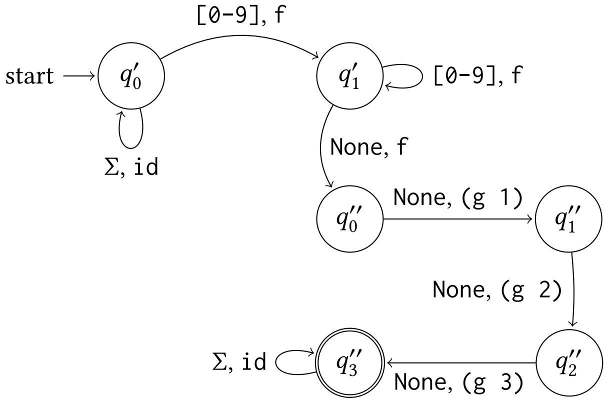


Figure 17. The SFT for the replacement operation `replace(`s`, /[0-9]+/, `“NUM”`)`

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

Comparison with SMT-LIB Semantics. It is important to note that our formalization differs from the standard SMT-LIB semantics for the replacement operation. In SMT-LIB, the `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.

The replacement operation model described above replaces a single occurrence of any substring matching the regular expression pattern. To support different replacement semantics, we may also need to model left-most replacement (which replaces the first occurrence) and replace-all operations (which substitute all occurrences of matching substrings).

The approach to modeling left-most replacement is illustrated in Figure 18. This construction involves two SFTs: one for non-matching characters and one for the matching pattern. The non-matching SFT is constructed by first computing the negation of the pattern (the complement of the regular expression's NFA) and converting it into an SFT by adding the identity function $\lambda x.x$. The matching SFT is constructed similarly to the previous example in Figure 17, but without self-loops at the initial state, creating an SFT that processes the pattern and generates the replacement string.

The two SFTs are composed by adding an ε-transition from the accepting state of the non-matching SFT to the initial state of the matching SFT. This transition enables the left-most replacement operation to proceed to pattern matching after processing any non-matching prefix.

The approach to modeling replace-all operations is illustrated in Figure 19. This construction is similar to left-most replacement but includes an additional ε-transition from the accepting state of the matching SFT back to the initial state of the non-matching SFT. This bidirectional connection allows the replace-all operation to continue processing the input string and replace all occurrences of matching substrings.

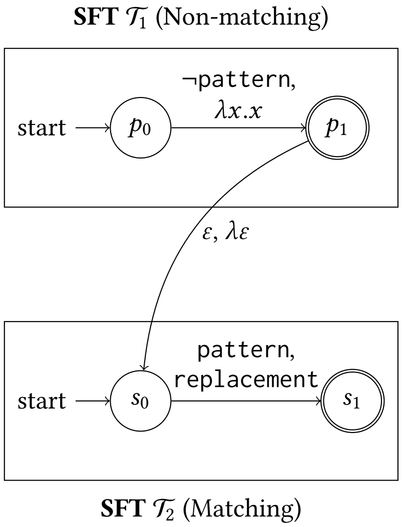
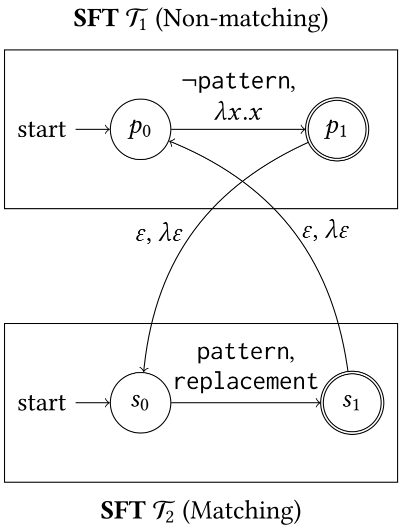

SFT $\mathcal{T}_1$ (Non-matching)
start
p_0
¬pattern,
$\lambda x.x$
p_1
ε, $\lambda\varepsilon$
start
s_0
pattern,
replacement
s_1
SFT $\mathcal{T}_2$ (Matching)

Figure 18. The modeling of left-most replacement operation.

4.3 Experiments of String Replacement Operations

We have implemented CertiStrR, an extension of CertiStr [17], to support string replacement operations. CertiStr [17] is a certified string solver that only support some basic string operations, such as concatenation, and regular constraints.


SFT $\mathcal{T}_1$ (Non-matching)

SFT $\mathcal{T}_2$ (Matching)

Figure 19. The modeling of replace all operation.

CertiStrR implements three different pattern matching introduced above: (1) singleton matching at any position, (2) left-most matching, and (3) replace-all matching. The pattern can be a regular expression or a constant string, which is a special case of regular expression.

Listing 2 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". But the left-matching will return UNSAT.

```
(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 2. Example SMT-LIB Code

We evaluate CertiStrR using benchmarks from SMT-LIB 2024 [25], 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/Inc	Time (s)	Tests
replace singleton	142/173/8	0.27	323
replace left-match	84/4/10	0.36	98
replace all	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:

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

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: $x_1{+}{+}x_2$, replace (x_3, p, r), x_4, and 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.

4.4 Effort of Certified Development

This subsection discusses the development effort required for our certified SFT formalization. Table 2 provides an overview of this effort. All efforts listed in the table are additional to the existing CertiStr framework that we build upon.

The abstract-level development encompasses all formalizations presented in Sections 2 and ??, including SFTs and εSFAs. The implementation-level development covers all formalizations in Section 3.2, including the refinement of the product operation. The final row corresponds to the interval formalization effort. The most challenging component proved to be the correctness proof of the product operation at the abstract level (Figure 6).

The interval formalization complexity increased dramatically in CertiStrR compared to the interval formalization in CertiStr [17], primarily due to extending intervals from single pairs to lists.

	Defs	Lemmas	Proofs (LOC)
Abstract	17	21	3274
Implementation	51	43	2700
Interval	15	29	1500

Table 2. Overview of the effort of certified development

5 Related Work

Symbolic Automata and Transducers. Symbolic Automata and Transducers [10–12, 26, 33] 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 [31] 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 [30], 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 [32], security-critical sanitizer analysis for web applications [15], runtime verification of system behaviors [34], 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 [7, 13, 18, 28], 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 εSFAs.

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 [29], Z3-str3 [5] builds on the Z3 framework, S3P [27], Ostrich [8], and SLOTH [14]. As more and more bugs have beein 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. [24], 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.

6 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 [25]. 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 Søe Luckow, 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.

[14] Lukás Holík, Petr Janku, Anthony W. Lin, Philipp Rümmer, and Tomás Vojnar. String constraints with concatenation and transducers solved efficiently. *Proc. ACM Program. Lang.*, 2(POPL):4:1–4:32, 2018. doi:10.1145/3158092.

[15] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. URL: http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf.

[16] Qinheping Hu and Loris D'Antoni. Automatic program inversion using symbolic transducers. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 376–389. ACM, 2017. doi:10.1145/3062341.3062345.

[17] Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. Certistr: a certified string solver. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 210–224. ACM, 2022. doi:10.1145/3497775.3503691.

[18] P. Lammich. The CAVA automata library. *Arch. Formal Proofs*, 2014, 2014.

[19] Peter Lammich. Refinement for monadic programs. *Archive of Formal Proofs*, January 2012. https://isa-afp.org/entries/Refine_Monadic.html, Formal proof development.

[20] Peter Lammich. Automatic Data Refinement. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013. doi:10.1007/978-3-642-39634-2_9.

[21] Alexander Lochmann, Bertram Felgenhauer, Christian Sternagel, René Thiemann, and Thomas Sternagel. Regular tree relations. *Arch. Formal Proofs*, 2021, 2021. URL: https://www.isa-afp.org/entries/Regular_Tree_Relations.html.

[22] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 701–712. ACM, 2020. doi:10.1145/3368089.3409763.

[23] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *S&P*, pages 513–528, 2010. doi:10.1109/SP.2010.38.

[24] Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Coqqfbv: A scalable certified SMT quantifier-free bit-vector solver. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, volume 12760 of *Lecture Notes in Computer Science*, pages 149–171. Springer, 2021. doi:10.1007/978-3-030-81688-9_7.

[25] SMT-LIB. Smt-lib benchmarks. https://smt-lib.org/benchmarks. Accessed: 2023-10-17.

[26] Hellis Tamm and Margus Veanes. Theoretical aspects of symbolic automata. In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jirí Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 - February 2, 2018, Proceedings*, volume 10706 of *Lecture Notes in Computer Science*, pages 428–441. Springer, 2018. doi:10.1007/978-3-319-73117-9_30.

[27] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1232–1243. ACM, 2014. doi:10.1145/2660267.2660372.

[28] Thomas Tuerk. A Formalisation of Finite Automata in Isabelle / HOL. https://www.thomas-tuerk.de/assets/talks/cava.pdf, 2012.

[29] Stanford University and University of Iowa. cvc5: An efficient open-source automatic theorem prover for smt problems. https://cvc5.github.io/, 2023. Accessed: 2023-10-15.

[30] Margus Veanes, Thomas Ball, Gabriel Ebner, and Ekaterina Zhuchko. Symbolic automata: Omega-regularity modulo theories. *Proc. ACM Program. Lang.*, 9(POPL):33–66, 2025. doi:10.1145/3704838.

[31] Margus Veanes and Nikolaj S. Bjørner. Symbolic tree transducers. In Edmund M. Clarke, Irina B. Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers*, volume 7162 of *Lecture Notes in Computer Science*, pages 377–393. Springer, 2011. doi:10.1007/978-3-642-29709-0_32.

[32] Margus Veanes, Nikolaj S. Bjørner, and Leonardo Mendonça de Moura. Symbolic automata constraint solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 640–654. Springer, 2010. doi:10.1007/978-3-642-16242-8_45.

[33] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj S. Bjørner. Symbolic finite state transducers: algorithms and applications. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 137–150. ACM, 2012. doi:10.1145/2103656.2103674.

[34] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 701–718. USENIX Association, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/yaseen.