Formalizing and verifying an automata-based string calculus within the Carcara proof checker

Vinicius Gomes
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
vinicius.gomes@dcc.ufmg.br

Prof. Dr. Haniel Barbosa
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brasil
hbarbosa@dcc.ufmg.br

***Abstract*—String constraints are crucial in many SMT-based applications, yet solvers that rely on automata-based reasoning, such as OSTRICH, currently lack support for producing independently checkable proofs. This work addresses this limitation by formalizing the core rules of the Regular Constraint Propagation calculus in the Alethe proof format. Furthermore, the Carcara proof checker is extended with verification procedures for these rules, supported by a dedicated automata module. Together, these contributions enable reliable checking of OSTRICH proofs and strengthen the certification infrastructure for automated reasoning with string constraints.**

***Index Terms*—formal methods, string constraints, satisfiability modulo theories, proof checking**

I. Introduction

Satisfiability Modulo Theories (SMT) constitutes a generalization of the Boolean satisfiability problem (SAT) through the incorporation of background theories such as arithmetic, bit-vectors, arrays, and strings [1]. In contrast to SAT solvers, SMT solvers – notably cvc5 [2] and Z3 [3] – are capable of reasoning about richer classes of logical constraints, thus making them central to a wide spectrum of applications, including software verification, interactive and automated theorem proving, and constraint-based analysis. The sustained progress of the field stems from a combination of solver efficiency improvements, theoretical innovations, and the rising need for automated reasoning tools.

An essential feature of SMT solvers is their ability to generate proofs of unsatisfiability, certifying the non-existence of models for a given formula. To ensure interoperability and enable independent verification, standardized proof formats have been introduced. Among these, Alethe [4] stands out as a prominent proposal, supported by solvers such as cvc5 and veriT [5]. By extending the SMT-LIB language [6] natively, Alethe allows solvers to emit proofs without translation overhead, thus simplifying proof production and enabling efficient checking. Proof checkers like Carcara [7] validate these proofs, reinforcing the reliability of SMT solvers in formal verification.

Guaranteeing the correctness of proofs across a wide range of SMT-LIB theories is of particular importance, especially in contexts involving complex reasoning domains. The theory of strings exemplifies this need, given its central role in applications such as security analysis, software validation, vulnerability detection, and bioinformatics [8]–[11]. The development of robust proof formats and proof-checking tools is therefore indispensable for ensuring the reliability of SMT-based reasoning in domains involving string constraints.

In this regard, the objective of this work is twofold: first, to formalize the inference rules of the Regular Constraint Propagation method, originally proposed as an automata-based approach for reasoning about string constraints [12], in the Alethe proof format; and second, to implement their verification within the Carcara proof checker. This extension expands the set of string proofs that Carcara can validate, while also providing support for an automata-based method that may yield more efficient reasoning for certain subclasses of string problems compared to the string calculus of cvc5. Ultimately, this work strengthens the interoperability of both Carcara and the Alethe proof format with different SMT solvers, particularly OSTRICH [13], which has more recently introduced support for Regular Constraint Propagation.

The remainder of this document is structured as follows. Section II provides the theoretical background required to understand the core concepts of this work, including SMT solvers, the theory of strings, and proof checking. Section III presents the research methodology, detailing the formalization of inference rules and their implementation in the Carcara proof checker. Section IV discusses the results and highlights the relevance and impact of the proposed contributions. Finally, Section V offers concluding remarks and outlines potential directions for future research.

II. Theoretical Background

A. String Theory and String Constraints

The theory of strings is fundamental in formal verification and symbolic reasoning tasks involving textual data, particularly in software analysis, automated bug detection, and security verification. Many programs manipulate strings in practice, for instance when parsing user input, constructing URLs, or generating database queries. Accurately modeling and reasoning about such operations is therefore crucial for analyzing system correctness and security. In this setting, string constraints naturally arise as logical conditions on string values that must hold for an execution to be considered correct or safe. They are central to tasks such as input sanitization, detection of injection vulnerabilities, and symbolic execution of programs [14].

String constraints involve operations such as equality, concatenation, substring extraction, prefix and suffix operations, length comparison, and membership in regular languages. This yields a powerful language for modeling the behavior of string-intensive programs. However, the expressiveness of the theory also introduces substantial complexity: while some fragments are decidable, even modest combinations of operations can lead to undecidability. The inclusion of arbitrary functions such as `substring`, `replace`, and `indexOf` further exacerbates this issue, rendering many fragments computationally intractable or undecidable [15]–[17].

The theory of strings has been standardized in SMT-LIB, which provides a common language and ensures interoperability across SMT solvers[1]. This standardization enables solvers to uniformly express and reason about string constraints, facilitating the exchange of problem instances and proofs. SMT-LIB specifies key operators such as $=$, $concat$, $substring$, $length$, and in_re (for regular expression membership), thereby supporting relations including equality, concatenation, substring extraction, length comparison, and pattern matching.

String constraints typically capture relations between string variables, allowing the modeling of complex program behaviors. Common operations include prefix and suffix checks (`str.prefixof`, `str.suffixof`), substring extraction (`str.substr`), length constraints (`str.len`), and membership in regular languages (`str.in_re`).

For example, a constraint may detect potentially malicious input by verifying whether a string variable contains substrings commonly associated with cross-site scripting (XSS) attacks. In SMT-LIB, such a constraint can be expressed as

```
(assert (str.contains userInput
          "<script>"))
```

which asserts that the variable `userInput` contains the substring `"<script>"` – a pattern frequently used in XSS payloads. When combined with other constraints, this formulation enables the modeling and verification of security properties in string-manipulating programs.

B. SMT solvers and OSTRICH

SMT solvers (Satisfiability Modulo Theories solvers) are computational tools designed to decide satisfiability problems that extend propositional logic with additional theories [18]. These solvers can handle formulas involving not only Boolean variables but also theories such as arithmetic (integer and real), bit-vectors, arrays, and strings. Given a formula, an SMT solver determines whether there exists an assignment of values to its variables that satisfies all constraints imposed by the underlying theories. If such an assignment exists, the solver produces a model; otherwise, it reports unsatisfiability.

[1]The definition of the string theory is available at https://smt-lib.org/theories-UnicodeStrings.shtml.

SMT solvers have broad applicability in domains requiring reasoning about complex systems with interacting components. In software verification, they are employed to ensure program correctness, detect bugs, and identify security vulnerabilities by checking whether properties hold under all possible inputs. In hardware design, they assist in verifying that circuit implementations satisfy specified requirements. Beyond verification, SMT solvers support automated theorem proving, constraint solving, and formal analysis, contributing to the correctness of mathematical models and algorithms. They also play a crucial role in security-related tasks, such as analyzing cryptographic protocols and detecting potential exploits in web applications.

At the core of an SMT solver, a SAT engine searches for satisfying assignments over a propositional abstraction of the problem. Many practical problems, however, require reasoning beyond propositional logic; this is addressed by dedicated theory solvers. Each theory solver handles constraints specific to its domain, operating in conjunction with the SAT engine: the SAT engine proposes assignments, and the theory solvers check their consistency with the semantics of the respective theories. If a conflict is detected, the theory solver reports back to the SAT engine, which backtracks and explores alternative assignments. This cooperative process, commonly referred to as the CDCL(τ) architecture [19], enables SMT solvers to efficiently tackle complex problems that combine propositional reasoning with domain-specific constraints.

Several SMT solvers distinguish themselves by performance, supported theories, and specialized features. Among the most prominent are cvc5 and Z3, previously discussed. Additional contributions come from solvers such as MathSAT [20] and SMTInterpol [21]. MathSAT supports advanced functionalities including model generation, interpolation, and optimization, making it particularly suitable for software verification and analysis. SMTInterpol focuses on producing interpolants from unsatisfiability proofs, a capability valuable in model checking and static analysis.

In this work, we focus on OSTRICH [13], a modern SMT solver specifically designed for reasoning about string constraints. OSTRICH adopts an automata-based approach and, more recently, has introduced support for the Regular Constraint Propagation method [12], which provides a distinct calculus for handling string problems. Unlike the algebraic string calculus implemented by solvers such as cvc5, this method repeatedly computes pre- or post-images of regular languages under the string functions present in a string formula, inferring more and more knowledge about the possible values of string variables, offering an alternative and potentially more efficient strategy for certain subclasses of string constraints. Our interest lies particularly in the verification of proofs generated by OSTRICH under this calculus. This entails not only a precise understanding of the Regular Constraint Propagation rules but also the development of mechanisms to ensure that such proofs can be independently and rigorously checked within Carcara.

C. String constraint SMT solvers

The growing importance of the theory of strings across multiple domains has driven the development of SMT solvers specialized in handling string constraints. Unlike general-purpose solvers, these tools employ algorithms and optimizations for string operations such as concatenation, substring extraction, and membership in regular languages. Prominent examples include Z3str4 [22], Z3-Noodler [23], CertiStr [24], OSTRICH [13], Trau [25], as well as cvc5 and Z3.

Each solver may implement one or more distinct string calculi, i.e., specific sets of inference rules, simplification strategies, and solving heuristics for string constraints. For instance, cvc5 employs an algebraic approach to solve quantifier-free constraints natively over unbounded strings, supporting operations such as `str.len` (string length) and membership in regular languages [2]. This approach enables symbolic reasoning about string properties without exhaustive enumeration. In contrast, solvers such as Z3-Noodler and OSTRICH use automata-based techniques, encoding string variables and operations as finite automata to determine satisfiability [13], [23]. Although the high-level strategies are similar, each solver relies on specific algorithms and heuristics, making them more or less suitable depending on the characteristics of the constraints. This diversity illustrates how different fragments of the theory of strings benefit from tailored solving techniques.

Employing a specialized solver for string constraints offers several advantages, particularly given the complexity of string operations. Such solvers often implement calculi optimized for specific fragments of the theory, enabling more efficient decision procedures and improved performance on benchmarks with complex string constraints. Their targeted design also enhances scalability, reduces solving time, and improves precision when reasoning about program behaviors involving intricate string manipulations, including input sanitization and security validation.

D. Regular Constraint Propagation

The Regular Constraint Propagation (RCP) method constitutes an alternative approach to the traditional technique of equation splitting in string constraint solving. Rather than decomposing word equations into smaller components, RCP operates by systematically propagating regular constraints across the functional structure of string terms. This is achieved through the computation of pre-images and post-images of regular languages under string functions such as concatenation, substring extraction, `replaceAll`, and transductions. In practice, the method iteratively refines the possible values of string variables by pushing constraints forward from function arguments to their outputs, and backward from function results to their arguments.

The propagation process is formalized as a collection of inference rules that govern the interaction between string operations and regular languages. Typical examples include rules for closing constraints under logical consequence, intersecting automata to combine restrictions, forward propagation rules that restrict the result of a function given constraints on its arguments, and backward propagation rules that infer constraints on the arguments from restrictions on the result. The application of these rules is orchestrated by a so-called Marking Algorithm, which ensures termination and completeness for a class of constraints known as orderable constraints. Through this mechanism, RCP provides a systematic framework for deriving contradictions, which certify unsatisfiability, or for establishing satisfiability by progressively narrowing the search space.

OSTRICH incorporates RCP within its automata-based architecture for string solving. Originally, OSTRICH relied on equation splitting and backward propagation to reason about string constraints. With the integration of RCP, the solver extends its reasoning capabilities by allowing regular constraints to be propagated uniformly through the functional structure of string formulas. This integration enables OSTRICH to apply automata-theoretic operations, such as intersection and emptiness checks, in a more targeted and efficient manner. Experimental results have shown that the RCP-enhanced version of OSTRICH achieves significant performance improvements, not only matching but in some cases outperforming state-of-the-art solvers on challenging benchmarks, including instances derived from the Post Correspondence Problem and applications in bioinformatics.

E. Proof production in SMT solvers

Proof generation in SAT solvers is essential for certifying unsatisfiability results. When a solver establishes that a formula has no satisfying assignment, it can output a proof that details, step by step, how this conclusion was obtained. In Conflict-Driven Clause Learning (CDCL) solvers, this process involves deriving new clauses through resolution. Each learned clause is a logical consequence of the original set, typically introduced by resolving conflicts during search. To construct the proof, the solver records the derivation path of learned clauses, specifying both the initial clauses involved and the intermediate resolutions. The procedure ultimately culminates in the derivation of the empty clause ($\perp$), which serves as a certificate of unsatisfiability [26].

Early SAT solvers employed formats such as DRUP (Delete Reverse Unit Propagation) [27] and DRAT (Delete Resolution Asymmetric Tautology) [28] for proof representation. In DRUP, each new clause is validated through reverse unit propagation, offering a straightforward but limited verification method. DRAT extended this approach by admitting asymmetric tautologies, thereby increasing its expressiveness. However, DRAT proofs omit explicit resolution information, forcing external checkers to reconstruct derivations. This can be computationally demanding and complicates formal verification.

To address these limitations, the LRAT (Labeled Resolution Asymmetric Tautology) format was introduced [29]. Unlike DRAT, LRAT annotations explicitly record the sequence of resolution steps leading to each derived clause. This makes proof checking significantly simpler and enables the development of formally verified checkers within theorem provers such as Rocq[2] . Although DRAT remains the standard in SAT competitions, several solvers, including CaDiCaL [30], already support LRAT and can generate detailed proofs natively.

In SMT solving, proof generation generalizes clause-based reasoning by incorporating inference steps from background theories. Whereas SAT solvers rely exclusively on propositional resolution, SMT solvers must also justify lemmas derived from theories such as arithmetic, arrays, or strings. Whenever a theory solver produces a conclusion that is used to learn a clause, this step must be annotated with a theory-specific justification according to a formal calculus. These annotations are accumulated during the solving process to form a complete proof of unsatisfiability that integrates both Boolean and theory-level reasoning [2], [6].

Modern SMT solvers, such as cvc5, achieve this through a layered proof architecture. Boolean reasoning is managed by a CDCL engine, while each theory solver generates local proof objects to explain its conclusions [2]. Proof generation results in a DAG of inferences, where theory lemmas are justified by theory-specific rules and combined with SAT-level derivations until the empty clause is reached. This integration requires a proof format capable of handling heterogeneous reasoning steps while maintaining a coherent structure.

These aspects illustrate the inherent complexity of proof generation in SMT solvers, which must balance multiple theories and different granularities of reasoning. Proof checkers play a crucial role in managing this complexity: by validating coarse-grained steps or elaborating them into more detailed derivations, they strengthen the guarantees of solver correctness without imposing excessive overhead. As a result, proof checkers are indispensable tools for ensuring trust in SMT-based verification workflows.

F. Alethe proof format

Alethe was designed to be a standardized and easily producible format for SMT solvers [31]. This is achieved by directly extending the term language of the SMT-LIB standard, removing the need for solvers to translate proofs between different languages before outputting them.

A proof format that directly extends SMT-LIB offers significant advantages over existing formats, like LFSC and DOT. While LFSC and DOT require solvers to translate terms into a separate language with different syntax and semantics, Alethe leverages the existing SMT-LIB term language, which is already widely adopted by the SMT community. This alignment reduces the implementation effort for solver developers, minimizes the risk of translation errors, and makes the resulting proofs easier to read and debug. Additionally, since SMT-LIB is a common ground for expressing problems across various theories, extending it for proof generation promotes greater consistency and interoperability within the SMT ecosystem.

From this point on, we briefly introduce the structure of the Alethe proof format, providing an overview of how proofs are represented. This section was inspired by a section with the same purpose in [7]. For a complete specification of its syntax and semantics, the reader is referred to [4]. Alethe proofs are constructed as refutations, proceeding through a series of inference steps until $\bot$ is derived, indicating that the original problem instance has been refuted – in other words, $\pi : \varphi_1 \land \varphi_2 \land \cdots \land \varphi_n \to \bot$. Proofs are composed of multiple steps, each represented as an indexed `step` in the proof. The `assume` command is syntactically similar to a `step` but is used exclusively to introduce the original problem's assumptions. Together, the collection of steps forms a directed acyclic graph (DAG), where the root corresponds to the step that derives $\bot$, and the leaves represent the initial assumptions. Steps represent the application of inference rules and have the following structure

$$c_1, \dots, c_k \ \triangleright \ i.\ \psi_1, \dots, \psi_l \ (\texttt{rule}\ p_1, \dots, p_n)\ [a_1, \dots, a_m]$$

In this structure, `rule` identifies the inference rule applied at step i, which derives a clause consisting of the literals $\psi_1, \dots, \psi_l$. The list of premises, $p_1, \dots, p_n$, refers to the identifiers of previously introduced steps or assumptions. Inference rules can also be parameterized by a list of arguments, $a_1, \dots, a_m$, which provide additional information required for the application of the rule. Furthermore, steps may occur within a specific context, defined by bound variables or substitutions $c_1, \dots, c_k$, introduced using the `anchor` command.

G. Carcara proof checker

Carcara is an open-source[3] independent proof checker for Alethe proofs, implemented in a high-performance programming language, Rust [7]. Proof checking is performed through a collection of modules, each implementing the appropriate verification for the rule being checked. Carcara also provides methods for proof elaboration, allowing coarse-grained steps to be translated into a set of fine-grained steps that are easier to check.

Carcara takes as input an SMT-LIB problem φ and an Alethe proof $\pi : \varphi \to \bot$. While checking a proof, it first parses booth of the inputs. The problem provides the declarations of sorts and symbols that may appear in the proof, as well as the set of assertions, which must match the assumptions in the proof. Each command in the proof is then individually checked by the rule checker corresponding to the rule applied in that command. The checker takes as input the premises, the conclusion, the arguments, and the context in which the command appears, and checks whether the given conclusion can indeed be derived from the application of that rule. Then, it prints `valid` when all steps have been successfully verified, `holey` when π is valid but contains unchecked steps ("holes" in the proof), or `invalid` along with an error message indicating the first step where verification failed and the reason for the failure.

Additionally, as previously mentioned, Carcara is also capable of elaborating proofs into more fine-grained versions. While elaborating a proof, it converts π into $\pi' : \varphi \to \bot$,

[2]Rocq is available at https://rocq-prover.org/.

[3]Carcara is available at https://github.com/ufmg-smite/carcara.

where some steps are replaced by a set of finer-grained steps, and then outputs the elaborated proof π'. This is especially useful for mitigating bottlenecks when checking coarse-grained Alethe proof steps, as it fills in missing details from the original proof and produces versions that are easier to verify. At the end of the elaboration process, the resulting proof is equivalent to the original one. Formally, a coarse-grained proof concluding ψ from premises $\psi_1, \ldots, \psi_n$ is elaborated into a proof π that has $\psi_1, \ldots, \psi_n$ as its leaves and derives the same conclusion ψ at the end. This expansion affects the proof only locally, as any step that originally depended on the conclusion of the coarse-grained proof can now use the conclusion of the elaborated proof interchangeably. Thus, the elaboration preserves the logical structure and correctness of the original argument while making it more suitable for fine-grained verification.

III. Research Methodology

The methodology of this work was structured into three main stages that guided the development process. The first stage consisted in developing a thorough understanding of the inference steps defined by the Regular Constraint Propagation calculus and formalizing them in the Alethe proof format. Each rule was analyzed in detail, its premises and conclusions were identified, and its semantics were captured in a precise form suitable for later verification within Carcara. This included all the rules already defined in [12] and the additional correspondence rule needed to relate SMT-LIB regular expressions to their automaton representations.

The second stage focused on extending Carcara's parsing infrastructure. The parser was augmented to handle the new operators introduced by the calculus, including the `re.from_automaton` operator, which embeds an automaton specification within proof terms. A dedicated parser was implemented to interpret these automaton singletons by extracting their states, initial and accepting states, and transitions, and converting them into Carcara's internal data structures. This extension enabled the tool to properly reconstruct the automata required by the verification procedures.

The final stage involved defining and implementing a robust automaton representation within Carcara. To maintain a small *trusted base*, the automata-related data structures and algorithms were implemented from scratch rather than relying on external libraries. This representation supports deterministic and nondeterministic automata, provides conversion from NFA to DFA, and includes fundamental operations such as intersection, emptiness checking, and equivalence checking. With this module in place, the verification procedures for each inference rule were implemented by combining the parsed proof information with the operations defined over automata.

IV. Results

The following subsections present the results obtained during the development of this work. Subsection IV-A presents the formalization of the key proof rules of the Regular Constraint Propagation calculus in the Alethe proof format[4], emphasizing the adaptations and structural adjustments required to ensure compatibility. This is followed, in Subsection IV-B, by a description of the implementation of these rules within the Carcara proof checker and the development of the automata module that supports their verification[5]. Finally, Subsection IV-C provides a concrete example that illustrates the verification process on a representative instance.

A. Theory formalization in Alethe

In this work, all four rules defined in [12] – `Close`, `Intersect`, `Forward Propagation`, and `Backward Propagation` – for the Regular Constraint Propagation calculus were fully formalized. In addition to these, a fifth rule was introduced to establish a correspondence between a regular expression defined using the SMT-LIB string operators and its representation as an automaton – the `re_convert` rule. This rule supports the verification process by ensuring that conclusions derived from the automaton-based representation also hold for the regular expression presented in the alternative syntax. This is particularly important because, in typical problem settings, the desired reasoning outcomes concern the regular expressions themselves rather than their automaton encodings.

Although the inference rules could be defined, several structural adjustments were required to map them faithfully into the Alethe proof format. In particular, the `Close` rule was mapped to the Alethe rule `re_empty_intersection`, which does not conclude an empty clause directly. Instead, it produces a set of terms from which resolution steps may be applied later, eventually leading to a conflict. Apart from this necessary adaptation, the remaining mappings follow the definitions from [12] quite closely.

In the end, the complete set of formalized rules consists of `re_convert`, `re_empty_intersection`, `re_intersection`, `re_forward_prop`, and `re_backward_prop`. Additional rules may be incorporated in the future as OSTRICH's proof production pipeline evolves, but this set has so far proven sufficient for solving, producing proofs for, and verifying instances that rely on this calculus.

B. Theory verification and the automata module

1) Automata module: To enable the correct verification of each inference rule defined in this work, a dedicated module implementing an automaton representation and its core operations had to be developed. Although several Rust libraries for automata manipulation exist, we opted for a custom implementation in order to reduce the *trusted base*. This approach gives us full control over the data structures and algorithms involved, allowing us to extend or modify the module as needed to support the verification of the calculus,

[4]Rule formalization is available at https://gitlab.uliege.be/verit/alethe/tree/regular-constraint-propagation.

[5]Rule verification and the automata module are available at https://github.com/vinisilvag/carcara/tree/regular-constraint-propagation.

and avoiding potential limitations or constraints imposed by existing libraries.

The module defines a complete automaton structure (Automaton) whose main component is a vector of State elements, representing all states in the automaton. Each State stores, among other information, whether it is accepting and a HashSet of its outgoing transitions to other states. These transitions are defined over ranges of integers between 0 and 255, corresponding to ASCII character codes, matching the internal representation used by OSTRICH. In addition to the data structures, the module implements the core operations required during verification. These include computing the intersection between automata, checking automaton equivalence, checking whether an automaton is empty (e.g., having no accepting states or having no accepting states reachable from the initial state), and converting an NFA into a DFA, among others. An important design decision was that all verification procedures operate exclusively on DFAs so that more efficient, well-studied algorithms from the literature can be applied. This decision motivated the implementation of the NFA-to-DFA conversion algorithm: whenever an NFA is constructed, it is immediately determinized to ensure that subsequent procedures work with a DFA.

The module also defines additional operations that, although not strictly automaton operations, are essential for the verification process. The most significant of these is the construction of automata from regular expressions written using the SMT-LIB string operators. In this case, the AST of the regular expression is traversed, and an equivalent automaton is constructed. Initially, this process produces an NFA; however, the determinization procedure is invoked before returning the result, guaranteeing that all automata used in the verification procedures are DFAs.

2) Verification procedures: To enable the verification of the inference rules, dedicated verification procedures were implemented in Carcara for each rule. However, for this to be possible, modifications beyond the automata module were required. In particular, Carcara's parser had to be extended to support the new operator introduced by the calculus: re.from_automaton, an operator that takes as input a string singleton encoding an automaton, including its states, transitions, initial state, and accepting states.

In addition to extending the parser to recognize this new operator, a parser for the automaton-encoding string singleton itself had to be developed. This parser was implemented using a Rust parser-combinator library and, once executed, returns an Automaton object. Instead of returning the operator directly after the automaton is parsed, the system produces a Constant::RegLan term containing both the original regular-expression operator and its automaton representation. Other RegLan terms that do not have an associated automaton simply store None in this field.

With these components in place, the verification procedures could be implemented. Each procedure checks the number and structure of premises and conclusions, ensuring that they match the expected shape of the rule. If these structural checks succeed, the semantic verification is performed by comparing the result returned by the solver with the expected result according to the rule's definition. In most cases, verification involves receiving automata directly from the premises and performing operations on them, or constructing automata from regular expressions and then applying the necessary computations.

For example, the re_convert rule is verified by constructing an automaton from the regular expression provided in the premise and checking whether it is equivalent to the automaton encoded in the accompanying term. The re_empty_intersection rule is verified by computing the intersection of the automata in the premises and checking whether any accepting state is reachable in the resulting automaton. The re_forward_prop rule is verified by constructing an automaton corresponding to the operation applied in the regular-expression premise (for instance, computing the concatenation of the automata associated with its operands) and checking its equivalence with the automaton appearing in the rule's conclusion. The remaining rules follow similar patterns.

C. Verification example

In this subsection, we present an example problem and illustrate how the implemented rules appear in the proof. Most importantly, we detail how the verification of these rules is carried out by Carcara.

Let Listing 1 be an example SMT-LIB input provided to OSTRICH. Since the instance is unsatisfiable, the proof must contain the reasoning steps performed by the solver that eventually lead to a contradiction (see Appendix A for the complete proof generated by OSTRICH). In this case the problem constraints require that a string w belongs to the regular language denoted by a^+ and, at the same time, belongs to either the language denoted by b^+ or by c^+. The proof therefore progresses toward deriving a contradiction from these two assertions. If it can be established that the intersection between these regular languages is empty, then no word w can satisfy both assertions simultaneously. This is precisely the direction taken by the solver to reach the contradiction.

```
(set-logic QF_S)

(declare-const w String)

(assert (str.in_re w (re.+ (str.to_re "a"))))
(assert (or (str.in_re w
                (re.+ (str.to_re "b")))
            (str.in_re w
                (re.+ (str.to_re "c")))))

(check-sat)
```

Listing 1. SMT-LIB input.

Accordingly, the proof contains steps that establish the correspondence between each regular expression and its automaton representation and then check the intersection of the corresponding automata. Because the intersections are empty,

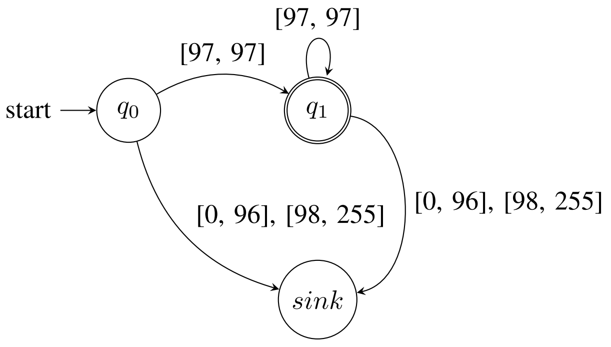

Fig. 1. Automaton for a^+.

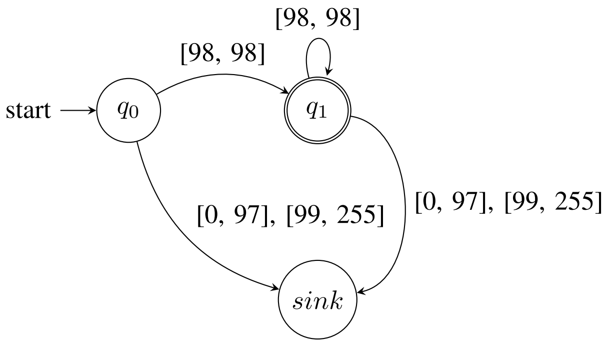

Fig. 2. Automaton for b^+.

the solver is able to apply resolution steps and ultimately derive the empty clause. Internally in Carcara, the application of `re_convert` is checked by constructing an automaton from the regular expression in the premise and comparing its equivalence with the automaton provided in the conclusion. Similarly, the `re_empty_intersection` check computes the intersection of the automata and verifies that it is empty.

For instance, in steps `t1` and `t3.s1`, the correspondence between each regular expression and its automaton is established for the languages defined by a^+ and b^+, respectively, through applications of `re_convert`. At this point, the automata 1 and 2 are introduced via the singleton that represents them in the proof. Note that this singleton describes an NFA, not a DFA; the previously shown automaton representation already includes the NFA-to-DFA conversion.

After that, in step `t3.s3`, the rule `re_empty_intersection` is applied, receiving the two automata as arguments and determining that their intersection is empty. Automaton 3 (the one produced by this step) encodes the result of the intersection. As can be observed, it has no reachable accepting states and therefore the intersection is empty. With this information, and after performing a few resolution steps, the empty clause can be derived for this branch at step `t3.s4`. Thus, we conclude that the intersection of a^+ and b^+ is empty and, consequently, the string w cannot belong to both a^+ and b^+ simultaneously. This leaves us with the task of checking whether w could belong to a^+ and c^+.

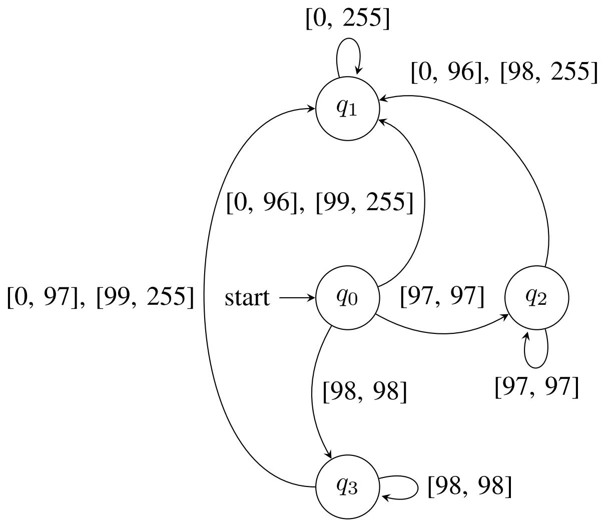

Fig. 3. Automaton for the intersection between a^+ and b^+.

An analogous line of reasoning is carried out for the second case, and once again the resulting intersection is empty. Together with these steps and the premises of the proof, a resolution step at `t8` derives the empty clause, thereby completing the proof.

The appendices provide the full implementations of the specific verification procedures used in the example discussed in this section. Appendix B includes the complete code for the verification of `re_convert`, and Appendix C contains the full implementation of the procedure for `re_empty_intersection`.

V. Conclusion

This work presented the formalization of the proof calculus for Regular Constraint Propagation as implemented in the OSTRICH solver. This effort involved the formal specification of the inference rules and the development of dedicated verification procedures for each of them within the Carcara proof checker. In order to enable the verification of these rules, we also implemented a module defining the structure of automata and the core operations used to manipulate them, allowing the calculus to be faithfully represented and checked within Carcara's internal framework.

As future work, we plan to conduct a thorough benchmarking process once OSTRICH is fully instrumented to produce Alethe-formatted proofs for string constraints. This will allow us to evaluate both the correctness and the performance of the verification procedures. Such an evaluation is particularly important for large inputs, where operations such as the conversion from NFAs to DFAs may lead to state explosion and significantly impact performance. These steps will help establish a more robust and certified infrastructure for the verification of automata-based reasoning in the domain of string constraints.

References

[1] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability, Second Edition* (A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, eds.), vol. 336 of *Frontiers in Artificial Intelligence and Applications*, ch. 33, pp. 825–885, IOS Press, Feb. 2021.

[2] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (D. Fisman and G. Rosu, eds.), vol. 13243 of *Lecture Notes in Computer Science*, pp. 415–442, Springer, 2022.

[3] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (C. R. Ramakrishnan and J. Rehof, eds.), vol. 4963 of *Lecture Notes in Computer Science*, pp. 337–340, Springer, 2008.

[4] H. Barbosa, M. Fleury, P. Fontaine, and H.-J. Schurr, "The Alethe Proof Format." https://verit.loria.fr/documentation/alethe-spec.pdf, 2023. Acessed on 04-03-2025.

[5] T. Bouton, D. C. B. D. Oliveira, D. Déharbe, and P. Fontaine, "veriT: An Open, Trustable and Efficient SMT-Solver," in *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings* (R. A. Schmidt, ed.), vol. 5663 of *Lecture Notes in Computer Science*, pp. 151–156, Springer, 2009.

[6] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)." https://smt-lib.org/, 2016. Acessed on 04-03-2025.

[7] B. Andreotti, H. Lachnitt, and H. Barbosa, "Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format," in *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I* (S. Sankaranarayanan and N. Sharygina, eds.), vol. 13993 of *Lecture Notes in Computer Science*, pp. 367–386, Springer, 2023.

[8] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Trau: SMT solver for string constraints," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018* (N. S. Bjørner and A. Gurfinkel, eds.), pp. 1–5, IEEE, 2018.

[9] R. Amadini, M. Andrlon, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Constraint programming for dynamic symbolic execution of javascript," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings* (L. Rousseau and K. Stergiou, eds.), vol. 11494 of *Lecture Notes in Computer Science*, pp. 1–19, Springer, 2019.

[10] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, and C. Zhang, "Combining string abstract domains for javascript analysis: An evaluation," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I* (A. Legay and T. Margaria, eds.), vol. 10205 of *Lecture Notes in Computer Science*, pp. 41–57, 2017.

[11] P. Barahona and L. Krippahl, "Constraint programming in structural bioinformatics," *Constraints An Int. J.*, vol. 13, no. 1-2, pp. 3–20, 2008.

[12] M. Hague, A. Jeż, A. W. Lin, O. Markgraf, and P. Rümmer, "The power of regular constraint propagation (technical report)," *arXiv preprint arXiv:2508.19888*, vol. abs/2508.19888, Aug. 2025.

[13] T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu, "Decision procedures for path feasibility of string-manipulating programs with complex operations," *CoRR*, vol. abs/1811.03167, 2018.

[14] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, and C. W. Barrett, "A decision procedure for regular membership and length constraints over unbounded strings," in *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings* (C. Lutz and S. Ranise, eds.), vol. 9322 of *Lecture Notes in Computer Science*, pp. 135–150, Springer, 2015.

[15] T. Chen, Y. Chen, M. Hague, A. W. Lin, and Z. Wu, "What is decidable about string constraints with the replaceall function," *CoRR*, vol. abs/1711.03363, 2017.

[16] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. W. Barrett, and M. Deters, "An efficient SMT solver for string constraints," *Formal Methods Syst. Des.*, vol. 48, no. 3, pp. 206–234, 2016.

[17] R. Amadini, "A survey on string constraint solving," *CoRR*, vol. abs/2002.02376, 2020.

[18] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability - Second Edition* (A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds.), vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 1267–1329, IOS Press, 2021.

[19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006.

[20] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (N. Piterman and S. A. Smolka, eds.), vol. 7795 of *Lecture Notes in Computer Science*, pp. 93–107, Springer, 2013.

[21] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: An interpolating SMT solver," in *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings* (A. F. Donaldson and D. Parker, eds.), vol. 7385 of *Lecture Notes in Computer Science*, pp. 248–254, Springer, 2012.

[22] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh, "Z3str4: A multi-armed string solver," in *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings* (M. Huisman, C. S. Pasareanu, and N. Zhan, eds.), vol. 13047 of *Lecture Notes in Computer Science*, pp. 389–406, Springer, 2021.

[23] Y. Chen, D. Chocholatý, V. Havlena, L. Holík, O. Lengál, and J. Síc, "Z3-noodler: An automata-based string solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I* (B. Finkbeiner and L. Kovács, eds.), vol. 14570 of *Lecture Notes in Computer Science*, pp. 24–33, Springer, 2024.

[24] S. Kan, A. W. Lin, P. Rümmer, and M. Schrader, "Certistr: A certified string solver (technical report)," *CoRR*, vol. abs/2112.06039, 2021.

[25] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Trau: SMT solver for string constraints," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018* (N. S. Bjørner and A. Gurfinkel, eds.), pp. 1–5, IEEE, 2018.

[26] M. J. H. Heule, "Proofs of unsatisfiability," in *Handbook of Satisfiability - Second Edition* (A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds.), vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 635–668, IOS Press, 2021.

[27] M. Heule, W. A. H. Jr., and N. Wetzler, "Trimming while checking clausal proofs," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 181–188, IEEE, 2013.

[28] M. J. H. Heule, "The DRAT format and drat-trim checker," *CoRR*, vol. abs/1610.06229, 2016.

[29] L. Cruz-Filipe, M. Heule, W. A. H. Jr., M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," *CoRR*, vol. abs/1612.02353, 2016.

[30] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "Cadical 2.0," in *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I* (A. Gurfinkel and V. Ganesh, eds.), vol. 14681 of *Lecture Notes in Computer Science*, pp. 133–152, Springer, 2024.

[31] F. Besson, P. Fontaine, and L. Théry, "A flexible proof format for SMT: a proposal," in *PxTP 2011: First International Workshop on Proof eXchange for Theorem Proving, Wrocław, Poland, August 1, 2011* (P. Fontaine and A. Stump, eds.), pp. 15–26, 2011.

APPENDIX A
PROOF OUTPUT

```
(assume input1 (str.in_re w (re.+ (str.to_re "a"))))
(assume input2 (or (str.in_re w (re.+ (str.to_re "b")))
                   (str.in_re w (re.+ (str.to_re "c")))))

(step t1 (cl
    (not (str.in_re w (re.+ (str.to_re "a"))))
    (str.in_re w
        (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [97, 97]; s1 -> s1 [97, 97];
        accepting s1;};"))
) :rule re_convert)

(step t1.a (cl
    (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [97, 97]; s1 -> s1
    [97, 97]; accepting s1;};"))
) :rule resolution :premises (input1 t1))

(step t2 (cl (str.in_re w (re.+ (str.to_re "b")))
             (str.in_re w (re.+ (str.to_re "c"))))
    :rule or :premises (input2))

; start proof branch that spans until t3
(anchor :step t3)
(assume t3.input (str.in_re w (re.+ (str.to_re "b"))))

(step t3.s1 (cl
    (not (str.in_re w (re.+ (str.to_re "b"))))
    (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [98, 98]; s1 -> s1
    [98, 98]; accepting s1;};"))
) :rule re_convert)
(step t3.s2 (cl
    (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [98, 98]; s1 -> s1
    [98, 98]; accepting s1;};"))
) :rule resolution :premises (t3.input t3.s1))

(step t3.s3 (cl
    (not (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [98, 98]; s1 ->
     s1 [98, 98]; accepting s1;};")))
    (not (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [97, 97]; s1 ->
     s1 [97, 97]; accepting s1;};")))
) :rule re_empty_intersection)
(step t3.s4 (cl) :rule resolution :premises (t1.a t3.s2 t3.s3))

(step t3 (cl (not (str.in_re w (re.+ (str.to_re "b")))) false) :rule subproof :discharge (
   t3.input))
(step t3.tmp0 (cl (not false)) :rule false)
(step t3.tmp1 (cl (not (str.in_re w (re.+ (str.to_re "b"))))) :rule resolution :premises (t3
   t3.tmp0))

(step t4 (cl (str.in_re w (re.+ (str.to_re "c"))))
      :rule resolution :premises (t2 t3.tmp1))

(step t5 (cl
    (not (str.in_re w (re.+ (str.to_re "c"))))
    (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [99, 99]; s1 -> s1
    [99, 99]; accepting s1;};"))
):rule re_convert)
(step t6 (cl
    (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [99, 99]; s1 -> s1
    [99, 99]; accepting s1;};"))
):rule resolution :premises (t4 t5))
```

```
(step t7 (cl
    (not (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [99, 99]; s1 ->
     s1 [99, 99]; accepting s1;};")))
    (not (str.in_re w (re.from_automaton "automaton value_0 { init s0; s0 -> s1 [97, 97]; s1 ->
     s1 [97, 97]; accepting s1;};")))
):rule re_empty_intersection)

(step t8 (cl) :rule resolution :premises (t1.a t6 t7))
```

Appendix B

RE_CONVERT CODE

```
pub fn re_convert(RuleArgs { conclusion, pool, .. }: RuleArgs) -> RuleResult {
    let (w1, a1) = match_term_err!((not (strinre w a1)) = &conclusion[0])?;
    let (w2, a2) = match_term_err!((strinre w a2) = &conclusion[1])?;

    assert_eq(w1, w2)?;

    let a1 = Automata::create_from_regex_operators(pool, a1)?.nfa_to_dfa();
    let a2 = a2.as_automata_err()?;

    if !operations::is_equivalent(a1.clone(), a2.clone()) {
        return Err(CheckerError::ExpectedAutomatasToBeEquivalent(a1, a2));
    }

    Ok(())
}
```

Appendix C

RE_EMPTY_INTERSECTION CODE

```
pub fn re_empty_intersection(RuleArgs { conclusion, .. }: RuleArgs) -> RuleResult {
    let (w1, a1) = match_term_err!((not (strinre w a1)) = &conclusion[0])?;
    let (w2, a2) = match_term_err!((not (strinre w a2)) = &conclusion[1])?;

    assert_eq(w1, w2)?;

    let a1 = a1.as_automata_err()?;
    let a2 = a2.as_automata_err()?;
    let intersection = operations::intersection(a1.clone(), a2.clone());

    if has_reachable_accepting_state(intersection.clone()) {
        return Err(CheckerError::ExpectedAutomataEmptyIntersection(
            intersection,
            a1,
            a2,
        ));
    }

    Ok(())
}
```