

HornStr: Invariant Synthesis for Regular Model Checking as Constrained Horn Clauses

Hongjian Jiang¹, Anthony W. Lin^{1,2}, Oliver Markgraf¹, Philipp Rümmer^{3,4}, and Daniel Stan^{5,6}



- ¹ University of Kaiserslautern-Landau, Kaiserslautern Germany,
² Max Planck Institute for Software Systems, Kaiserslautern, Germany
³ University of Regensburg, Regensburg, Germany
⁴ Uppsala University, Uppsala, Sweden

⁵ EPITA, Laboratoire de Recherche de l'EPITA (LRE), 14-16 Rue Voltaire, 94270 Le Kremlin Bicêtre, France

⁶ Université de Strasbourg, CNRS, ICube UMR7357, F-67000 Strasbourg, France



Abstract. We present **HornStr**, the first solver for invariant synthesis for Regular Model Checking (RMC) with the specification provided in the SMT-LIB 2.6 theory of strings. It is well-known that invariant synthesis for RMC subsumes various important verification problems, including safety verification for parameterized systems. To achieve a simple and standardized file format, we treat the invariant synthesis problem as a problem of solving Constrained Horn Clauses (CHCs) over strings. Two strategies for synthesizing invariants in terms of regular constraints are supported: (1) L* automata learning, and (2) SAT-based automata learning. **HornStr** implements these strategies with the help of existing SMT solvers for strings, which are interfaced through SMT-LIB. **HornStr** provides an easy-to-use interface for string solver developers to apply their techniques to verification. At the same time, it allows verification researchers to painlessly tap into the wealth of modern string solving techniques. To assess the effectiveness of **HornStr**, we conducted a comprehensive evaluation using benchmarks derived from applications including parameterized verification and string rewriting tasks. Our experiments highlight **HornStr**'s capacity to effectively handle these benchmarks, e.g., as the first solver to verify the challenging MU puzzle automatically. Finally, **HornStr** can be used to automatically generate a new class of interesting SMT-LIB 2.6 string constraint benchmarks, which might in the future be used in the SMT-COMP strings track. In particular, our experiments on the above invariant synthesis benchmarks produce more than 30000 new **QF_S** constraints. We also detail the performance of various integrated string solvers, providing insights into their effectiveness on our new benchmarks.

1 Introduction

Regular Model Checking (RMC) [1,53,32,10,39] is a prominent framework for modeling an infinite-state transition system as a string rewrite system. Classically, the transition relation is specified as a length-preserving transducer. It is

well-known that RMC can be used to model a variety of systems, most notably *parameterized systems*, i.e., distributed protocols with an arbitrary number of processes. Many RMC tools have been developed, focusing on safety verification, e.g., [52,4,3,10,53,9,45,1,35,15,43], to name a few.

Despite the amount of work on RMC in the past decades and the potential of RMC in addressing highly impactful verification problems, RMC tools are typically cumbersome to use. The first problem is the need for the user to specify the model in a low-level language, usually in terms of transducers. The second problem is the absence of a standard file format agreed upon by RMC tool developers. Perhaps this is one main reason that most RMC tools attracted very few users and are mostly no longer maintained today.

SMT-LIB 2.6 Theory of Strings. String constraints have been standardized as part of SMT-LIB 2.6 since 2020, enabling the organization of a track for string solvers at the annual SMT-COMP. The theory over strings has since attracted significant interest in academia [12,22,29] and industry [47,40,6]. The theory provides rich support for string operators (concatenation, replace-all, regular constraints, length constraints, etc.), allowing one to conveniently express operations performed in string-manipulating programs in a high-level language like JavaScript. Out of the many existing string solvers [2,44,33,48,50,54,51,31], at least five solvers (Z3 [16], Z3-alpha [41], Z3-noodler [14], cvc5 [7], and OSTRICH [13]) now support the SMT-LIB 2.6 format.

RMC meets String Solvers. In this paper, we propose to connect RMC with string solvers. Our goal is to provide an easy-to-use and *unified* interface: (i) for string solver developers to apply their techniques to verification, and (ii) for verification researchers/users who could benefit from RMC and parameterized verification to easily tap into the wealth of modern string-solving techniques. To this end, we propose to *treat invariant synthesis for RMC as a sub-problem of Constrained Horn Clauses (CHCs) over the theory of strings*. CHCs [8,20] form a fragment of first-order logic over background theories that serves as an intermediate language for expressing safety verification problems. A CHC formulation of RMC benefits from the *standard and familiar SMT-LIB specification language*. Before our work, no existing CHC solvers directly supported the theory of strings.

Our **first contribution** is, therefore, to develop the first solver **HornStr** for invariant synthesis for RMC expressed as a CHC problem over strings. Our solver **HornStr** supports two strategies for synthesizing invariants in terms of regular constraints: (1) L* automata learning [5], and (2) SAT-based automata learning [23]. Both solvers interact with a string solver via *equivalence queries*, which ask the string solver to verify whether an invariant candidate is correct. The first strategy also interacts with the string solver via *membership queries*, which check whether a guessed string is contained in all invariants. To handle both kinds of queries, **HornStr** uses other string solvers as backends through the SMT-LIB 2.6 interface. Note that similar strategies were already used in other RMC tools [15,45], where these queries were answered by interacting with an *ad-hoc* automata implementation, in contrast to string solvers, which are con-

tinuously being improved. To assess the effectiveness of HornStr, we conducted a comprehensive evaluation using benchmarks derived from applications, including parameterized verification and string rewriting tasks, integrating the available string solvers individually and in combination. Our experiments highlight HornStr’s ability to effectively handle these benchmarks, e.g., as the first solver to verify the challenging MU puzzle automatically.

As a by-product of our tool development, our **second contribution** is the generation of a new class of **QF_S** constraints, which could be used in future SMT-COMP competitions for string solvers. These constraints differ from most benchmarks currently available in SMT-LIB, as they are derived from an invariant synthesis problem. In contrast, the majority of existing benchmarks stem from symbolic execution (like, e.g., the PyEx family). We have evaluated available string solvers on these benchmarks and report the results in this paper.

2 Constraint Horn Clauses

We describe in this section the CHC formalism used as input format by HornStr, as well as examples of applications, illustrating the relationship with RMC and string-rewrite systems.

Definition 1. *A Constrained Horn Clause (CHC) is a first-order logic formula of the form*

$$\forall \mathcal{X}. \varphi \wedge p_1(T_1) \wedge \cdots \wedge p_k(T_k) \rightarrow \psi, \quad (k \geq 0),$$

in which the term ψ is either an uninterpreted predicate $h(T)$ or \perp , and p_1, \dots, p_k are uninterpreted predicates. The set of variables \mathcal{X} contains all variables from $T \cup \bigcup_{i=1}^k T_i$. The formula φ represents a constraint in the background theory, such as linear arithmetic or strings.

A CHC system is a conjunction of constrained Horn clauses. To solve a CHC system, it is necessary to find interpretations of the uninterpreted predicates that satisfy all clauses. We focus in the following on CHC systems over the theory of strings, with one unary uninterpreted predicate. Finding a valuation for this predicate p amounts to finding a set of words w for which $p(w)$ holds, so that all clauses are satisfied. As a finite representation is needed, HornStr will focus on regular language solutions.

2.1 Regular Model Checking

Example 1. Consider the token passing protocol on a ring topology, with two initial tokens, red and blue, at first and last position respectively, moving synchronously in opposite directions, without possibly colliding. A configuration can be seen as a word over $\Sigma = \{r, b, n\}$ where n denotes the absence of a token. Assume we are interested in the safety property “the two tokens never reach the other end”, invalidated by a word in the language $\mathcal{L}(b \cdot n^* \cdot r)$. One can observe that an initial odd distance between the two tokens is a necessary and sufficient condition for avoiding these configurations.

Checking the safety of this protocol can therefore be specified with the following CHC System:

$$V_i \in \mathcal{L}(rn(nn)^*b) \rightarrow p(V_i) \quad (1)$$

$$p(V_i) \wedge V_i \in \mathcal{L}(bn^*r) \rightarrow \perp \quad (2)$$

$$p(V_i) \wedge V_i = A \cdot (rn) \cdot B \cdot (nb) \cdot C \wedge V_o = A \cdot (nr) \cdot B \cdot (bn) \cdot C \rightarrow p(V_o) \quad (3)$$

$$p(V_i) \wedge V_i = A \cdot (nb) \cdot B \cdot (rn) \cdot C \wedge V_o = A \cdot (bn) \cdot B \cdot (nr) \cdot C \rightarrow p(V_o) \quad (4)$$

$$p(V_i) \wedge V_i = A \cdot (rb) \cdot B \wedge V_o = A \cdot (br) \cdot B \rightarrow p(V_o) \quad (5)$$

The variables V_i, V_o, A, B, C in the clauses are implicitly universally quantified. The clauses can be partitioned into three categories: $Init = \{(1)\}$ expresses membership of an initial configuration, while $Bad = \{(2)\}$ expresses undesired configurations. The rest of the clauses, $Tr = \{(3), (4), (5)\}$, model the different transitions where tokens move synchronously, possibly changing their order in (5). Note that arbitrarily many extra string variables may be used as long as they are universally quantified. The different constraints involve string constraints either in the form of regular expression constraints ((1) and (2)) or in terms of string equality with concatenation operations ((3) – (5)).

Example 1 is a rather usual instance of RMC problem, where one asks whether a system is safe by finding an *inductive invariant*, that is, a set of states or words containing all initial states (1), no bad state (2), and that is closed under the transitions (3) – (5).

Several candidate sets can be considered, such as the set of all reachable words from an initial clause (the strongest possible invariant), or the set of words from which no bad state can be reached (the weakest possible invariant). Recall, however, that we need to compute finite representations of the considered invariants; in our case, as regular languages. The previously mentioned sets are therefore less useful: any reachable and any unsafe configuration must have tokens at equal distance for the word borders, making the language irregular. However, a suitable regular inductive invariant does exist, for example $\mathcal{L}(n^*\Sigma(n(nn)^*)\Sigma n^*)$, which translates to “an odd distance between two tokens”.

2.2 String-rewrite system: The MU puzzle

The previous CHC system provided an example of a Regular Model Checking problem for a system with an initial state of arbitrary length, but where transitions preserve the length of the word. Such transitions can usually be represented by length-preserving transducers. HornStr’s input formalism is, however, not restricted to this setting, and can, for example, be applied to string-rewrite systems:

Example 2. The MU puzzle [25] is a string-rewrite system over the alphabet $\Sigma = \{M, I, U\}$: Its objective is to determine whether the string MU can be derived from the Initial string MI by applying the given rewriting rules: $R = \{(xI \rightarrow xIU), (Mx \rightarrow Mxx), (xIIIy \rightarrow xUy), (xUUy \rightarrow xy) \mid x, y \in \Sigma^*\}$. For example, using the first rule, the string MI is transformed to MIU in one step.

We can model the puzzle using the following CHCs, where all variables $V_i, V_o, x, y \in \Sigma^*$ are universally quantified:

$$V_i = MI \rightarrow p(V_i) \quad (1)$$

$$p(V_i) \wedge V_i = xI \wedge V_o = xIU \rightarrow p(V_o) \quad (2)$$

$$p(V_i) \wedge V_i = Mx \wedge V_o = Mxx \rightarrow p(V_o) \quad (3)$$

$$p(V_i) \wedge V_i = xIIIy \wedge V_o = xUy \rightarrow p(V_o) \quad (4)$$

$$p(V_i) \wedge V_i = xUUy \wedge V_o = xy \rightarrow p(V_o) \quad (5)$$

$$p(V_i) \wedge V_i = MU \rightarrow \perp \quad (6)$$

This CHC system is satisfiable, proving that the MU puzzle cannot be solved.

3 Architecture of HornStr

The HornStr framework integrates CHC and string constraints, leveraging automata learning techniques in combination with string solvers. This integration addresses complex problems expressed in SMT-LIB files, and modeling, e.g., parameterized systems or string-rewrite systems. Figure 1 illustrates the overall architecture of HornStr.

The framework commences with an SMT-LIB formatted file as its input, a format prevalent in the SMT community for describing problems that require solutions to satisfy constraints involving complex data types and operations. The *Learners* play a crucial role in synthesizing predicates based on regular constraints. They employ two different strategies:

1. *SAT-based Enumeration* utilizes SAT solvers to generate potential solutions, as well as string solvers to assess whether the solution satisfies given CHCs. In case of violated CHCs, the string solvers can provide a counterexample. Initially, the set of counterexamples is empty. The learner constructs a Deterministic Finite-state Automaton (DFA) as a hypothesis solution that accepts every word. This DFA is transformed into a regular expression via an intermediate translator, implemented by Brzozowski and McCluskey’s state elimination method [11]. The translator then sends an SMT-LIB query to the *String Solvers* to check for consistency with the CHCs. Upon receiving the query, the string solvers check the solution behind the scenes, returning either *unsat* or a counterexample to the learner.
2. *Active Learner* directly interacts with the learning model through queries. This learner constructs both equivalence queries and membership queries to verify if a string or sequence belongs to the model’s language or reachability queries to determine if a certain state or condition is achievable. It maintains an observation table [5] in its cache, from which it constructs a DFA. The *Reachability* module is responsible for communicating with the string solvers to ascertain whether the queried word is within the language: this involves several string queries, enumerating initial words (*Init*), then using all applicable transitions (*Tr*) to find all reachable words iteratively. The

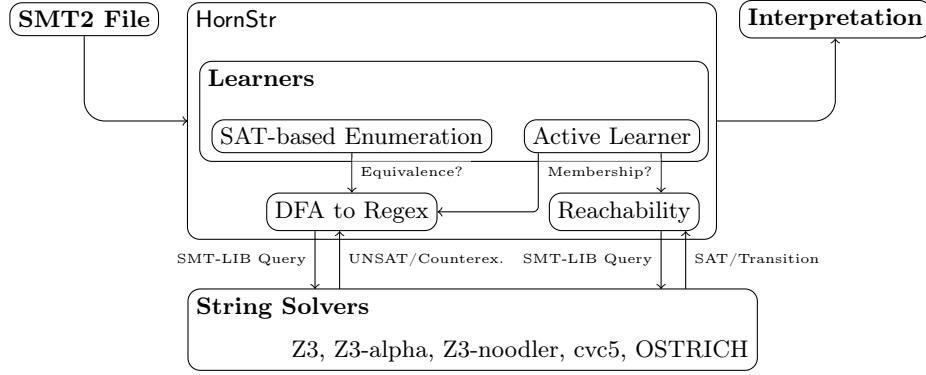


Fig. 1. The overall framework of HornStr.

membership query is answered positively when the desired word is found in the reachable fragment, or negatively if all the words of the same length, or up to a fixed constant, have been explored. The latter rule constitutes a heuristic inspired by the length-preserving transition models.

Furthermore, String Solvers respond to queries from the Learners by resolving a series of string constraints. To enhance the efficiency of answering equivalence and membership queries, the framework has integrated an incremental solving technique. Each CHC is assigned to a dedicated solver thread, for pre-computation purposes. For each word or automata query, the system saves the current constraints (push), inserts the new query constraint, computes the result, and upon obtaining the result, it restores the saved constraints (pop), provides the response, and prepares for the next query. Through a command line argument, the user can also instruct HornStr to handle all CHCs using a single solver, as this may save processing time for larger equivalence queries. On the contrary, word queries involve small input values, so they usually benefit from specific String Solver optimizations, one for each clause. HornStr employs a variety of state-of-the-art solvers, such as Z3, Z3-alpha, Z3-noodler, cvc5, and OSTRICH. Each of these solvers brings unique capabilities that range from basic string manipulations to more complex pattern matching and replacement operations. These specialized tools are adept at managing string operations within the constraints specified in SMT-LIB queries. Additionally, the framework offers a configuration file for users to specify their own string solver, as an external implementation of the interactive mode of the SMT-LIB 2.6 standard.

The execution of HornStr progresses through the following phases:

1. **Initialization:** The procedure begins with the selection of a suitable learning strategy and a string solver. Subsequently, an SMT-LIB file containing the uninterpreted predicate declaration, followed by constraint Horn clauses, is loaded, and the designated solver is instantiated together with the necessary oracles.

2. **Query Processing and Model Refinement:** When employing the SAT-based enumeration approach [45], the learner initially constructs a DFA with a single state and an empty counterexample set using a SAT solver. Once an appropriate DFA is generated that integrates the counterexample set, an equivalence check is conducted against the hypothesis using the string solver. If a new counterexample is detected, the hypothesis undergoes refinement and reconstruction. If the SAT solver returns an unsatisfiable outcome, the automaton’s state space is incrementally expanded, and the process iterates until the string solver fails to find further counterexamples and accepts the hypothesis.
Alternatively, if the active learner is selected [15], membership queries are issued to verify whether a given word w belongs to the target language L , leveraging the reachability module. This initiates an iterative process in which membership queries facilitate hypothesis generation, which is subsequently validated via equivalence queries.
3. **Solution Generation:** Based on the preliminary results and constructed queries, the string solver is employed to analyze and resolve regular constraints. Within this framework, the solver integrates the *Init* and *Tr* components to determine whether a word w is accepted. Conversely, if the word is rejected, the decision is justified through the *Bad* and *Tr* components.
For equivalence queries, all Horn clauses are evaluated by testing them with two free variables, var_{in} and var_{out} . For example, if var_{out} appears as part of a word in the hypothesis and satisfies the *Bad* clause, it is classified as a negative counterexample. Similarly, positive and inductive counterexamples can be identified using the *Init* and *Tr* components, respectively. If unsupported by the learner, inductive counterexamples are converted into positive and negative counterexamples thanks to reachability analysis, following the *strict but generous teacher* [15] concept.

4 Evaluation

In this section, we evaluate the performance and capabilities of HornStr⁷ [30] on a set of benchmarks derived from the verification of distributed systems and string rewriting systems. HornStr uses string solvers as oracles for membership and equivalence queries, the choice of the solvers in use is an important aspect of its performance.

Our evaluation is divided into two parts. First, we examine how the different string solvers can handle the string formulas generated as queries during the CHC-solving process. As described in Section 3, HornStr supports incremental solving, which can improve efficiency by reusing information across related queries. We compare the performance of string solvers on both incremental and non-incremental queries.

Second, we evaluate HornStr’s overall performance using the string solvers that performed best in the first part of the evaluation. Experiments were con-

⁷ <https://arg-git.informatik.uni-kl.de/pub/string-chc-lib>

ducted on an Intel Core i7-10510U CPU at 1.8GHz with 16 GB of RAM running on Windows 11.

Our benchmarks are derived from two distinct domains:

- **Verification of Distributed Systems:** We transform Regular Model Checking protocols [15,18] into Constrained Horn Clause (CHC) programs using automatic translations: Bakery[34], Szymanski[49,21], Dijkstra[42], Burns[42], Dining Philosopher Protocol[24], Israeli-Jalfon’s self-stabilising protocol[28], Resource-allocator protocol[17], David Gries’s coffee can problem[37], german protocol[3] and Kanban production system[19].
- **String Rewriting Systems:** We also manually model the MU puzzle and EqDist protocols as CHC programs, demonstrating the versatility of the approach.

4.1 Results of the String Solver Experiments

Table 1 provides a comparison of the string solvers Z3, cvc5, Z3-noodler, Z3-alpha, and OSTRICH. The benchmarks are categorized into incremental and non-incremental queries, further divided by the query type: membership or equivalence. Our primary metric of interest is the number of benchmarks solved, as failing to resolve even a single query can prevent the CHC solver from terminating. The timeout for each benchmark is set to 30s.

Equivalence queries predominantly involve reasoning over regular expressions but may also include word equations when these are part of the Horn clause. Membership queries, while also involving regular expressions, tend to emphasize disequalities ($x \neq c$, where x is a string variable and c is a string constant).

In the incremental setting, the membership results are relatively similar, with all solvers processing over 514 benchmarks. Notably, Z3-Noodler leads by solving all 523 benchmarks in an average of 109.7 seconds, whereas OSTRICH, cvc5, Z3, and Z3-alpha solve between 514 and 518 benchmarks in slightly higher runtimes.

For the incremental equivalence queries, we see different behaviors among the solvers. Z3-noodler solves all 396 queries in just 15.5 seconds, while OSTRICH manages 378. On the other hand, cvc5, Z3, and Z3-alpha only solve between 109 and 126 queries. A similar pattern shows up in the non-incremental equivalence queries: Z3-noodler handles all 848 queries, with OSTRICH coming in close with 784, whereas cvc5, Z3, and Z3-alpha solve between 403 and 457 queries. In the case of membership queries, every solver covers nearly all of the 30,902 benchmarks, with only cvc5 and OSTRICH missing about 1%, while Z3-noodler and Z3 turn out to be the fastest to solve them all.

Across both incremental and non-incremental benchmarks, the results demonstrate a consistent pattern: membership queries are generally handled well by most solvers, while equivalence queries involving regular expressions remain a challenge for many. Notably, automata-based solvers such as Z3-noodler and OSTRICH consistently show superior performance on equivalence queries, likely due to their design being well-suited for reasoning over regular expressions. These

results also highlight the high incrementality of our approach, as seen when comparing the total time spent on all incremental vs. non-incremental queries. Note that OSTRICH’s overall runtime is a bit higher partly due to the JVM startup time incurred for each benchmark.

To address the challenges faced by solvers struggling with equivalence queries, we experimented with different settings and flags for those solvers and implemented a regular expression simplifier on our end before sending the queries. The simplifier aimed to reduce the nesting of Kleene stars using algebraic transformations on regular expressions. While this led to marginal improvements for some poorly performing solvers, it had little impact overall and even worsened performance for solvers already handling regular expressions effectively.

Table 1. Comparison of state-of-the-art string solvers. Benchmarks are divided into incremental and non-incremental membership and equivalence queries. The timeout is 30s. Timeouts are excluded from solved time.

Solver	Incremental				Non-Incremental			
	Mem	Time (s)	Equiv	Time (s)	Mem	Time (s)	Equiv	Time (s)
OSTRICH	514	453.2	378	410.8	30 773	16 504.9	784	980.8
cvc5	517	97.8	126	7948.4	30 652	610.5	457	270.7
Z3	517	33.7	109	506.7	30 902	1511.7	403	102.1
Z3-noodler	523	109.7	396	15.5	30 902	806.3	848	17.9
Z3-alpha	518	86.4	109	516.6	30 902	3839.1	404	162.7

4.2 Results of the HornStr Experiments

After evaluating the performance of various string solvers as membership and equivalence oracles in our preliminary experiments, we now assess HornStr for CHC solving. Based on the incremental benchmark results (Table 1), we chose Z3 for membership queries and Z3-noodler for equivalence queries.

We developed an automatic parser that transforms length-preserving RMC protocols into CHC SMT2 format, incorporating word equations and regular membership constraints. Next, we evaluate the efficiency of HornStr using both SAT-based Enumeration and the Active Learner, as described in Section 3. In our evaluation, we record whether HornStr produces a deterministic finite automaton for the uninterpreted invariant within a predefined time limit or identifies an unsafe trace during the benchmark evaluation.

Our evaluation demonstrates that our tool solved most benchmarks in under a second using either SAT-based enumeration or the active learner. Notably, SAT-based enumeration solved every protocol listed in Table 2, whereas the active learner failed to find solutions for some benchmarks. However, certain protocols—such as *Kanban* and *German*—exceeded the 60-second timeout due to

the complexity of transitions in their CHC representations. Detailed evaluation results are presented in Table 2.

Table 2. Comparison of protocols: automaton size and learning time across SAT-Based and active Learner

Protocol	SAT-based Enumeration		Active Learner	
	Size	Time(s)	Size	Time(s)
Token Pass	3	0.41	3	0.10
2 Tokens Pass	3	0.78	6	0.57
3 Tokens Pass	2	0.30	2	0.17
Power-Binary	1	0.2	1	0.01
Bakery	2	0.15	3	0.37
Burns	2	2.09	✗	TO
Coffee-Can	2	0.52	5	9.66
Coffee-Can-v2	3	0.31	4	23.45
Herman-Linear	2	0.11	2	0.08
Herman-Ring	2	0.51	2	0.33
Israeli-Jalfon	3	0.35	4	0.46
LR-Philo	2	0.80	3	2.84
Mux-Array	2	0.49	✗	TO
Resource-Allocator	2	0.14	4	25.19
Eqdist	3	1.45	✗	TO
MU Puzzle	3	11.01	✗	TO
Water-Jug	2	2.05	✗	TO
Dining-Crypt	2	10.02	✗	TO

5 Conclusions

We introduced **HornStr**, the first solver for invariant synthesis in RMC that leverages the SMT-LIB 2.6 Theory over Strings. By formulating invariant synthesis as a problem of solving CHCs over strings, **HornStr** provides a standardized, scalable, and automated approach to verification. Our approach enables seamless integration of modern SMT solvers into RMC verification, bridging parameterized verification and string solving in a novel way.

Our evaluation demonstrated **HornStr**’s effectiveness in handling complex verification tasks, including parameterized systems and string rewriting problems (e.g., the MU puzzle). By integrating incremental solving techniques, **HornStr** significantly improves the performance of string solvers, reducing computational overhead and enhancing scalability. Additionally, our work contributes more than 10,000 new **QF_S** constraints, providing a valuable benchmark suite for SMT solver evaluations.

We mention several future research avenues. The first is to extend **HornStr** by handling general CHCs over strings, i.e., non-linear and monadic CHCs that

permit *symbolic alphabets*. This would allow one to model certain protocols, wherein process IDs are passed around (e.g. Chang-Roberts protocol; see [26,46]). Second, one could extend our CHC framework to other types of RMC verification including liveness [38,36] and bisimulation [27,37].

Acknowledgement. Jiang, Lin and Markgraf are supported by the European Research Council under the European Union’s Horizon 2020 research and innovation programme under number 101089343. Rümmer is supported by the Swedish Research Council through grant 2021-06327. The authors thank the CAV reviewers and all of the OSTRICH, cvc5, Z3, Z3-noodler, and the Z3-alpha team developers.

References

1. Abdulla, P.A.: Regular model checking. *STTT* **14**(2), 109–118 (2012). <https://doi.org/10.1007/s10009-011-0216-8>
2. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An smt solver for string constraints. In: International conference on computer aided verification. pp. 462–469. Springer (2015)
3. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24–April 1, 2007. Proceedings 13. pp. 721–736. Springer (2007)
4. Abdulla, P.A., Haziza, F., Holík, L.: All for the price of few. In: Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20–22, 2013. Proceedings. pp. 476–495 (2013)
5. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and computation* **75**(2), 87–106 (1987)
6. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 – November 2, 2018. pp. 1–9. IEEE (2018). <https://doi.org/10.23919/FMCAD.2018.8602994>, <https://doi.org/10.23919/FMCAD.2018.8602994>
7. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
8. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, pp. 24–51. Springer (2015)
9. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. *STTT* **14**(2), 167–191 (2012). <https://doi.org/10.1007/s10009-011-0205-y>, <http://dx.doi.org/10.1007/s10009-011-0205-y>
10. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: CAV. pp. 403–418 (2000)
11. Brzozowski, J.A., McCluskey, E.J.: Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers* (2), 67–76 (1963)
12. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. *Proc. ACM Program. Lang.* **6**(POPL), 1–31 (2022). <https://doi.org/10.1145/3498707>, <https://doi.org/10.1145/3498707>
13. Chen, T., Hague, M., Lin, A., Ruemmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proceedings of the ACM on Programming Languages* **3**, 1–30 (01 2019)

14. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 24–33. Springer Nature Switzerland, Cham (2024)
15. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Stewart, D., Weissenbacher, G. (eds.) *2017 Formal Methods in Computer Aided Design, FMCAD 2017*, Vienna, Austria, October 2-6, 2017. pp. 76–83. IEEE (2017). <https://doi.org/10.23919/FMCAD.2017.8102244>
16. De Moura, L., Bjørner, N.: Z3: an efficient smt solver. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. p. 337–340. TACAS’08/ETAPS’08, Springer-Verlag, Berlin, Heidelberg (2008)
17. Donaldson, A.F.: Automatic techniques for detecting and exploiting symmetry in model checking. Ph.D. thesis, University of Glasgow (2007)
18. Esparza, J., Raskin, M., Welzel, C.: Regular Model Checking Upside-Down: An Invariant-Based Approach. In: Klin, B., Lasota, S., Muscholl, A. (eds.) *33rd International Conference on Concurrency Theory (CONCUR 2022)*. Leibniz International Proceedings in Informatics (LIPIcs), vol. 243, pp. 23:1–23:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). <https://doi.org/10.4230/LIPIcs.CONCUR.2022.23>, <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2022.23>
19. Geeraerts, G., Raskin, J.F., Van Begin, L.: Expand, enlarge and check: New algorithms for the coverability problem of wsts. *Journal of Computer and system Sciences* **72**(1), 180–203 (2006)
20. Grebenschikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. *ACM SIGPLAN Notices* **47**(6), 405–416 (2012)
21. Gribomont, E.P., Zenner, G.: Automated verification of szymanski’s algorithm. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 424–438. Springer (1998)
22. Gutiérrez, C.: Solving equations in strings: On makanin’s algorithm. In: *Latin American Symposium on Theoretical Informatics*. pp. 358–373. Springer (1998)
23. Heule, M.J., Verwer, S.: Exact dfa identification using sat solvers. In: *Grammatical Inference: Theoretical Results and Applications: 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings 10*. pp. 66–79. Springer (2010)
24. Hoare, C.A.R.: Communicating sequential processes. *Communications of the ACM* **21**(8), 666–677 (1978)
25. Hofstadter, D.R.: Gödel, Escher, Bach: An Eternal Golden Braid. Basic Books, Inc. (1979)
26. Hong, C., Lin, A.W.: Regular abstractions for array systems. *Proc. ACM Program. Lang.* **8**(POPL), 638–666 (2024)
27. Hong, C., Lin, A.W., Majumdar, R., Rümmer, P.: Probabilistic bisimulation for parameterized systems - (with applications to verifying anonymous protocols). In: *CAV (1)*. *Lecture Notes in Computer Science*, vol. 11561, pp. 455–474. Springer (2019)
28. Israeli, A., Jalfon, M.: Uniform self-stabilizing ring orientation. *Information and Computation* **104**(2), 175–196 (1993)
29. Jež, A.: Recompression: A simple and powerful technique for word equations. *J. ACM* **63**(1) (Feb 2016). <https://doi.org/10.1145/2743014>, <https://doi.org/10.1145/2743014>

30. Jiang, H., Lin, A.W., Markgraf, O., Rümmer, P., Stan, D.: Hornstr: Invariant synthesis for regular model checking as constrained horn clauses (Apr 2025). <https://doi.org/10.5281/zenodo.15190404>, <https://doi.org/10.5281/zenodo.15190404>
31. Kan, S., Lin, A.W., Rümmer, P., Schrader, M.: Certistr: a certified string solver. In: CPP. pp. 210–224. ACM (2022)
32. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.* **256**(1-2), 93–112 (2001). [https://doi.org/10.1016/S0304-3975\(00\)00103-1](https://doi.org/10.1016/S0304-3975(00)00103-1), [https://doi.org/10.1016/S0304-3975\(00\)00103-1](https://doi.org/10.1016/S0304-3975(00)00103-1)
33. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: Proceedings of the eighteenth international symposium on Software testing and analysis. pp. 105–116 (2009)
34. Lamport, L.: A new solution of dijkstra’s concurrent programming problem. In: Concurrency: the works of leslie lamport, pp. 171–178 (2019)
35. Legay, A.: T(O)RMC: A tool for (omega)-regular model checking. In: CAV. pp. 548–551 (2008)
36. Lengál, O., Lin, A.W., Majumdar, R., Rümmer, P.: Fair termination for parameterized probabilistic concurrent systems. In: TACAS (1). Lecture Notes in Computer Science, vol. 10205, pp. 499–517 (2017)
37. Lin, A.W., Nguyen, T.K., Rümmer, P., Sun, J.: Regular symmetry patterns. In: Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17–19, 2016. Proceedings 17. pp. 455–475. Springer (2016)
38. Lin, A.W., Rümmer, P.: Liveness of randomised parameterised systems under arbitrary schedulers. In: CAV (2). Lecture Notes in Computer Science, vol. 9780, pp. 112–133. Springer (2016)
39. Lin, A.W., Rümmer, P.: Regular model checking revisited. In: Model Checking, Synthesis, and Learning. Lecture Notes in Computer Science, vol. 13030, pp. 97–114. Springer (2021)
40. Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Majumdar, R., Nowotka, D.: Solving string constraints using SAT. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 187–208. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_9, https://doi.org/10.1007/978-3-031-37703-7_9
41. Lu, Z., Siemer, S., Jha, P., Day, J., Manea, F., Ganesh, V.: Layered and staged monte carlo tree search for smt strategy synthesis. In: Larson, K. (ed.) Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24. pp. 1907–1915. International Joint Conferences on Artificial Intelligence Organization (8 2024). <https://doi.org/10.24963/ijcai.2024/211>, <https://doi.org/10.24963/ijcai.2024/211>, main Track
42. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
43. Markgraf, O., Hong, C.D., Lin, A.W., Najib, M., Neider, D.: Parameterized synthesis with safety properties. In: Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings 18. pp. 273–292. Springer (2020)
44. Mora, F., Berzish, M., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4: A multi-armed string solver. In: Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24. pp. 389–406. Springer (2021)

45. Neider, D., Jansen, N.: Regular model checking using solver technologies and automata learning. In: NFM. pp. 16–31 (2013). https://doi.org/10.1007/978-3-642-38088-4_2
46. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016. pp. 614–630. ACM (2016). <https://doi.org/10.1145/2908080.2908118>, <https://doi.org/10.1145/2908080.2908118>
47. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 3–18. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_1, https://doi.org/10.1007/978-3-031-13185-1_1
48. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 2010 IEEE Symposium on Security and Privacy. pp. 513–528. IEEE (2010)
49. Szymanski, B.K.: Mutual exclusion revisited. In: Proceedings of the 5th Jerusalem Conference on Information Technology, 1990. 'Next Decade in Information Technology'. pp. 110–117. IEEE (1990)
50. Tateishi, T., Pistoia, M., Tripp, O.: Path-and index-sensitive string analysis based on monadic second-order logic. ACM Transactions on Software Engineering and Methodology (TOSEM) **22**(4), 1–33 (2013)
51. Trinh, M.T., Chu, D.H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 218–240. Springer International Publishing, Cham (2016)
52. Vardhan, A., Viswanathan, M.: LEVER: A tool for learning based verification. In: CAV. pp. 471–474 (2006). https://doi.org/10.1007/11817963_43
53. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: CAV. pp. 88–97 (1998). <https://doi.org/10.1007/BFb0028736>, <http://dx.doi.org/10.1007/BFb0028736>
54. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for php. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 154–157. Springer (2010)