CertiStr: A Certified String Solver

Shuanglong Kan  
Department of Computer Science  
Technische Universität Kaiserslautern  
Kaiserslautern, Germany  
shuanglong@cs.uni-kl.de

Philipp Rümmer  
Department of Information Technology  
Uppsala University  
Uppsala, Sweden  
philipp.ruemmer@it.uu.se

Anthony Widjaja Lin  
Department of Computer Science  
Technische Universität Kaiserslautern & MPI-SWS  
Kaiserslautern, Germany  
lin@cs.uni-kl.de

Micha Schrader  
Department of Computer Science  
Technische Universität Kaiserslautern  
Kaiserslautern, Germany  
schrader@rhrk.uni-kl.de

Abstract

Theories over strings are among the most heavily researched logical theories in the SMT community in the past decade, owing to the error-prone nature of string manipulations, which often leads to security vulnerabilities (e.g. cross-site scripting and code injection). The majority of the existing decision procedures and solvers for these theories are themselves intricate; they are complicated algorithmically, and also have to deal with a very rich vocabulary of operations. This has led to a plethora of bugs in implementation, which have for instance been discovered through fuzzing.

In this paper, we present CertiStr, a certified implementation of a string constraint solver for the theory of strings with concatenation and regular constraints. CertiStr aims to solve string constraints using a forward-propagation algorithm based on symbolic representations of regular constraints as symbolic automata, which returns three results: sat, unsat, and unknown, and is guaranteed to terminate for the string constraints whose concatenation dependencies are acyclic. The implementation has been developed and proven correct in Isabelle/HOL, through which an effective solver in OCaml was generated. We demonstrate the effectiveness and efficiency of CertiStr against the standard Kaluza benchmark, in which 80.4% tests are in the string constraint fragment of CertiStr. Of these 80.4% tests, CertiStr can solve 83.5% (i.e. CertiStr returns sat or unsat) within 60s.

CCS Concepts: • Theory of computation → Automated reasoning: Formal languages and automata theory.

1 Introduction

Strings are among the most fundamental and commonly used data types in virtually all modern programming languages, especially with the rapidly growing popularity of dynamic languages, including JavaScript and Python. Programs written in such languages often implement security-critical infrastructure, for instance web applications, and they tend to process data and code in string representation by applying built-in string-manipulating functions; for instance, to split, concatenate, encode/decode, match, or replace parts of a string. Functions of this kind are complex to reason about and can easily lead to programming mistakes. In some cases, such mistakes can have serious consequences, e.g., in the case of web applications, cross-site scripting (XSS) attacks can be used by a malicious user to attack both the web server or the browsers of other users.

One promising research direction, which has been intensively pursued in the SMT community in the past ten years, is the development of SMT solvers for theories of strings (dubbed string solvers) including Kaluza [35], CVC4 [28], Z3 [17], Z3-str3 [4], Z3-Trau [9], S3P [39], OSTRICH [13], SLOTH [22], and Norm [1], to name only a few. Such solvers are highly optimized and find application, among others, in bounded model checkers and symbolic execution tools [14, 34], but also in tools tailored to verification of the security properties [3].

It has long been observed that constraint solvers are extremely complicated procedures, and that implementations...
are prone to bugs. Defects that affect soundness or completeness are routinely found even in well-maintained state-of-the-art tools, both in real-world applications and through techniques like fuzzing [6, 8, 31]. String solvers are particularly troublesome in this context, since, unlike other SMT theories, the theory of strings requires a multitude of operations including concatenation, regular constraints, string replacement, length constraints, and many others. This inherent intricacy is also reflected in the recently developed SMT-LIB standard for strings (http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml). Many of the string solvers that arose in the past decade relied on inevitably intricate decision procedures, which resulted in subtle bugs in implementations themselves even among the mainstream string solvers (e.g., see [8, 31]). This implies the unfortunate fact that one cannot blindly trust the answer provided by string solvers.

**Contributions.** In this paper, we present CertiStr, the first certified implementation of a string solver for the standard theory of strings with concatenation and regular constraints (a.k.a. word equations with regular constraints [18, 20, 23, 30]). CertiStr aims to solve string constraints by means of the so-called forward-propagation algorithm, which relies on a simple idea of propagating regular constraints in a forward direction in order to derive contradiction, or prove the absence thereof (see Section 2 for an example). Similar ideas are already present in abstract interpretation of string-manipulating programs (e.g., see [32, 42, 43]), but not yet at the level of string solvers, i.e., which operate on string constraints. We have proven in Isabelle/HOL [33] some crucial properties of the forward-propagation algorithm and more specifically with respect to our implementation CertiStr: (i) [TERMINATION]: the algorithm terminates on string constraints without cyclic concatenation dependencies, (ii) [SOUNDNESS]: CertiStr is sound for unsatisfiable results (if forward-propagation detects inconsistencies in a constraint, then the constraint is indeed unsatisfiable), and (iii) [COMPLETEENESS]: CertiStr is complete for constraints satisfying the tree property (i.e., in which on the right-hand side of each equation every variable appears at most once). Here completeness amounts to the fact that, if forward-propagation does not detect inconsistencies for a constraint satisfying the tree property, then the constraint is indeed satisfiable. For the constraints that do not satisfy the tree property and CertiStr cannot decide them as unsatisfiable, CertiStr returns unknown.

In order to facilitate the propagation of regular constraints, we also implement a certified library of Symbolic Non-deterministic Finite Automata (s-NFA) by allowing transition labels to be a set of characters in the (potentially infinite) alphabet, represented by an element of a boolean algebra (e.g. the interval algebra or the BDD algebra), instead of a single character. s-NFAs are especially crucial for an efficient implementation of automata-based string solving algorithms and many string processing algorithms [1, 13, 16, 24, 41], for which reason our certified implementation of an s-NFA library is of independent interests.

Last but not least, we have automatically generated a verified implementation CertiStr in Isabelle/HOL, which we have extensively evaluated against the standard string solving benchmark from Kaluza [35] with around 38000 string constraints. For the first time, we demonstrate that the simple forward-propagation algorithm in fact performs surprisingly well even compared to other highly optimized solvers (which are not verified implementations). In particular, we show that the majority of these constraints (83.5%) are solved by CertiStr; for the rest, the tool either returns unknown, or times out. Moreover, CertiStr terminates within 60 seconds on 98% of the constraints, witnessing its efficiency.

To summarize, our contributions are:

- we developed in Isabelle/HOL the tool CertiStr, the first certified implementation of a string solver.
- we implemented the first certified symbolic automata library, which is crucial for an efficient implementation of many string processing applications [16].
- CertiStr was evaluated over Kaluza benchmark [35], with around 38000 tests. In this benchmark, 83.5% of the tests can be solved with the results sat or unsat. Moreover, the solver terminates within 60 seconds on 98% of the tests, witnessing the surprising competitiveness of the simple forward-propagation algorithm against more complicated algorithms.

## 2 Motivating Example

We start by illustrating the decision procedure implemented by CertiStr. CertiStr uses the so-called forward-propagation algorithm for solving satisfiability of string theory with concatenation and regular membership constraints. To illustrate the algorithm, consider the formula:

\[
\text{domain} \in ([a-zA-Z]+) \\
\land \ dir. file \in ([a-zA-Z0-9.]+) \\
\land \ path = \text{dir} + \"\" + \text{file} \\
\land \ url = \"http://\" + \text{domain} + \"\" + \text{path} \\
[ \land \ url \in /.*<script>.*]/
\]

The formula contains string variables (domain, dir, etc.), and uses both regular membership constraints and word equations with concatenation to model the construction of a URL from individual components. Regular expressions are written using the standard PCRE syntax (https://perldoc.perl.}
CertiStr: A Certified String Solver

The framework of CertiStr. The input of the algorithm will check the consistency with constraints already assumed. Since we also show that forward propagation is complete for acyclic formulas, we say that \( \mu \) satisfies \( c \) if: (i) \( \mu(x) \in \mathcal{L}(\mathcal{A}) \) for each constraint \( x \in \mathcal{A} \) in \( c \), and (ii) \( \mu(x) = \mu(x_1) \mu(x_2) \) for each constraint \( x = x_1 + x_2 \). The constraint \( c \) is satisfiable if one such solution \( \mu \) for \( c \) exists.

Our string constraint language is as general as word equations with regular constraints \([18, 20, 23, 30]\), which forms the basis of the recently published Unicode string constraint language in SMT-LIB 2.6, and already makes up the bulk of existing string constraint benchmarks. Notice that our restriction to string equation of the form \( x = x_1 + x_2 \) is not a real restriction since general string constraints can be obtained by means of desugaring. For instance, the concatenation of more than two variables, like \( x = x_1 + x_2 + x_3 \) can be desugared as two concatenation constraints: (1) \( x' = x_1 + x_2 \); \( x = x' + x_3 \). Moreover, a subset of string constraints with length functions (called monadic length) can also be translated into this fragment with regular constraints. For instance, \( |x| \leq 6 \), where \( |x| \) denotes the length of the word of \( x \), can be translated to a regular membership constraint as \( x \in \mathcal{A} \), where \( \mathcal{A} \) is an NFA that accepts any word whose length is at most 6. Recent experimental evidence shows that a substantial portion of length constraints that appear in practice are essentially monadic \([21]\). There are also some string constraints using disjunction (\( \lor \)). For instance, \( c_1 \lor c_2 \), where \( c_1 \) and \( c_2 \) are string constraints in our fragment. This also can be solved with CertiStr by first solving \( c_1 \) and \( c_2 \) separately, and then checking whether one of them is satisfiable.

Figure 1 shows the framework of CertiStr. The input of the solver is an SMT file with string constraints. The tool has two parts: (1) the front-end (non-certified) and (2) the back-end (certified). The front-end parses the SMT file, which

\[
url \in /http://[a-zA-Z0-9.]+/[a-zA-Z0-9.]+/ 
\]
will first translate all Regular Expressions (RE) to NFAs and desugar some string constraints that are not in the fragment, and then translate the string constraints to the Intermediate Data Structures (IDS), which are the inputs of the back-end. The IDS contains three parts: (1) a set \( S \) of variables that are used in the string constraints, (2) a concatenation constraint map \( \text{Concat} \), which maps variables in \( S \) to the set of their concatenation constraints, and (3) a regular constraint map \( \text{Reg} \), which maps each variable in \( S \) to its regular membership constraints.

The back-end contains two parts: (1) the Automata Library, which contains a collection of automata operations, such as the product and concatenation of two NFAs, and (2) a forward-propagation to check whether the string constraints are satisfiable. The results of the certified string solver can be (1) \( \text{sat} \) the string constraint is satisfiable, (2) \( \text{unsat} \) the string constraint is unsatisfiable, and (3) \( \text{unknown} \) the solver cannot decide whether the string constraint is satisfiable or not. Note that, \( \text{unknown} \) does not mean that CertiStr is non-terminating. It means the results after executing the forward-propagation for string constraints are not sufficient to decide their satisfiability.

4 Forward-propagation of String Constraints

In this section, we present the algorithm of the forward-propagation and the properties proven for it.

4.1 The Algorithm of Forward-propagation

We use the following example of string constraints to illustrate our forward-propagation analysis.

**Example 4.1.** \( x_1 \in A_1 \land x_2 \in A_2 \land x_3 \in A_3 \land x_5 = x_3 + x_4 \land x_3 = x_1 + x_2 \).

![Figure 2. Dependence graph of variables in Example 4.1](image)

Example 4.1 contains five variables \( x_1, x_2, x_3, x_4, x_5 \) and two concatenation constraints \( x_3 = x_3 + x_4 \land x_3 = x_1 + x_2 \). We can view these concatenation constraints as a dependence graph shown in Figure 2. For a concatenation constraint \( x_k = x_i + x_j \), it yields two dependence relations: \( x_k \leftrightarrow x_i \) and \( x_k \leftrightarrow x_j \). This means that before propagating the concatenation of the regular constraints of \( x_i \) and \( x_j \), we need to first compute the propagation to \( x_i \) and \( x_j \). Let the initial regular constraint of \( x_k \) be \( A_k \). Let the regular constraints of \( x_i \) and \( x_j \) be \( A_i \) and \( A_j \), respectively. The new regular constraint of \( x_k \) after the propagation can be refined to an automaton, whose language is \( L(A_k) \cap (L(A_i) + L(A_j)) \), where \( L_1 + L_2 = \{w_1w_2 | w_1 \in L_1 \land w_2 \in L_2\} \). Here, the term "refine" means narrowing the regular constraint of a variable by propagating the concatenation constraints.

The forward-propagation repeats the following computations until all variables are refined:

- detect a set of variables such that for each variable in the set, all its dependence variables have already been refined.
- refine the regular constraints of the variables detected in the last step with respect to their concatenation constraints.

We illustrate the idea of the forward-propagation with Example 4.1. The first iteration of the forward-propagation will detect the set of variables \( \{x_1, x_2, x_4\} \) since they have no dependence variables and we do not need to refine their regular constraints. The second iteration of the forward-propagation will detect the set of variables: \( \{x_3\} \), as \( x_3 \) only depends on the variables in \( \{x_1, x_2, x_4\} \). Its regular constraint will be refined to an automaton, whose language is \( L(A_3) \cap (L(A_1) + L(A_2)) \) after the propagation for the constraint \( x_3 = x_1 + x_2 \). Assume this new regular constraint as \( A'_3 \).

The third iteration detects the variable \( x_5 \), whose regular constraint will be refined to an automaton, whose language is \( \Sigma' \cap (L(A'_3) + \Sigma') \). Note that we do not specify the initial regular constraints of \( x_4 \) and \( x_3 \) in the string constraint and therefore their initial regular constraints are \( \Sigma' \) by default.

Algorithm 1 shows the forward-propagation algorithm. It contains three procedures: \texttt{Forward Prop, Ready Set, and Var Lang}. The procedure \texttt{Forward Prop} is the entry point
Algorithm 1 The algorithm of forward-propagation

1: procedure Forward_Prop(S, Concat, Reg)
2:     \( R \leftarrow \emptyset \)
3:     \( \text{while } \mathcal{C} \neq \emptyset \text{ do} \)
4:         \( \mathcal{C} \leftarrow \text{Ready_Set}(S, \text{Concat}, R) \)
5:         \( \text{Reg} \leftarrow \text{Var.Lang}(C, \text{Concat}, \text{Reg}) \)
6:         \( S \leftarrow S \cup C; R \leftarrow R \cup C \)
7:     \( \text{end while} \)
8:     \( \text{return } \text{Reg} \)
9: end procedure

10: procedure Var.Lang(C, Concat, Reg)
11:     \( \text{for each } v \in C \text{ do} \)
12:         \( \mathcal{A} \leftarrow (\text{Reg } v) \)
13:         \( \text{for each } (v_1, v_2) \in (\text{Concat } v) \text{ do} \)
14:             \( \mathcal{A}' \leftarrow \text{NFA_concat } (\text{Reg } v_1) (\text{Reg } v_2) \)
15:             \( \mathcal{A} \leftarrow \mathcal{NFA_product } \mathcal{A} \mathcal{A}' \)
16:         \( \text{end for} \)
17:     \( \text{Reg} \leftarrow \text{Reg } [v \mapsto \mathcal{A}] \)
18: end for
19: \( \text{return } \text{Reg} \)
20: end procedure

21: procedure Ready_Set(S, Concat, R)
22:     \( C \leftarrow \emptyset \)
23:     \( \text{for each } v \in S \text{ do} \)
24:         \( D \leftarrow \emptyset \)
25:         \( \text{for each } (v_1, v_2) \in (\text{Concat } v) \text{ do} \)
26:             \( D \leftarrow D \cup \{v_1, v_2\} \)
27:         \( \text{end for} \)
28:     \( \text{if } D \subseteq R \text{ then} \)
29:         \( C \leftarrow C \cup \{v\} \)
30:     \( \text{end if} \)
31: \( \text{return } C \)
32: end procedure

In the while loop of the procedure Forward_Prop, the computation iteratively refines the regular constraints of variables by propagating the regular constraints of variables on the right hand side of the concatenation constraints. The first step in the loop computes the set \( C \) of variables whose dependence variables are all in the set \( R \) (computed by the procedure Ready_Set). \( R \) is initialized empty and it denotes the set of variables that have already been refined by the previous loops. The second step in the loop updates the regular constraints of the variables in \( C \) stored in \( \text{Reg} \) by propagating the concatenation constraints in Concat (computed by Var.Lang).

At the end of the loop body, the set \( S \) is updated by removing the variables in \( R \) as they have already been refined and these removed variables are added to the set \( R \), i.e., they have already been refined.

The procedure Ready_Set computes the set of variables that are only dependent on the variables in \( R \). For each variable \( v \), it stores all its dependence variables in \( D \) and then checks whether \( D \) is a subset of \( R \). If \( D \) is a subset of \( R \) then \( v \) is moved to \( C \) as all its dependence variables have been refined.

The procedure Var.Lang refines the regular constraints of variables in \( C \). In the inner loop of Var.Lang, it traverses all concatenation constraints \( v = v_1 + v_2 \) and refines the regular constraint of \( v \) by propagating the concatenation of the regular constraints of \( v_1 \) and \( v_2 \). \( \text{Reg } [v \mapsto \mathcal{A}] \) means updating the regular constraint of \( v \) in \( \text{Reg} \) to \( \mathcal{A} \). The functions NFA_concat and NFA_product denote constructing the concatenation and product of two NFAs, respectively. These two functions will be introduced in Section 5.

Assume a string constraint is stored in \( S \), Concat and \( \text{Reg} \), then after calling “Forward_Prop(S, Concat, Reg)”, we will get a new \( \text{Reg} \) such that for each variable \( v \), \( \text{(Reg'} v) \) stores the refined regular constraint of \( v \), which is an NFA. We have two cases to decide whether the string constraint is satisfiable or not:

- If there exists a variable \( v \) such that the language of \( \text{(Reg'} v) \) is empty then the string constraint is unsatisfiable, because the refined regular constraint of \( v \) is the empty language.
- If for all variables \( v \) such that the language of \( \text{(Reg'} v) \) is not empty then unfortunately we cannot decide whether the string constraint is satisfiable or not. Consider the string constraint: \( (1) \ y = x + x \land (2) \ y \in \{ab\} \land (3) \ x \in \{a, b\} \)”, where \( ab \), \( a \), and \( b \) are words, the refined regular constraints of \( x \) and \( y \) are both not empty. But this string constraint is not satisfiable as constraints (1) and (2) require \( x \) to have two different values. We will introduce a property, called tree property later, such that if the string constraint satisfies the tree property then it is satisfiable, otherwise CertiStr returns unknown.

4.2 Theorems for Forward-propagation

Now we introduce the key theorem proven for the algorithm. Firstly we present the definitions of well-formed inputs (wf in Figure 3) and the acyclic property (acyclic in Figure 3) for concatenation constraints, which are the premises of the correctness theorem.
definition \texttt{wf} where
\begin{align*}
\texttt{wf} \ S \ \text{Concat} \ \text{Reg} &= \\
& (\text{dom} \ \text{Concat} \subseteq S) \land (\text{dom} \ \text{Reg} = S) \land \\
& (\forall v \ v_1, v_2 \ v \in (\text{dom} \ \text{Concat}) \land (v_1, v_2) \in (\text{Concat} \ v) \\
& \quad \rightarrow v_1 \in S \land v_2 \in S) \land \text{finite} \ S
\end{align*}

\textbf{Definition \texttt{acyclic} where}
\begin{align*}
\text{acyclic} \ S \ \text{Concat} \ I &= S = \bigcup (\text{set} \ I) \land (|I| = 0) \\
& \lor (|I| = 1) \\
& \lor (\forall v_1, v_2 \ v_1, v_2 \in S \land (v_1, v_2) \in (\text{Concat} \ v) \\
& \quad \rightarrow (v_1, v_2) \subseteq \bigcup (\text{set} \ I') \land (\text{acyclic} \ (S - v) \ \text{Concat} \ I'))
\end{align*}

\textbf{Figure 3. The definitions of \texttt{wf} and \texttt{acyclic}}

The \texttt{wf} predicate requires that all variables appearing in the domain and codomain of \texttt{Concat} should be in the set \(S\), and the domain of \texttt{Reg} should be the set \(S\), i.e., all variables should have their initial regular constraints. \(S\) should be finite. Now we consider the definition of \texttt{acyclic}, which checks whether the dependence graphs of concatenation constraints are acyclic. \texttt{Forward Prop} cannot solve string constraints that are not acyclic. For instance, the string constraint \(\texttt{x} = y_1 + y_2 \land y_1 = z_1 + x^k\) does not satisfy the acyclic property, as \(x \mapsto y_1 \mapsto x\).

In order to specify the acyclic property, we arrange the set \(S\) of variables to a list in which elements are disjoint sets of variables in \(S\) and the list characterizes the dependence levels among variables. For instance, in Example 4.1, there are 5 variables and they can be organized in a list: \([\{v_5\}, \{v_3\}, \{v_1, v_2, v_4\}]\). Indeed, the list characterizes the dependence levels among variables. For instance, the variable \(v_1\) only depends on the variables in \(\{v_3\}\) and \(\{v_1, v_2, v_4\}\) and \(v_3\) only depends on the variables in \(\{v_1, v_2, v_4\}\) but not \(\{v_3\}\). The variables in \(\{v_1, v_2, v_4\}\) do not depend on any variables. If \texttt{Concat} does not satisfy the acyclic property then we cannot arrange all variables in such a list, as all variables in a cycle always depend on another variable in the cycle.

The predicate \texttt{acyclic} contains two parts. (1) \(S = \bigcup (\text{set} \ I)\), which means the elements in \(S\) and \(I\) should be exactly the same. \(\text{set} \ I\) translates the list \(I\) to a set of elements in \(I\) and \(\bigcup (\text{set} \ I) = \{x \mid \exists y \in \text{set} \ I \land x \in y\}\). (2) Part 2 further has two cases. The first case is \(|I| = 1\), i.e., empty list. In this case, the predicate is obviously true. The second case is \(|I| = s\#I'\), i.e., it has at least one element. \(s\) is the first element and \(I'\) is the tail. In this case, firstly, it requires the elements in \(s\) to be disjoint with the elements in the sets of the tail \(I'\) (checked by \(s \cap \bigcup (\text{set} \ I') = \emptyset\)). Secondly, let \(v\) be a variable in \(s\). For any two variables \(v_1\) and \(v_2\) that \(v\) depends on, \(v_1\) and \(v_2\) can only be in the sets of the tail \(I'\). Thirdly, \texttt{acyclic} checks whether the remaining elements in \(S - s\) are still acyclic w.r.t. \texttt{Concat} and \(I'\). In order to check whether the variables in \(S\) are acyclic w.r.t. \texttt{Concat}, we can specify it as “\(\exists! \text{acyclic} \ S \ \text{Concat} \ I\)”.

Now we are ready to introduce our correctness theorem (Theorem 4.2). The keyword \texttt{fixes} in Isabelle is used to define universally quantified variables. The keyword \texttt{assumes} is used to specify assumptions of the theorem and the keyword \texttt{shows} is used to specify the conclusion of the theorem. The \texttt{shows} part is of the form “\(M \leq \text{SPEC} (\lambda \text{Reg}'. \ \Phi)'\), which means the algorithm \(M\) is correct with respect to the specification \(\Phi\). More precisely, the result of applying the predicate \((\lambda \text{Reg}'. \ \Phi)\) to the output of \(M\) should be true. “\(\text{NFA accept } w \ \mathcal{A}\)” checks whether \(w\) is accepted by \(\mathcal{A}\), which will be introduced in Section 5. The symbol “\(@\)” is the list concatenation operation in Isabelle. We use a list of elements in \(\Sigma\) to represent a word.

\textbf{Theorem 4.2 (Correctness of Forward Prop).}
\begin{itemize}
\item \texttt{fixes} \(S \ \text{Concat} \ \text{Reg}\)
\item \texttt{assumes} 1. \texttt{wf} \(S \ \text{Concat} \ \text{Reg}\)
2. \(\exists! \ \text{acyclic} \ S \ \text{Concat} \ I\)
\item \texttt{shows} \(\text{Forward Prop}(S, \ \text{Concat}, \ \text{Reg}) \leq \text{SPEC}(\lambda \text{Reg}'. \ \Phi)\)
\begin{itemize}
\item \((\forall w. \ \text{NFA accept} \ w \ \text{(Reg'} w)) \rightarrow \text{NFA accept} w \ \text{(Reg} w)\)
\item \((\forall w_1, w_2. \ w_1 = w_1 @ w_2 \land \text{NFA accept} \ w_1 \ \text{(Reg'} \ w_1) \land \text{NFA accept} \ w_2 \ \text{(Reg'} w_2)))\)
\end{itemize}
\end{itemize}

Theorem 4.2 shows that if \(S, \ \text{Concat}, \ \text{Reg}\) are well-formed and acyclic then after calling \texttt{Forward Prop}(\(S, \ \text{Concat}, \ \text{Reg}\)) we can get \texttt{Reg’}, which satisfies: any \(w\) is accepted by the automaton \((\text{Reg} \ w)\) if and only if
\begin{itemize}
\item 1. \(w\) is accepted by the original regular constraint of \(v\), i.e., \((\text{Reg} w)\), and
\item 2. for all variables \(v_1\) and \(v_2\), \((v_1, v_2) \in \text{Concat} v\) implies there exist \(w_1, w_2\) such that \(w = w_1 @ w_2\) and \(w_i\), for \(i = 1, 2\), is accepted by the regular constraint \((\text{Reg} \ v_i)\).
\end{itemize}

That is, in the new \texttt{Reg’}, the concatenation constraints are used to refine the corresponding variables.

Moreover, in Isabelle, Theorem 4.2 also requires us to prove the termination of Algorithm 1. The termination is proven by showing that in the procedure \texttt{Forward Prop}, the number of variables in \(S\) decreases progressively for each iteration. Since \(S\) is finite, the algorithm must terminate.

Even though Theorem 4.2 is the key of the correctness of the forward-propagation, it does not give us an intuition with respect to the satisfiability semantics of string constraints introduced in Section 3. In the following, we will show that our forward-propagation is sound w.r.t. \texttt{unsat} results.

Firstly we need to formalize the semantics of \texttt{sat} and \texttt{unsat} for string constraints. Let \(\mu\) be an assignment, which is a map \(S \rightarrow \Sigma^*\) from variables in \(S\) to words. We define the predicate \texttt{sat_str} in Figure 4 for the semantics of the satisfiability of string constraints.
The predicate \texttt{sat\_str} contains two parts: (1) the assignment \(\mu\) should respect all regular constraints and (2) the assignment \(\mu\) should respect all concatenation constraints.

Theorem 4.3 shows that after executing \texttt{Forward\_Prop}, we get a new map \(\texttt{Reg}'\). If there exists a variable \(v\), such that \(L(\texttt{Reg}'\ v) = \emptyset\), i.e., the language of the refined regular constraint of \(v\) is the empty set, then there is no assignment \(\mu\) that makes the predicate \(\texttt{sat\_str} \ S \ \texttt{Concat} \ \texttt{Reg} \ \mu\) true.

Unfortunately, the theorem holds only for \texttt{unsat} results. As we already analyzed before, the case \(\forall v. L(\texttt{Reg}'\ v) \neq \emptyset\) cannot make us conclude that the string constraint is satisfiable.

In this paper, we characterize a subset of the string constraints, using a property such that the forward-propagation’s satisfiable results are also sound. We call it the \textit{tree property}. The intuition behind the \textit{tree property} is that the dependence graph of a string constraint should constitute a tree or forest. The example \(y = x + x\) does not satisfy the \textit{tree property}, because \(y\) has two edges pointing to a same node \(x\) in the dependence graph. In other words, the node \(x\) has indegree 2. But a tree requires that each node has at most indegree 1. As the map \texttt{Concat} stores the dependence graphs of string constraints, we only need to check \texttt{Concat} for the tree property. The predicate \texttt{tree} is defined in Figure 4.

This definition has three cases separated by the conjunction operator: (1) the first case requires that for any concatenation constraint \(v = v_1 + v_2\), we have \(v_1 \neq v_2\). (2) The second case requires that for any variable \(v\) with two concatenations \(v = v_1 + v_2\) and \(v = v_3 + v_4\), \(\{v_1, v_2\}\) and \(\{v_3, v_4\}\) are disjoint. (3) The third case requires that for any two different variables \(v\) and \(v'\) with the concatenation constraints \(v = v_1 + v_2\) and \(v' = v_3 + v_4\), \(\{v_1, v_2\}\) and \(\{v_3, v_4\}\) are disjoint.

With this definition we have the following theorem.

**Theorem 4.4 (Completeness for the tree property).**

**Proof:**

**Assumptions:**

1. \(\texttt{wf} \ S \ \texttt{Concat} \ \texttt{Reg}\)
2. \((\exists! \ l. \texttt{acyclic} S \ \texttt{Concat} \ l)\)
3. \(\texttt{tree} \ \texttt{Concat}\)

**Shows:**

\begin{align*}
\text{Forward\_Prop}(S, \texttt{Concat}, \texttt{Reg}) \leq \text{SPEC}(\lambda \texttt{Reg}'. ((\forall v. L(\texttt{Reg}'\ v) \neq \emptyset)) \\
(\exists \mu. \texttt{sat\_str} \ S \ \texttt{Concat} \ \texttt{Reg} \ \mu)))
\end{align*}

This theorem shows that if \texttt{Concat} satisfies the tree property then we have: \(\texttt{Reg}'\) satisfies that for all \(v\), the language of \(\texttt{Reg}'\ v\) is not empty if there exists an assignment, which makes the string constraint satisfiable.

In order to check whether a string constraint satisfies the tree property, we propose Algorithm 2. It first stores all the variables, which are on the right hand side of the concatenation constraints, into a list, and then checks whether the list is distinct. If it is not distinct then there must exist a variable with at least indegree 2 and thus \texttt{Concat} cannot satisfy the tree property.

**Algorithm 2 Tree property checking**

```
1: procedure Check\_Tree(S, \texttt{Concat})
2: \(l \leftarrow \emptyset\)
3: for each \(v \in S\) do
4: for each \((v_1, v_2) \in (\texttt{Concat} \ v)\) do
5: \(l \leftarrow v_1 \# v_2 \# l\)
6: end for
7: end for
8: return distinct \(l\)
9: end procedure
```

Now we show that if Algorithm 2 returns \texttt{True} then the tree property is satisfied.

**Theorem 4.5 (Correctness of Check\_Tree).**

**Proof:**

**Assumptions:**

1. \(\texttt{dom} \ \texttt{Concat} \subseteq S\)
2. \((\forall v_1, v_2. (v_1, v_2) \in (\texttt{Concat} \ v) \rightarrow \{v_1, v_2\} \subseteq S)\)

**Shows:**

\begin{align*}
\text{Check\_Tree}(S, \texttt{Concat}) \rightarrow \texttt{tree} \ \texttt{Concat}
\end{align*}

With the function \texttt{Check\_Tree}, CertiStr can return three results: \texttt{unsat}, \texttt{sat}, and \texttt{unknown}, in which \texttt{unknown} means the string constraint does not satisfy the tree property and the forward-propagation does not refine the regular constraint of any variable to empty.

We now finish introducing the forward-propagation procedure. As it uses many NFA operations, in the following
section, we will introduce our implementation for these NFA operations.

5 Symbolic Automata Formalization

Automata operations are the key to our certified string solver. For instance, we need to check the language emptiness of an NFA as well as construct the concatenation and product of two NFAs. In order to make the string solver practically useful, we should also keep the efficiency in mind.

As argued by D’Antoni et. al [15], the classical definition of NFA, whose transition labels are a character in the alphabet, is not suitable for practical implementation, and they propose s-NFAs, which allow transition labels to carry a set of characters, to address this limitation. In this section, we present our formalization of s-NFAs in Isabelle.

Definition 5.1 (Abstract s-NFAs). An s-NFA is a 5-tuple: $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$, where $Q$ is a finite set of states, $\Sigma$ is an alphabet ($\Sigma$ may be infinite), $\Delta \subseteq Q \times 2^\Sigma \times Q$ is a finite set of transition relations, $I \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states.

For a transition $(q, \alpha, q')$ in an s-NFA, the implementation of $\alpha$ can use various different data structures, such as intervals for string solvers and Binary Decision Diagrams (BDD) for symbolic model checkers. Definition 5.1 of s-NFAs is called abstract s-NFAs because the transition labels are denoted as sets instead of using concrete data structures, such as intervals and BDDs.

In order to make our formalization of s-NFAs reusable for different implementation of transition labels, we exploit Isabelle’s refinement framework and divide the formalization of s-NFAs in two levels: (1) the abstract level (introduced in this section) and (2) the implementation level (introduced in the following section). When the correctness of the algorithms at the abstract level is proven, the implementation level does not need to re-prove it, only the refinement relations between these two levels must be shown.

At the abstract level, transition labels are modeled as a set as shown in Definition 5.1. At the implementation level, the sets of characters are refined to concrete data structures, such as intervals or BDDs.

Our formalization of s-NFAs extends an existing classical NFA formalization developed by Tuerk et al. [40]; this formalization is part of the verified CAVA model checker [26]. We extend it to support s-NFAs and further operations, such as the concatenation operation of two NFAs. Figure 5 shows the Isabelle formalization of s-NFAs. An s-NFA is defined by a record with four elements: (1) $Q$ is the set of states, (2) $I$ is the set of initial states, (3) $F$ is the set of accepting states, and (4) $\Delta$ is the set of transitions with transition labels defined by sets. The definition well-formed is a predicate to check whether an NFA satisfies (1) the states occurring in the sets of initial and accepting states are also in the set $(Q, \mathcal{A})$ of states. In Isabelle, the value of a field $fd$ in a record $rd$ can be extracted by $(fd rd)$. Therefore, $(Q, \mathcal{A})$ denotes the value of the field $Q$ in $\mathcal{A}$. (2) The states in the set of transitions should also be in $(Q, \mathcal{A})$. (3) $(Q, \mathcal{A})$ is a finite set. (4) $\Delta$ is finite.

For a word $w = a_1 a_2 \ldots a_n$ and two states $q$ and $q'$, the function reachable checks whether there exists a path $q_0, a_1, q_1, \ldots, a_n, q_n$ in the NFA $\mathcal{A}$ such that $q = q_0$ and $q_1' = q_2, \ldots, a_{n-1}, q_{n-1}, a_n, q_n \in (\Delta, I), 1 \leq i \leq n$ and $a_i \in a_i$, $1 \leq j \leq n$. The definition NFA_accept checks whether a word is accepted by an NFA and the definition of $L$ is the language of an NFA.

Based on this definition of NFAs, a collection of automata operations are defined and their correctness lemmas are proven. Here we only present the operation of the concatenation shown in Figure 6.

NFA_concat_basic constructs the concatenation of two NFAs. The correctness of this definition relies on the fact that the sets of states in the two NFAs are disjoint. In the definition of NFA_concat_basic, the set of transitions of the concatenation is the union of (1) $(\Delta, \mathcal{A}_1)$, (2) $(\Delta, \mathcal{A}_2)$, and (3) $\{(q, a, q') : q \in (\Delta, \mathcal{A}_1) \land q' \in (\Delta, \mathcal{A}_2)\}$. The transitions in (3) concatenate the language of $\mathcal{A}_1$ to the language of $\mathcal{A}_2$.

The set of the initial states in the concatenation is computed by first checking whether there is a state that is in both the sets of initial and accepting states of $\mathcal{A}_1$. If there exists such a state then the initial states in the concatenation should include the initial states of $\mathcal{A}_2$, as $\mathcal{A}_1$ accepts the
definition NFA_concat_basic where
NFA_concat_basic $A_1$ $A_2$ ≡ |
Q = (Q $A_1$) ∪ (Q $A_2$),
Δ = (Δ $A_1$) ∪ (Δ $A_2$) ∪ 
{(q, a, q”) | ∃q’. (q, a, q’) ∈ (Δ $A_1$) ∧ 
q’ ∈ (F $A_1$) ∧ q” ∈ (I $A_2$)},
I = if ((I $A_1$) ∩ (F $A_1$)) = ∅ then (I $A_1$) 
else ((I $A_1$) ∪ (I $A_2$)),
F = (F $A_2$)
|

definition NFA_concat where
NFA_concat $A_1$ $A_2$ $f_1$ $f_2$ =
remove_unreachable_states
(NFA_concat_basic
(NFA_rename $f_1$ $A_1$)) (NFA_rename $f_2$ $A_2$))

Figure 6. The definition of concatenation operation

empty word, otherwise the initial states of the concatenation are $A_1$’s initial states.

The definition of NFA_concat firstly renames the states in both NFAs to ensure their sets of states disjoint. In the term "NFA_rename $f$ $A$", $f$ is a renaming function, for instance, $f$ can be “$\lambda$q. (q, 1)”, which renames any state q to (q, 1). "NFA_rename $f$ $A$" renames all states in (Q $A$) by the function $f$. Correspondingly, all states in transitions, initial states, and accepting states are also renamed by $f$. Renaming states makes it easier to ensure the sets of states of two NFAs are disjoint. The function remove_unreachable_states removes all states in an NFA that are not reachable from the initial states of the NFA. This enables us to check the language emptiness of an NFA by only checking the emptiness of its set of accepting states. At the implementation level, the algorithm also only generates reachable states for NFAs. But with remove_unreachable_states states used in NFA_concat at the abstract level, the refinement relation between these two levels can be specified as the NFA isomorphism of the two output concatenations of automata at the two levels, instead of language equivalence, which is more challenge to prove.

In order to ensure the correctness of this operation, we need to prove the following lemma in Isabelle.

Lemma 5.2 (Correctness of NFA_concat).

fixes $A_1$ $A_2$ $f_1$ $f_2$

assumes 1. well-formed $A_1$
2. well-formed $A_2$
3. (image $f_1$ (Q $A_1$)) ∩ (image $f_2$ (Q $A_2$)) = ∅
4. inj_on $f_1$ (Q $A_1$) ∧ inj_on $f_2$ (Q $A_2$)

shows $L$(NFA_concat $A_1$ $A_2$ $f_1$ $f_2$) = 
{w_1@w_2, w_1 ∈ L($A_1$) ∧ w_2 ∈ L($A_2$)}

The assumptions well-formed $A_1$ and well-formed $A_2$ require the input NFAs to be well-formed. In addition, the assumption (image $f_1$ (Q $A_1$)) ∩ (image $f_2$ (Q $A_2$)) = ∅ requires that after renaming the states in both $A_1$ and $A_2$, the new sets of states of the two NFAs are disjoint. The term “image f S” applies the function $f$ to the elements in the set S and returns a new set {f e | e ∈ S}. The premises inj_on $f_1$ (Q $A_1$) and inj_on $f_2$ (Q $A_2$) require that the functions $f_1$ and $f_2$ are injective over the sets (Q $A_1$) and (Q $A_2$), respectively. The conclusion in “shows” specifies that the language of the concatenation of $A_1$ and $A_2$ equals the set of the concatenations of the words in $L$($A_1$) and the words in $L$($A_2$).

Moreover, some other important functions are defined, for instance:

- The product of two NFAs: NFA_product $A_1$ $A_2$, which computes the product of $A_1$ and $A_2$. We proved the theorem: $L$(NFA_product $A_1$ $A_2$) = $L$($A_1$) $\cap$ $L$($A_2$).
- Checking the isomorphism of two NFAs: NFA_isomorphism $A_1$ $A_2$ ≡ (∃f. inj_on f (Q $A_1$) ∧ NFA_rename f $A_1$ = $A_2$). Some lemmas are proven for it, such as, well-formed $A_1$ $\cap$ well-formed $A_2$ ∧ NFA_isomorphism $A_1$ $A_2$ $\rightarrow$ $L$($A_1$) = $L$($A_2$).

Section 4 and this section present the abstract level algorithms for the forward-propagation and the NFA operations, respectively. In order to make CertiStr efficient and practically useful, we need to use some efficiently implemented data structures, such as red-black-trees (RBT) and hash maps. This will be introduced in the following section.

6 Implementation-Level Algorithms

In this section, we will first present the relations between abstract concepts, like sets and maps, and the implementation data structures in Isabelle (Subsection 6.1), which is the prerequisite to understand the implementation-level algorithm in Subsection 6.2.

6.1 Isabelle Collections Framework

Isabelle Collections Framework (ICF) [27] provides an efficient, extensible, and machine checked collections framework. The framework features the use of data refinement techniques [25] to refine an abstract specification (using high-level concepts like sets) to a more concrete implementation (using collection data structures, like RBT and hashmaps).

The code-generator of Isabelle can be used to generate efficient, extensible, and machine checked collections framework. Isabelle Collections Framework (ICF) [27] provides an efficient, extensible, and machine checked collections framework. The framework features the use of data refinement techniques [25] to refine an abstract specification (using high-level concepts like sets) to a more concrete implementation (using collection data structures, like RBT and hashmaps). The code-generator of Isabelle can be used to generate efficient code.

The concrete data structures implement a collection of interfaces that mimic the operations of abstract concepts. We list the following RBT implementation of set interfaces that will be used to present the implementation-level algorithms:

- RBT.empty, generates an RBT representation of the empty set.
- RBT.mem e D, checks whether the element e is in the set represented by RBT D.
NFA_rbt = | q : "'q RBT" |
\( \Delta : \) ("'q * 'a Interval * 'q\) RBT |
I : "'q RBT" |
F : "'q RBT"

Figure 7. NFA data structure definition using RBT

- \texttt{RBT.insert} \( e \) \( D \), inserts the element \( e \) into the set represented by \( RBT \) \( D \).
- \texttt{RBT.inter} \( D_1, D_2 \), computes the RBT representation of the set intersection of \( D_1 \) and \( D_2 \). \( D_1 \) and \( D_2 \) are both RBT represented sets.
- \texttt{RBT.union} \( D_1, D_2 \), computes the RBT representation of the set union of \( D_1 \) and \( D_2 \). \( D_1 \) and \( D_2 \) are both RBT represented sets.
- \texttt{RBT.iterator} \( D \) \( f \), where \( f \) is of the type \( t_1 \Rightarrow t_2 \Rightarrow t_2, v \) is of the type \( t_2 \), and the elements in \( D \) are of the type \( t_1 \). This interface implementation iteratively applies the function \( f \) to the elements in \( D \) with the initial value \( v \). For instance, the first iteration of the computation of \( "RBT.iterator\ \{ e_1, e_2\} \ v \) \( f " \) selects an element from \( \{ e_1, e_2\} \), assume \( e_1 \) is selected, and then applies \( f \) to \( e_1 \) and \( v \), i.e., \( "f \ e_1 \ v " \). Assume the value of \( "f \ e_1 \ v " \) is \( v' \). The second iteration applies \( f \) to \( e_2 \) and \( v' \). Now all elements in the set are traversed, \( "RBT.iterator\ \{ e_1, e_2\} \ v \) \( f " \) finishes with the return value of \( "f \ e_2 \ v " \).
- \texttt{RBT.to list} \( D \) translates a set represented by \( D \) to a list.
- \texttt{RBT.a D} translates RBT represented set \( D \) to the set in the abstract concept.

6.2 Implementation-Level Algorithm for \texttt{NFA.concat}

In this subsection, we present the implementation-level algorithm by the NFA operation \texttt{NFA.concat}.

In order to formalize implementation-level algorithms, the first step is to consider the data refinement. At the abstract level, we use sets to store states, transitions, initial states, and accepting states. At the implementation level, we use RBTs to replace sets to store these elements of NFAs. The type of NFAs using RBT data structures is specified in Figure 7. Note that, here we only give a simplified version of the RBT NFA definition to make it easy to understand. Indeed, RBT has more type arguments.

Moreover, for each transition \( (q, \alpha, q') \), the label \( \alpha \) is represented by a set at the abstract level. We use intervals to replace sets to store \( \alpha \). An interval \([n_1, n_2]\) is a pair of elements in a totally ordered set. The semantics of an interval is defined by a set as \texttt{semI} \[ n_1, n_2 \] = \{ \( n \mid n_1 \leq n \leq n_2 \) \}.

In order to support NFA operation implementation, some operations for intervals are implemented. We list the following 3 interval operations here.

- Computing the intersection of two intervals: \texttt{intersection} \([ I_1, I_2 \] \[ I_1', I_2' \]) = \{ \max(I_1, I_1'), \min(I_2, I_2') \}. The term \( \max(I_1, I_1') \) returns the bigger one of \( I_1 \) and \( I_1' \), and \( \min(I_2, I_2') \) returns the smaller one of \( I_2 \) and \( I_2' \).
- Checking the non-emptiness of the interval \([ I_1, I_2 \]): \texttt{emptyI} \([ I_1, I_2 \]) = \{ I_1 \leq I_2 \}.
- Checking the membership of an element \( e \) and the interval \([ I_1, I_2 \]): \texttt{memI} \([ e, I_1, I_2 \]) = \{ I_1 \leq e \leq I_2 \}.

Algorithm 3 The algorithm idea of NFA concatenation

```
1: procedure Construct_Trans(\( S_{A_1}, S_{A_2}, S_{T_2}, S_{F_1} \))
2: \( S_D \leftarrow \text{RBT.union} \( S_{A_1}, S_{A_2} \)\)
3: \( \text{RBT.iterator} \( S_D \) \( I \) \( F \) \( \lambda(q, a, q') \) \( S \)\)
4: \( \text{if} \ (\text{RBT.mem} q' \ S_{F_1}) \ \text{then} \)
5: \( \text{RBT.iterator} \( S_T \) \( S \)\)
6: \( \lambda(q' \ S'). \text{RBT.insert} (q, a, q') \ S' \)
7: \( \text{end if} \);
8: \( \text{return} \ S_D \);
9: end procedure

10: procedure NFA_Concat_Impl(\( A_1, A_2, f_1, f_2 \))
11: \( A_1', A_2' \leftarrow \text{NFA_Rename_Impl} \( f_1, A_1; A_2 \)\)
12: \( A_1'' \leftarrow \text{NFA_Rename_Impl} \( f_2, A_1' \)\)
13: \( \text{if} \ (\text{RBT.inter} (A_1', (f \ A_1' \ F \ A_1')) \neq \text{RBT.empty}) \ \text{then} \)
14: \( S_I \leftarrow \text{RBT.union} (A_1'', (I \ A_2'')) \)
15: \( \text{else} \)
16: \( S_I \leftarrow (I \ A_1') \)
17: \( \text{end if} \);
18: \( S_Q \leftarrow S_I; S_A \leftarrow \text{RBT.empty}; \text{wl} \leftarrow \text{RBT.to list} \( S_Q \)\)
19: \( \text{while} \ \text{wl} \neq [] \ \text{do} \)
20: \( q_s \leftarrow \text{wl.rm_first} \)
21: \( \text{if} \ q_s = q_a \ \text{and} \ (\text{emptyI} \alpha) \ \text{then} \)
22: \( \text{if} \ \neg (\text{RBT.mem} q_s \ S_Q) \ \text{then} \)
23: \( \text{RBT.insert} q_s \ S_Q; \)
24: \( \text{wl} \leftarrow \text{wl}@[q_s] ; \)
25: \( \text{end if} \);
26: \( \text{end if} \);
27: \( \text{end while} \)
28: \( \text{return} \ (S_Q, S_A, S_I, \text{RBT.inter} \ (f \ A_1'')) \)
29: end procedure
```
Algorithm 3 shows the basic idea of the NFA_concat implementation (the procedure NFAConcat_Impl). The texts after “⇒” are comments. NFA_Rename_Impl is the implementation of the NFA renaming function NFA_Rename at the abstract level.

The sub-procedure Construct_Trans generates the set of transitions for the concatenation of the two automata, which contains the transitions in both \((\Delta_1 A_1)\) and \((\Delta_2 A_2)\), and the transitions that concatenate the two automata (cf. Figure 6 for the abstract algorithm of the concatenation ).

In Algorithm 3, Line 16-20 computes the initial states of the concatenation. Line 21 initializes \(S_0, S_\Delta, \) and \(wl\), where \(S_0\) stores the states of the concatenation NFA, \(S_\Delta\) stores the transitions of the concatenation NFA, \(wl\) is a list to mimic a queue that stores the states to be expanded. Line 23 to 37 is the while loop for expanding the reachable states and transitions. Finally Line 38 returns the concatenation of the two NFAs. This algorithm constructs the concatenation of two NFAs with only reachable states, which means that we can check the emptiness of the concatenation NFA by only checking the emptiness of its set of accepting states.

In order to ensure the correctness of the implementation for concatenation, we need to prove the refinement relation between NFA_concat and NFA_Concate_Impl.

In Figure 8, we formalize the refinement relations between the two NFAs at the abstract level and the implementation level. The function NFA_\(\alpha\) translates an implementation-level NFA \(A\) (its type is NFA_rbt) to an abstract-level NFA of the type NFA. The definition refine_rel defines the refinement relation between an abstract-level NFA \(A\) and an implementation NFA \(A'\), which requires the isomorphism between \(A\) and \(NFA_\alpha A'\).

Lemma 6.1 specifies that the concatenation implementation correctly refines the abstract concatenation definition.

Lemma 6.1 (Correctness of NFA_Concate_Impl).

fixes \(A_1, A_2, A'_1, A'_2, f_1, f_2, f_3, f_4\)

assumes

1. refine_rel \(A_1 A'_1 \land\) refine_rel \(A_2 A'_2\)
2. inj_on \(f_1\) \((Q A_1) \land\) inj_on \(f_2\) \((Q A_2)\)
3. \((\text{image } f_1(Q A_1)) \cap \text{image } f_2(Q A_2)) = \emptyset\)
4. inj_on \(f_3\) \((\text{image } f_1(Q A'_1)) \land\) inj_on \(f_4\) \((\text{image } f_2(Q A'_2))\)
5. \((\text{image } f_3(\text{image } f_1(Q A'_1))) \cap\) \((\text{image } f_4(\text{image } f_2(Q A'_2))) = \emptyset\)
6. \(A = \text{NFA_concat}(A_1, A_2, f_1, f_2)\)
7. \(A' = \text{NFA_Concate_Impl}(A'_1, A'_2, f_3, f_4)\)

shows refine_rel \(A A'\)

This theorem shows that the implementation-level concatenation of two NFAs is a refinement for the concatenation of the corresponding abstract-level NFAs. With Lemma 6.1 and Lemma refine_rel_lang, we can easily infer that the language of the concatenation at the abstract level is equal to the language of the concatenation at the implementation level.

In this section, we presented the implementation of the concatenation operation. This implementation uses intervals to store transition labels. Indeed, our formalization of s-NFAs can be reused to implement s-NFAs using other data structures, such as BDDs, bitvectors, and Boolean formulas, to store transition labels. The corresponding operations, such as intersectionI, nemptyI, and memI, for these data structures need to be implemented at the implementation level accordingly.

Moreover, the forward-propagation algorithm and other automata operations also have their corresponding implementations and their refinement relations with the abstract algorithms have been proven.

7 Evaluation and Development Efforts

The tool CertiStr (https://github.com/uuverifiers/ostriich/tree/CertiStr) is developed in the Isabelle proof assistant (version 2020) and can extract executable OCaml code from the formalization in Isabelle. To obtain a complete solver, we added a (non-certified) front-end for parsing SMT-LIB problems (Figure 1); this front-end is written in Scala, and mostly borrowed from OSTRICH [13]. In this section, we evaluate CertiStr against the Kaluza benchmark [35], the most commonly used benchmark to compare string solvers. Moreover, we also elaborate the development efforts for CertiStr.

7.1 The Effectiveness and Efficiency

The Kaluza benchmark contains 47284 tests, among which 38043 tests (80.4%) are in the string constraint fragment of CertiStr. CertiStr supports string constraints with word equations of concatenation, regular constraints, and monadic length. The benchmark is classified into 4 groups according to the results of the Kaluza string solver: (1) sat/small, (2) sat/big, (3) unsat/small, and (4) unsat/big.
Note that the benchmark classification is not in all cases accurate, in the sense that there are some files in unsat/big and unsat/small that are satisfiable, owing to known incorrect results that were produced by the original Kaluza string solver [28, 35].

We ran CertiStr over these 38043 tests on an AMD Opteron 2220 SE machine, running 64-bit Linux and Java 1.8 (for running the Scala front-end of CertiStr). The results are shown in Table 1. The column “total tests” denotes the total number of tests in a group. The columns “sat”, “unknown”, “unsat” denote the numbers of tests for which CertiStr returns sat, unknown, and unsat, respectively. The column “solved%” denotes the percentage of the tests for which the string solver returns sat or unsat. The columns “avg.time(s)” and “timeout” denote the average time for running each test in CertiStr and the number of tests that time out, respectively. The column “CVC4” denotes the average execution time of the state of the art string solver CVC4, which can efficiently solve all the string constraints without timeout. The time limit for solving each test is 60 seconds.

From Table 1, we can conclude that in total, 83.5% of the tests can be solved by CertiStr with the result sat or unsat. The remaining of the tests are unknown or timeout. The groups sat_small and unsat_small have the higher solved percentages, more than 90%, compared with sat_big and unsat_big. This is easy to understand as small tests have a higher probability to satisfy the tree property. The worst group is unsat_big. CertiStr can solve 45% tests in this group. The reason is that in this group, there are a lot of tests with more than 100 concatenation constraints, which significantly increases their probability of violating the tree property. For the groups unsat_big and unsat_small, CertiStr detects 2502 and 7376 tests, respectively, that are indeed satisfiable.

Moreover, in unsat_big, there are 728 tests that time out. After analyzing these tests, we found that a variable with a lot of concatenation constraints (i.e., the variable appears many times on the left-hand sides of the concatenation constraints) can easily yield s-NFA state and transition explosion during the forward-propagation. In order to evaluate such a state explosion problem, we set a test of the form in Example 7.1:

**Example 7.1.**

\[ x = x_1 + x_1 \land x = x_2 + x_2 \land x = x_3 + x_3 \land \ldots \]

It is a test case with only the variable \( x \) on the left-hand side. On the right-hand side, there are concatenations of the form \( x_i + x_i \). No regular constraints are in the test.

We ran CertiStr over the test case by increasingly adding more concatenations for the variable \( x \). Table 2 shows the results for solving the test case. Each column contains (1) the number of concatenation constraints (No. Concat), (2) the size of the automaton of the variable \( x \) after executing the forward-propagation (The numbers of states and transitions of the automaton), and (3) the execution time for running the forward-propagation over the test (time).

From the table we can conclude that after adding 16 concatenation constraints for the variable \( x \), the automaton generated for \( x \) contains 65536 states and 43046721 transitions. Solving this test takes 168.28s.

Now consider the 728 tests that time out. These tests have similar constraints, in which some variables have a lot of concatenation constraints. During the forward-propagation, state and transition explosion happens for these tests, therefore they cannot be solved in 60s.

<table>
<thead>
<tr>
<th>No. Concat</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>16384</td>
<td>32768</td>
<td>65536</td>
</tr>
<tr>
<td>transitions</td>
<td>177147</td>
<td>531441</td>
<td>1594323</td>
<td>4782969</td>
<td>14348907</td>
<td>43046721</td>
</tr>
<tr>
<td>time (s)</td>
<td>0.43</td>
<td>1.29</td>
<td>4.10</td>
<td>12.37</td>
<td>50.59</td>
<td>168.28</td>
</tr>
</tbody>
</table>

**Table 1.** Experimental results of the Kaluza benchmark

<table>
<thead>
<tr>
<th></th>
<th>total tests</th>
<th>sat</th>
<th>unknown</th>
<th>unsat</th>
<th>solved%</th>
<th>avg. time(s)</th>
<th>timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>sat_small</td>
<td>19634</td>
<td>19302</td>
<td>332</td>
<td>0</td>
<td>98.3%</td>
<td>&lt; 0.01</td>
<td>0</td>
</tr>
<tr>
<td>sat_big</td>
<td>774</td>
<td>521</td>
<td>253</td>
<td>0</td>
<td>67.3%</td>
<td>0.84</td>
<td>0</td>
</tr>
<tr>
<td>unsat_small</td>
<td>8775</td>
<td>7376</td>
<td>824</td>
<td>575</td>
<td>91%</td>
<td>0.58</td>
<td>0</td>
</tr>
<tr>
<td>unsat_big</td>
<td>8860</td>
<td>2502</td>
<td>4880</td>
<td>1478</td>
<td>45%</td>
<td>8.01</td>
<td>728</td>
</tr>
<tr>
<td>total</td>
<td>38043</td>
<td>29700</td>
<td>6289</td>
<td>2054</td>
<td>83.5%</td>
<td>1.87</td>
<td>728</td>
</tr>
</tbody>
</table>

**Table 2.** Experimental results for state and transition explosion
optimizations in a verified solver is challenging, as more cases need to be proven correct.

### 7.2 The Efforts of Developing CertiStr

In this subsection, we discuss the efforts of developing CertiStr. Table 3 shows the efforts. The rows “Abs Automata Lib” and “Imp Automata Lib” denote abstract level and implementation level automata libraries, respectively. The rows “Abs Forward Prop” and “Imp Forward Prop” denote abstract level and implementation level forward-propagations, respectively. The column “Loc” denotes the lines of Isabelle code for each module. The column “Terms” denotes the number of definitions, functions, locales, and classes. The column “Theorems” denotes the number of theorems and lemmas in a module. The development needs around one person-year efforts.

**Table 3. Development effort of certified string solver**

<table>
<thead>
<tr>
<th>Module</th>
<th>Loc</th>
<th>Terms</th>
<th>Theorems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs Automata Lib</td>
<td>4484</td>
<td>100</td>
<td>254</td>
</tr>
<tr>
<td>Imp Automata Lib</td>
<td>8498</td>
<td>270</td>
<td>203</td>
</tr>
<tr>
<td>Abs Forward Prop</td>
<td>6850</td>
<td>37</td>
<td>39</td>
</tr>
<tr>
<td>Imp Forward Prop</td>
<td>2175</td>
<td>41</td>
<td>18</td>
</tr>
<tr>
<td>Total</td>
<td>22007</td>
<td>448</td>
<td>514</td>
</tr>
</tbody>
</table>

We now discuss the challenges in developing CertiStr. The challenge in the automata library is the usage of sets and labeled automata. Classical NFAs require only the equality comparison for transition labels. But for s-NFAs, we have more operations for labels. For instance, we require the operations `semil`, `intersectionl`, `nemptyl`, `meml` for intervals. These operations make it harder to prove the correctness of the automata library.

The challenge in the forward-propagation module is the usage of sets as transition labels. Classical NFAs require only the equality comparison for transition labels. But for s-NFAs, we have more operations for labels. For instance, we require the operations `semil`, `intersectionl`, `nemptyl`, `meml` for intervals. These operations make it harder to prove the correctness of the automata library.

The challenge in the forward-propagation module is the proofs of Theorem 4.2, 4.3, and 4.4. These theorems need around 4000 lines of code to prove.

### 8 Related Works

**String solvers.** As introduced in Section 1, there already exist various non-certified string solvers, such as Kaluza [35], CVC4 [28], Z3 [17], Z3-str3 [4, 5], Z3-Trau [9], S3P [39], OSTRICH [11–13], SLOTH [22], and Norn [1], among many others. These solvers are intricate and support more string operations than CertiStr. Some of the solvers, such as Z3-str3 and CVC4, opted to support more string operations and settle with incomplete solvers (e.g. with no guarantee of termination) that could still solve many constraints that arise in practice. Other solvers are designed with stronger theoretical guarantees; for instance, OSTRICH is complete for the straight-line string constraint fragment [29]. CertiStr can solve the string constraints that are out of the scope of the straight-line fragment and guarantees to terminate for the constraints without cyclic concatenation dependencies, but it will return *unknown* for constraints that do not satisfy the tree property.

**Symbolic Automata.** CertiStr depends on automata operations for regular expression propagation and consistency checking. An efficient implementation of automata is crucial. In comparison to finite-state automata in the classical sense, symbolic automata [15, 16] have proven to be more appropriate for applications like model checking, natural language processing, and networking. **Certified automata libraries** [7, 26] can provide trustworthiness for automata-based analysis in applications. However, to the best of our knowledge, existing certified automata libraries are based on the classical definition of automata, which makes them inefficient for practical applications with very large alphabets. Our work in this paper contributes to the development of certified symbolic automata libraries.

**Paradigms toward certified constraint solvers.** Two main paradigms have emerged for the verification of constraint solvers: (i) the verification of the solver itself, i.e., the development of solvers with machine-checked end-to-end correctness guarantees. For instance, Shi et al. [36] built a certified SMT quantifier-free bit-vector solver. (ii) The generation of certificates, i.e., the actual solver is not verified, but its outputs are accompanied by proof witnesses that can then be independently checked by a verified, trustworthy certificate checker. For instance, Ekici et al. [19] provide an independent and certified checker for SAT and SMT proof witnesses. In our work, we follow the first paradigm.

**Security applications based on string analysis.** There are various security applications of automata-based analysis and string solvers, such as detecting web security vulnerabilities. Yu et al. [43] investigated the approach to use automata theory for detecting security vulnerabilities, such as XSS, in PHP programs. Their work also depends on a forward analysis. But the forward analysis is over the control flow graphs of PHP programs. CertiStr aims to build a stand-alone string solver and the forward-propagation is only over the string constraints. Many developments of string solvers, string analysis techniques, and automata-theoretic techniques were also directly motivated by security vulnerability detection, e.g., [2, 3, 10, 13, 22, 29, 35, 37–39]. To the best of our knowledge, CertiStr is the first tool that provides certification of a string analysis method that is applicable to security vulnerability detection. We believe there is a need for further development of certified string analyzers: string solving techniques are intricate and hence error-prone, but are applied for security analysis whose correctness is of critical importance.

### 9 Conclusion and Future Works

In this paper, we present CertiStr, a certified string solver for the theory of concatenation and regular constraints. The backend of CertiStr is verified in Isabelle proof assistant, which provides a rigorous guarantee for the results of the
solver. We ran CertiStr over the benchmark Kaluza to show the efficacy of CertiStr.

As future works, firstly, we plan to support more string operations, such as string replacement and capture groups, which are also widely used in programming languages, such as JavaScript and PHP, increasing the applicability of CertiStr. Secondly, the front end of CertiStr still needs to be verified in Isabelle, especially the correctness of the desugaring from string constraints with monadic length functions and disjunctions to the language of CertiStr.

Acknowledgments

The authors would like to thank anonymous reviewers for their valuable comments. This research was supported in part by the ERC Starting Grant 759969 (AV-SMP), Max-Planck Fellowship, Amazon Research Award, the Swedish Research Council (VR) under grant 2018-04727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RTIT17-0011).

References


Workshop on Hammers for Type Theories, HaTT@ETAPS 2016, Coimbra, Portugal, July 1, 2016 (ETAPS, Vol. 210), Janmin Christian Blanchette and Cezary Kaliszyk (Eds.). 21–29. https://doi.org/10.4204/ETAPS.210.5


