Unbounded Model-Checking with Interpolation for Regular Language Constraints

Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte

The University of Melbourne
{ggange,jnavas,pjs,harald,schachte}@csse.unimelb.edu.au

Abstract. We present a decision procedure for the problem of, given a set of regular expressions $R_1, \ldots, R_n$, determining whether $R = R_1 \cap \cdots \cap R_n$ is empty. Our solver, REVENANT, finitely unrolls automata for $R_1, \ldots, R_n$, encoding each as a set of propositional constraints. If a SAT solver determines satisfiability then R is non-empty. Otherwise our solver uses unbounded model checking techniques to extract an interpolant from the bounded proof. This interpolant serves as an overapproximation of R. If the solver reaches a fixed-point with the constraints remaining unsatisfiable, it has proven R to be empty. Otherwise, it increases the unrolling depth and repeats. We compare REVENANT with other state-of-the-art string solvers. Evaluation suggests that it behaves better for constraints that express the intersection of sets of regular languages, a case of interest in the context of verification.

1 Introduction

Strings are ubiquitous in software. Many web applications, for example, construct database queries from user-provided strings. The rapid rise in the popularity of these applications and the proliferation of vulnerabilities to attacks such as SQL injection and cross-site scripting can explain a renewed interest in developing practical, efficient verification techniques for reasoning about strings.

Regular expressions are commonly used to define sanitization checks over strings. For example, a regular expression can be used as a filter to exclude strings that exhibit a particular attack pattern. Given a set of sanitization filters $F_1, \ldots, F_n$ and an attack pattern P, we wish to determine if $F_1 \cap \cdots \cap F_n \cap P$ is empty. Although this problem is decidable, the implementation of practical algorithms is still an open issue. Most state-of-the-art solutions (e.g., [22,9,11]) rely on the classical product algorithm for intersection of DFAs, but they differ in how they tackle the two main performance bottlenecks: exponential blowup while converting regular expressions to DFAs, and the large state space of the product automaton. These solvers, particularly lazy solvers [11], are very efficient when the query is underconstrained because they can avoid building the full product automaton. To prove unsatisfiability, however, they must enumerate the complete set of reachable product states. This is not desirable in the context of verification, where we may be testing the intersection of many languages (with a potentially exponential product automaton), and unsatisfiable queries are common.

In this paper, we develop an alternative approach for checking intersection of a set of regular expressions using established SAT-based unbounded model-checking techniques. We first translate the regular expressions $R_1, \ldots, R_n$ into a set of *SFAs (symbolic finite-state automata)* [21]. An SFA is a generalization of a finite-state automaton where transitions are labelled with a symbolic encoding of a set of values, rather than requiring a separate transition for each value. Although the use of SFAs is not new, it is worth mentioning that our method does not require any determinization of the SFAs. Next, we unroll each SFA up to a fixed depth k, encode each unrolled SFA as a set of propositional constraints, and use a SAT solver to determine satisfiability. This encoding consists mainly of the conjunction of the constraints originating from the initial states, transitions, and final states of each unrolled SFA. If the constraints are satisfiable then we return a string w that belongs to the intersection of the languages as a witness. Otherwise, we have proven that the intersection is empty for strings up to length k. However, this is not sufficient to prove that the intersection is empty in the unbounded case. To overcome this, we apply McMillan induction [16]. The idea is to use *interpolation* [5] to generalize a proof for the length-k case to one that proves the intersection empty for any length. In summary:

- We address the unbounded model checking problem as applied to string solving; unlike other "unbounded" methods, we combine SAT solving with the interpolation-based approach of McMillan [16], instantiating that framework to the case of SFA unrolling.
- We describe REVENANT, a publicly available solver designed to handle the intersection of *sets* (beyond *pairs*) of regular languages efficiently.
- We compare with the state-of-the-art solvers REX [22], DPRLE [9], and STRSOLVE [11], using a standard benchmark set of regular expressions extracted from real applications [21], together with intersection instances designed to stress test solvers. REVENANT performs very well on instances in its target domain, while remaining competitive across benchmarks.

2 Related Work

Methods for solving language constraints can loosely be divided into bounded and unbounded methods.

Bounded methods (e.g., HAMPI [14], KUDZU [20], and CFGANALYZER [1]) unroll the constraints to a given length bound, encode the unrolled problem as a set of propositional formulas, and use a SAT solver to determine satisfiability. These methods can be quite efficient finding a satisfying assignment and often can express a wider range of constraints than the unbounded methods. However, if unsatisfiability results then no useful conclusions can be derived. Thus, these tools are not suitable for verification, which is our main motivation.

Existing unbounded methods instead build the classical decision procedures. Wasserman *et al.* [23] build on ideas by Minamide [18] to overapproximate string variables with context-free grammars and model a potential SQL attack with a finite automaton. They build the product of a push-down automaton, constructed from the context-free grammar, with the finite automaton that captures a potential SQL attack, and check if the language of the resulting automaton is empty. Rex [22] improves upon the classical FSA algorithms by introducing *symbolic* finite-state automata (SFAs), where each edge is annotated with a *set* (in the form of a one-place predicate), rather than a single symbol. Rex then uses the SMT solver Z3 [6] to manipulate edge constraints during operations such as intersection and determinization. Efficiency is achieved by keeping SFAs "clean" (avoiding unsatisfiable formulas as edge labels on moves). Hooimeijer *et al.* [9] present DPRLE which also relies on the classical algorithms for regular languages involving concatenation and subset constraints. DPRLE utilises dependency analysis information to slice away product automaton states that are irrelevant for the query. The same authors have later developed a lazy solver called StrSolve [11] which outperforms previous approaches. StrSolve performs a lazy search space enumeration by considering only those states from the product automata needed for the query.

While our method falls in the "unbounded" class, we differ from previous approaches in our use of McMillan induction. As mentioned, our work can be seen as an application of McMillan's interpolation-based framework [16].

3 Unbounded Model Checking with Interpolation

Consider an unsatisfiable set F of Boolean formulas which has been partitioned into two sets A and B. An *interpolant* [5] of A and B is a formula P containing only variables that are common between A and B, and satisfying the properties

$$A \models P$$
$$P \wedge B \models \perp$$

It is well known that, given an unsatisfiability proof for $A \wedge B$, an interpolant P can be generated in linear time [19,17].

The use of interpolants for SAT-based model checking was pioneered by McMillan [16]. SAT-based unbounded model-checking is formulated in terms of a transition system $T = (S, I, \delta, F)$, with a set of state variables S, initial conditions I, transition relation δ and final conditions F. A propositional encoding is constructed for the given transition system unrolled to depth k, and is tested for satisfiability. If the finite unrolling is satisfiable, we have produced a concrete error trace. Otherwise, we can generate an interpolant in accordance with the partitioning shown in Fig. 1. Note that $A = I \wedge \delta_0$ represents the set of T states reachable in one step from the initial conditions. Since the interpolant P is expressed in terms of state variables s_1 (the only variables shared by A and B), and satisfies the property $I \wedge \delta_0 \models P$, P is an overapproximation of states reachable in one step from the initial state. Now, by replacing each variable from s_1 in P by the corresponding variable from s_0, an over-approximation $P[s_0/s_1]$ of the reachable states is obtained, according to which F is still unreachable. If

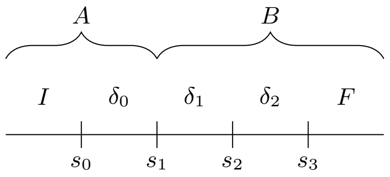

Fig. 1. The partitioning used for interpolant generation. Note that the only variables shared between A and B are s_1.

$P[s_0/s_1] \models I$, our initial conditions encompass all the reachable states; and since F is still unreachable, it must remain so after unrolling to any depth. If not, we can relax the initial condition to $I \vee P[s_0/s_1]$ and repeat the process from there. Eventually, either the relaxation will fail to weaken the initial condition, that is, the condition reaches a fixed-point, in which case we have proven unsatisfiability in the unbounded case, or the conjunction of constraints becomes satisfiable, in which case we must perform a longer unroll. This process is guaranteed to terminate [16].

4 Regular Language Representations

We now describe how regular languages are represented as symbolic finite-state automata, and how we manipulate these. We consider a simple constraint language given by the following grammar:

$$\begin{array}{lll} \mathit{Constraint} & \rightarrow & \mathit{Var} \in \mathit{RegExp} \\ \mathit{RegExp} & \rightarrow & \mathit{Lit} \mid \mathit{RegExp} + \mathit{RegExp} \mid \mathit{RegExp}\ \mathit{RegExp} \mid \mathit{RegExp}^* \end{array}$$

The only possible constraints are membership queries. *Lit* is the set of string literals. Intersection between regular expressions $R_1, \ldots, R_n$ can be expressed via the constraints $x \in R_1$, $\ldots$, $x \in R_n$. For convenience, our implementation supports other standard constructions such as ranges, bounded repetitions, special characters (\d, \w, and so on) which are made to conform with the grammar in a preprocessing step.

4.1 Symbolic Finite State Automata

Formally, a finite-state automaton is defined by a tuple $(Q, \Sigma, \delta, q_0, F)$. The automaton begins in state $q_0 \in Q$; at each step, the state is updated according to the transition relation δ. The automaton is said to *accept* if, at the end of input, it is in a state $q_i \in F$.

In a typical finite-state automaton, each edge is expressed as a triple (q_s, α, q_e), with $q_s, q_e \in Q$ and $\alpha \in \Sigma$. A *symbolic* finite-state automaton [22] extends this by encoding the edge as (q_s, ψ, q_e), where $\psi \subseteq \Sigma$ encodes the set of input values permitted by the transition. A number of encodings have been proposed for these sets of values, including hash-sets, range predicates and bit-vector constraints; these are discussed in [8].

Given that we wish to construct a propositional encoding of the automaton, we also require an encoding that can be conveniently transformed into a propositional formula, in addition to providing efficient construction and a concise encoding of value sets. Accordingly, we construct binary decision diagrams over the bit-vector encoding of the characters.

4.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are often used to represent Boolean functions. *BDD expressions* are defined inductively:

- $\mathcal{F}$ and $\mathcal{T}$ are BDD expressions.
- If x is a variable and e_1 and e_2 are BDD expressions then $\mathsf{ite}(x, e_1, e_2)$ is a BDD expression.

The *meaning* of a BDD expression is defined:

$$\begin{aligned} [\![\mathcal{F}]\!] &= \mathit{false} \\ [\![\mathcal{T}]\!] &= \mathit{true} \\ [\![\mathsf{ite}(x, e_1, e_2)]\!] &= (x \wedge [\![e_1]\!]) \vee (\neg x \wedge [\![e_2]\!]) \end{aligned}$$

BDDs are the directed acyclic graphs that result when sub-expressions are allowed to be shared.

An *ordered* BDD assumes that variables are ordered by a linear order relation $\prec$. A BDD is an OBDD iff, whenever it is of form $\mathsf{ite}(x, e_1, e_2)$, e_1 and e_2 are OBDDs and each x' occurring in e_1 or e_2 satisfies $x \prec x'$. An OBDD e is *reduced* (and is called an *ROBDD*) iff $[\![\cdot]\!]$ is injective across e, that is, for all BDDs e_1 and e_2 appearing in e, $[\![e_1]\!] \equiv [\![e_2]\!] \Rightarrow e_1 = e_2$.

While the size of BDDs may be exponential in the number of variables, limited unions of character ranges can be concisely represented, as illustrated in Fig. 2. In these diagrams, $\mathsf{ite}(x, e_1, e_2)$ is captured by showing a solid arc from node x to the root of e_1 and a dashed arc from x to the root of e_2; except we omit the sink $\mathcal{F}$ and all arcs leading to it.

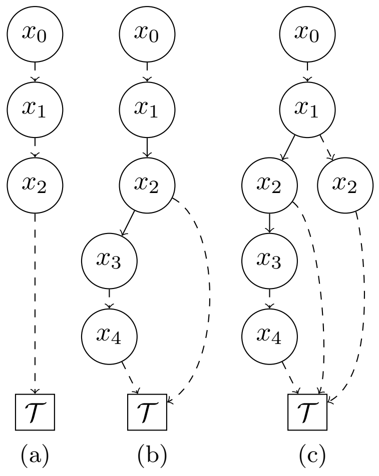

Fig. 2. BDDs that represent the 5-bit ranges (a) [0-3], (b) [8-12], and (c) the disjunction of the two

```
$\mathbf{sfa\_reduce}((Q, \Sigma, \delta, q_0, F))$
    $depend := (q \mapsto \{q' \mid (q', \psi, q) \in \delta, q' \neq q\})$
    $\mathbf{foreach}\ q \in Q\ \mathbf{do}$
        $ufind.\mathsf{make}(q)$
        $queue.\mathsf{insert}(q)$
    $shash := \emptyset$
    $\mathbf{while}(\neg queue.\mathsf{empty}())$
        $q := queue.\mathsf{pop}()$
        $q_m := ufind.\mathsf{find}(q)$
        $dests := \{q \mapsto \bot \mid q \in Q\}$
        $\mathbf{for}\ (q_s, \psi, q_d) \in \delta,\ \text{such that}\ q_s = q_m$
            $dests(q_d) := \psi \vee dests(q_d)$
        $q_t := shash(\langle q_m \in F, dests\rangle)$
        $\mathbf{if}(\ q_t \neq \textsc{NotFound})$
            $\mathbf{if}(q_m \neq q_t)$
                $ufind.\mathsf{merge}(q_m, q_t)$
                $\mathbf{foreach}\ q' \in depend(q_m)$
                    $queue.\mathsf{insert}(ufind.\mathsf{find}(q'))$
                $depend(q_t) := depend(q_t) \cup depend(q_m)$
        $\mathbf{else}$
            $shash(\langle q_m \in F, dests\rangle) := q_m$
    $Q' := \{q \in Q \mid ufind.\mathsf{find}(q) = q\}$
    $q'_0 := ufind.\mathsf{find}(q_0)$
    $\delta' := \emptyset$
    $\mathbf{foreach}\ q'_i, q'_j \in Q'$
        $\alpha' := \bigvee \{\alpha \mid (q_i, \alpha, q_j) \in \delta, ufind.\mathsf{find}(q_i) = q'_i, ufind.\mathsf{find}(q_j) = q'_j\}$
        $\delta' := \delta' \cup \{(q'_i, \alpha', q'_j)\}$
    $F' := \{q \in F \mid ufind.\mathsf{find}(q) = q\}$
    $\mathbf{return}\ (Q', \Sigma, \delta', q'_0, F')$
```

Fig. 3. Pseudo-code for SFA reduction

4.3 SFA Reduction

The standard construction of an NFA from a regular expression often introduces a considerable number of redundant and equivalent states. The approach taken by Rex is to give symbolic equivalents to the classical ϵ-elimination, determinization and DFA minimization algorithms.

Given that the size of a deterministic automaton is potentially exponential relative to the corresponding NFA, we would prefer to reduce the size of the non-deterministic SFA directly. While finding the minimum number of states for an NFA is PSPACE-hard, approaches have been presented [13,12] for reducing the size of an NFA directly.

We first eliminate ϵ transitions, following the procedure used by Rex. We then use a simple structural hashing approach to eliminate redundant states introduced during automaton construction. All states are initially assumed to be distinct, and we progressively merge pairs of states which have identical transition relations.

The pseudo-code for this is given in Fig. 3. *ufind* maintains the renaming of equivalent states, and can be efficiently implemented using a union-find data-structure. *depend*(q_j) is the set of states that must be checked if state q_j is renamed; *queue* maintains the set of states that still need to be checked, and *shash* is used to check state equivalences. This approach is strictly weaker than the partition refinement of Ilie and Yu [13]; however, in the presence of symbolic edges, it avoids the need to test the intersection of large numbers of transitions.

5 Model Checking Formulation

We consider the problem of, given a set of regular expressions $R_1, \ldots, R_n$, determining whether the intersection $R_1 \cap R_2 \cap \cdots \cap R_n$ is empty. By converting each regular expression R_i into a SFA A_i, this can be reduced to determining whether there is a sequence $x \in \Sigma^* = x_1, \ldots, x_k$ of inputs that will leave every automaton in an accept state.

We can reformulate this as a transition system with state space $Q' = Q_1 \times \ldots \times Q_n$, initial state $q'_0 = \langle q_1^0, \ldots, q_n^0 \rangle$, accepting states $F' = F_1 \times \ldots \times F_n$, and transition relation

$$\delta(\langle s_1, \ldots, s_n \rangle, x) = \langle \delta_1(s_1, x), \ldots, \delta_n(s_n, x) \rangle$$

where δ_i is the transition relation for A_i. We wish to determine if there is any reachable state of the form

$$\langle q_1, \ldots, q_n \rangle \in F' \qquad \text{(i.e., } \forall_{i \in \{1, \ldots, n\}}\ q_i \in F_i)$$

We can then apply the unbounded model-checking procedure to this revised formulation. The procedure is described in Fig. 4 and resembles the one described by McMillan [16]. The main differences are in how we unroll the SFAs and define the interpolation groups A and B in order to approximate the bounded proofs generated by the SAT solver. Fig. 4 gives a high level description of the method. The procedure **Intersection** takes as inputs the set of transition systems that represent all the automata to be intersected and a value k that represents the unrolling depth. The algorithm makes use of I, F, and the procedure **unroll** which are explained in Section 5.1. For now, suffice it to say that I and F denote the Boolean encoding of the initial states q'_0 and accepting states F', respectively. The procedure **unroll** unwinds the transition system up to depth k. For convenience, **unroll** can be called to return the layers from 0 to 1 and 1 to k separately, so as to simplify the formation of interpolation groups A and B.

If the procedure **Intersection** returns INCONCLUSIVE then we need to increase the value of k. Although the process will eventually terminate, judicious choice of the next k can speed up the convergence of the fixed-point significantly. Experimentally we have observed that a good choice the first time we get inconclusive results is to increase k to the maximum of the shortest accepting run from any state in a single automaton. After that, we increase k by doubling its value.

```
Intersection($\{T_1, \dots, T_n\}, k$)
    // $T_1 \equiv \langle Q_1, \Sigma, \delta_1, q_1^0, F_1 \rangle, \dots, T_n \equiv \langle Q_n, \Sigma, \delta_n, q_n^0, F_n \rangle$
    $R := I$
    $A' := \bigwedge_{1 \le i \le n}$ unroll$(0, 1, T_i)$
    $B := \bigwedge_{1 \le i \le n}$ unroll$(1, k, T_i) \wedge F$
    while ($true$)
        $A := R \wedge A'$
        Run SAT solver on $A \wedge B$
        if $A \wedge B$ is satisfiable then
            if $R = I$ then
                return Sat
            else
                return Inconclusive
        else
            $P :=$ genInterpolant($A$,$B$)
            if $P[s_1/s_0] \models R$ then
                return UnSat
            else
                $R := R \vee P[s_1/s_0]$
```

Fig. 4. Pseudo-code for the procedure based on unbounded model checking with interpolation for testing whether the intersection of multiple SFAs is empty

5.1 Finite Unrolling

We introduce a Boolean variable $\langle q_i^k \rangle$ to represent the automaton being in state q_i at time k, and $\langle e_{i,j}^k \rangle$ to represent the automaton transitioning from state q_i to q_j during the k^{th} step. We use $\psi_{i,j}^k$ to denote the corresponding transition constraint (we assume that all transitions between a pair of states are merged into a single edge). $\mathsf{pred}(q_j)$ denotes the set of states with an outgoing edge to q_j.

We can use these variables to encode the transition relation at each layer:

$$\bigwedge_{(q_i,\psi,q_j)\in\delta} (\neg\langle q_i^k \rangle \Rightarrow \neg\langle e_{i,j}^k \rangle) \wedge (\neg\langle \psi^k \rangle \Rightarrow \neg\langle e_{i,j}^k \rangle) \wedge \bigwedge_{(q_j \in Q)} \Big(\bigwedge_{q_i \in \mathsf{pred}(q_j)} \neg\langle e_{i,j}^k \rangle\Big) \Rightarrow \neg\langle q_j^{k+1} \rangle$$

$$\bigwedge_{(q_i,\psi,q_j)\in\delta} (\langle q_i^k \rangle \wedge \langle \psi^k \rangle \Rightarrow \langle e_{i,j}^k \rangle) \wedge \bigwedge_{(q_i,\psi,q_j)\in\delta} (\langle e_{i,j}^k \rangle \Rightarrow \langle q_j^{k+1} \rangle) \qquad (\star)$$

The formulas marked $(\star)$ are not necessary for correctness but can reduce the state space of the problem.

However, directly encoding the final condition would require checking at *every* step whether every automaton is in an accept state. To avoid this, we allow the language accepted by each automaton to be padded with an additional termination character (denoted \$ in Fig. 5). We then only need to test for acceptance at the final step. Unlike a conventional automaton unrolling, where we unroll only from the start state, we must introduce all state variables at the top layer;

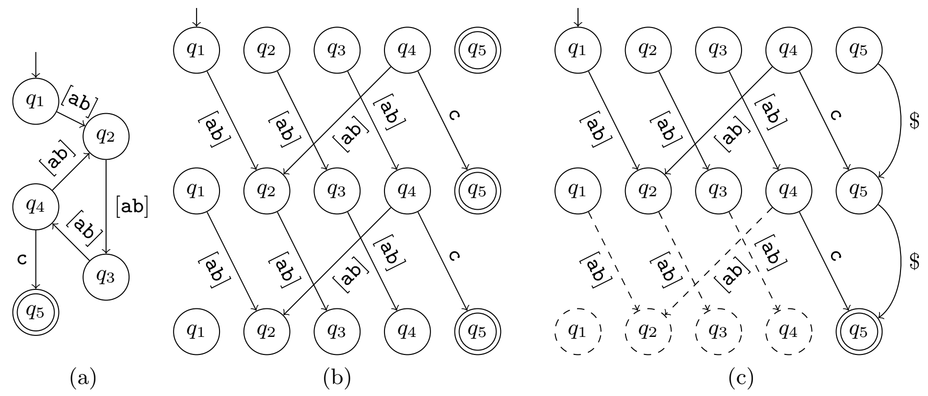

Fig. 5. Transition relation for an automaton of (a) ([ab]{3})+ c, (b) unrolled two steps, and (c) after adding transitions to allow for padding the end of string. States and edges that can be safely eliminated are shown dashed.

otherwise we cannot correctly compute the relaxed initial conditions, and may incorrectly conclude unsatisfiability.

In layers 2 to k, there may be states and edges which cannot reach an accept state in layer k. These states cannot affect the satisfiability of the overall clauses, and can be safely omitted.

Example 1. Consider the automaton shown in Fig. 5(a). The transition relation for this is (b) unrolled two steps, and then (c) corrected to allow for \$-padding. Consider the clauses generated for state q_5 in the second layer. We introduce $\langle e^1_{4,5}\rangle$ and $\langle e^1_{5,5}\rangle$ for the incoming edges, and $\langle q^1_5\rangle$ for the node, and the following formulae:

$$(\neg\langle q^1_4\rangle \Rightarrow \neg\langle e^1_{4,5}\rangle) \wedge (\neg\langle x_1 \in [\mathtt{ab}]\rangle \Rightarrow \neg\langle e^1_{4,5}\rangle)$$
$$(\neg\langle q^1_5\rangle \Rightarrow \neg\langle e^1_{5,5}\rangle) \wedge (\neg\langle x_1 = \$\rangle \Rightarrow \neg\langle e^1_{5,5}\rangle)$$
$$\neg\langle e^1_{4,5}\rangle \wedge \neg\langle e^1_{5,5}\rangle \Rightarrow \neg\langle q^2_5\rangle$$

After generating similar clauses for each edge and node in the unrolled graph, we add the initial and final conditions requiring that the machine begins in the start state, and ends in an accept state:

$$I = \neg\langle q^0_2\rangle \wedge \neg\langle q^0_3\rangle \wedge \neg\langle q^0_4\rangle \wedge \neg\langle q^0_5\rangle \qquad\qquad F = \langle q^2_5\rangle$$

Notice the dotted states q^2_1 to q^2_4. The truth value of state q^2_5 is not dependent on the value of these states; as such, they cannot cause unsatisfiability, or affect the interpolant. In general, however, we require all variables for the first unrolled state in order to generate correct interpolants.

At the first iteration, this conjunction of formulas is clearly unsatisfiable; there is no path from q_1^0 to q_5^2. We then compute the interpolant for the system of constraints, yielding $P = \neg\langle q_4^1\rangle \wedge \neg\langle q_5^1\rangle$. This is not a fixed-point, since there is a solution satisfying P that doesn't satisfy I. At the second iteration, we compute the relaxed initial conditions $I' = I \vee P$ (which upon simplification gives P). As I' permits the machine to be in state q_3, the system of constraints is now satisfiable. So we cannot prove unsatisfiability at this depth; we must unroll the automaton further.

It may be tempting to also omit states which are known to be false, such as q_1^1 shown in Fig. 6. However, if $\langle q_1^1\rangle$ is omitted, a possible interpolant that may be generated is $P = \neg\langle q_2^1\rangle \wedge \neg\langle r_3^1\rangle$. When this is mapped back to the initial state, the algorithm will incorrectly detect satisfiability (with $q_1^0 \wedge r_2^0$), and unroll. The same interpolant will be generated after any number of unrolling steps, so the solver will never terminate.

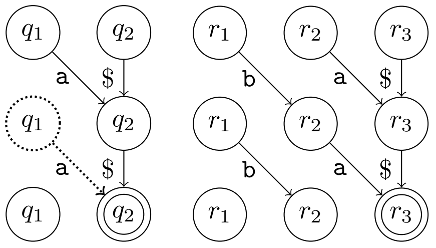

Fig. 6. Dotted state q_1^1 is always false, as it has no incoming edges. Still, it cannot be eliminated from the encoding.

5.2 Language Relaxation

Several of the languages tested in our first experiment in Section 6 generate automata with large numbers of states owing to the use of bounded repetition. The presence of these states can cause performance of the solver do degrade substantially; we conjecture that this is due to MathSAT not performing simplification of the generated interpolants, resulting in very large encodings of the set of reachable states.

However, in most of the cases we considered, the cause of unsatisfiability for the intersection of a pair of languages was unrelated to repetition operators. As a result, for regular expressions R_1 to R_n which make use of bounded repetition, we first check the satisfiability of $U(R_1) \cap \cdots \cap U(R_n)$, where U eliminates bounded repetition as follows:

$$\begin{aligned} U(e\{\mathtt{0},\mathtt{1}\}) &= U(e)? & U(e_1 \ \mathbf{op}\ e_2) &= U(e_1)\ \mathbf{op}\ U(e_2) \\ U(e\{\mathtt{0},\mathtt{j}\}) &= U(e)* & U(\mathbf{op}(e)) &= \mathbf{op}(U(e_1)) \\ U(e\{\mathtt{i},\mathtt{j}\}) &= U(e)+ \end{aligned}$$

If the intersection of these overapproximated languages is empty, we can terminate early without testing the full automata.

6 Experimental Results

To evaluate the method described in the previous sections, we have implemented a prototype solver, REVENANT,[1] in C++ using MathSAT5 [7] for SAT-solving and interpolant generation. All experiments have been run on a single core of a 2.7GHz Core i7-26202M with 7.8Gb memory. We compare the performance of REVENANT with REX [22][2] and DPRLE [9], and STRSOLVE [11][3] on a range of common benchmarks (first and second experiments).

Previous papers have focused on the intersection of only pairs of languages, for which the product automaton has in the worst case $O(n^2)$ states. However, a general solver for regular language constraints should be able to handle more complex systems of constraints. To test the performance of these methods on larger conjunctions of automata, we also present two classes of problems (third and fourth experiments) which exhibit more challenging behaviour.

Intersection of Multiple Languages. We generate intersections of multiple languages $\bigcap_{i \in \{2,\ldots,5\}} R_i$ such that R_i is each of the ten regular expressions extracted from some real-world applications that appeared originally in [15]. Table 1(a) shows the results of our evaluation running the different tools. Note that previous works (e.g., [22,9,11]) used the same set of regular expressions but regular set difference of pairs of languages was used instead of intersection. The reason why we do not perform the same experiment here is that our current implementation does not handle regular complement. Column T is the solving time of each tool, column T_{out} denotes the number of times that a timeout of 60 seconds expired, and S/U is the number of satisfiable versus unsatisfiable instances.

Unsurprisingly, REVENANT does not outperform the existing solvers in the case of pairs of automata, as it has the overhead of introducing $O(|R|k)$ variables and the corresponding clauses. However, as the number of languages increases, this up-front cost is outweighed by the gain from not generating the complete product automaton.

Generation of Long Strings. Our next experiment evaluates the performance of each solver for generating long strings from underconstrained systems. For this, we repeat an experiment from [22], probing the intersection of the regular expressions $[\mathtt{a}-\mathtt{c}]{*}\mathtt{a}[\mathtt{a}-\mathtt{c}]\{\mathtt{n}+\mathtt{1}\}$ and $[\mathtt{a}-\mathtt{c}]{*}\mathtt{b}[\mathtt{a}-\mathtt{c}]\{\mathtt{n}\}$. Table 1(b) shows, for various n, the time spent by each tool to generate a single string that matches both regular expressions. This is a worst-case scenario for our method since the two regular expressions are trivially satisfiable and therefore, our full encoding of the automata does not pay off.

[1] REVENANT is available at `http://ww2.cs.mu.oz.au/~ggange/revenant/`
[2] We run REX using the Mono framework 2.10.8.1.
[3] At the time of writing, the available STRSOLVE version [10] only supports intersection of pairs of languages.

Table 1. Comparison of REVENANT with existing string solvers, on several classes of regular expressions. All times are in seconds.

	REVENANT			REX			DPRLE			STRSOLVE		
	T	T_{out}	S/U	T	T_{out}	S/U	T	T_{out}	S/U	T	T_{out}	S/U
$i = 2$	4.48	0	22/23	38.78	0	22/23	2.08	0	22/23	0.32	0	22/23
$i = 3$	18.55	0	35/85	173.19	1	34/85	102.60	1	34/85	N/A	N/A	N/A
$i = 4$	130.88	1	35/174	401.22	4	31/175	613.71	7	28/175	N/A	N/A	N/A
$i = 5$	83.67	1	21/230	503.93	6	15/231	865.80	13	8/231	N/A	N/A	N/A

(a) Intersection of real-world regular expressions

	50	100	150	200	250	300	350	400	450	500
REVENANT	0.15	0.54	1.18	2.12	3.42	5.08	7.39	9.78	13.15	17.42
REX	0.10	0.16	0.27	0.46	0.73	1.24	1.92	2.90	4.00	5.54
DPRLE	0.01	0.06	0.09	0.17	0.25	0.36	0.48	0.65	0.78	0.96
StrSolve	0.00	0.00	0.02	0.03	0.04	0.06	0.09	0.11	0.17	0.21

(b) Generation of long strings

	2	4	6	8	10	12	14	16	18
REVENANT	0.01	0.02	0.04	0.06	0.06	0.05	0.09	0.08	0.14
REX	0.10	0.10	0.12	0.16	0.30	0.79	3.75	16.86	OutOfMemory
DPRLE	0.00	0.00	0.00	0.02	0.08	0.48	3.09	29.57	333.80

(c) Exponential branching

	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$
REVENANT	0.01	0.00	0.02	0.02	0.02	0.03
REX	0.10	0.10	0.18	3.27	OutOfMemory	OutOfMemory
DPRLE	0.00	0.00	0.03	0.40	15.21	OutOfMemory

(d) Exponential cycles

Exponential Branching. Even if we restrict attention to finite languages, the size of the product automaton may still be exponential in size. We construct a family of languages of the form

$$\begin{aligned} L_i = \;& (\texttt{[0-1]\{i-1\}0[0-1]\{n-1\}0[0-1]\{n-i\}}\varphi_{\texttt{i}}) \\ & \mid (\texttt{[0-1]\{i-1\}1[0-1]\{n-1\}1[0-1]\{n-i\}}\varphi_{\texttt{i}}) \end{aligned}$$

such that $\varphi_1 \cap \cdots \cap \varphi_n$ is empty. An example language family of this kind is

$$\begin{aligned} L_1 &= \texttt{[0-1]\{0\}0[0-1]\{3\}0[0-1]\{3\}[bcd]} \mid \texttt{[0-1]\{0\}1[0-1]\{3\}1[0-1]\{3\}[bcd]} \\ L_2 &= \texttt{[0-1]\{1\}0[0-1]\{3\}0[0-1]\{2\}[acd]} \mid \texttt{[0-1]\{1\}1[0-1]\{3\}1[0-1]\{2\}[acd]} \\ L_3 &= \texttt{[0-1]\{2\}0[0-1]\{3\}0[0-1]\{1\}[abd]} \mid \texttt{[0-1]\{2\}1[0-1]\{3\}1[0-1]\{1\}[abd]} \\ L_4 &= \texttt{[0-1]\{3\}0[0-1]\{3\}0[0-1]\{0\}[abc]} \mid \texttt{[0-1]\{3\}1[0-1]\{3\}1[0-1]\{0\}[abc]} \end{aligned}$$

Table 1(c) shows the time for running the solvers for different values of n. For this experiment, we run REVENANT without relaxation, as the relaxed languages are trivially unsatisfiable. Clearly, this is an ideal case for REVENANT, as we can immediately prove unsatisfiability, where other solvers must explore the entire state space.

Exponential Paths. Conjunctions of languages may also contain cycles of exponential length. Consider the set of languages

$$\begin{aligned} L_1 &= \texttt{[a-c]*([a-c]\{3\})+[bc]} \\ L_2 &= \texttt{[a-c]*([a-c]\{5\})+[ac]} \\ L_3 &= \texttt{[a-c]*([a-c]\{7\})+[ab]} \end{aligned}$$

The intersection of languages $L_1 \cap L_2 \cap L_3$ is empty. However, as the cycle length of each language is coprime, both the product construction and search-based methods will generate all possible combinations of cycle-positions before the automata are synchronized at the loop exit, and the intersection can be proven empty. Table 1(d) shows the time spent for each tool to prove unsatisfiability. As in the previous case, we run REVENANT without relaxation. As before, REVENANT is substantially faster, as it can prove unsatisfiability without unrolling to the synchronization point.

The last two experiments have illustrated extreme cases in which REVENANT can significantly outperform the other existing tools. Similarly, we could construct other examples where our tool would perform very poorly. Consider the following set of languages similar to the previous one

$$\begin{aligned} L_1 &= \texttt{[a-c]+[bc]d[a-c]\{3\}+} \\ L_2 &= \texttt{[a-c]+[ac]d[a-c]\{5\}+} \\ L_3 &= \texttt{[a-c]+[ab]d[a-c]\{7\}+} \end{aligned}$$

In this case our SAT-based method, when used without relaxation, detects unsatisfiability due to the unsynchronized loop exits, rather than the $\varphi_{\texttt{i}}\texttt{d}$ chokepoint. The corresponding interpolant weakens the initial conditions too far, and the problem must be fully unrolled before unsatisfiability can be proven. With relaxation, however, we prove unsatisfiability without unrolling.

7 Conclusions and Further Work

We have described a new method for testing emptiness of the intersection of multiple regular languages, based on unbounded model-checking techniques. We have implemented a prototype solver, REVENANT, which uses this method; combined with language relaxation, REVENANT is competitive with existing solvers on realistic problem instances. We have also illustrated families of problems where this method is exponentially faster than existing techniques.

The differences between solvers on various families of constraints suggests that hybrid approaches should be studied, in particular for software verification. While our prototype currently handles only language intersection constraints, we intend to expand this to support concatenation constraints ($x \circ y \in L$), as well as negation and disjunction of constraints.

The relaxation described in Section 5.2 is an approximation of a traditional abstraction-refinement loop [4], where we only perform a single refinement step. It would be interesting to replace this with finer-grained progressive strengthening.

Our relaxation is currently a purely syntactic transformation. Such a scheme is only possible if the bounded repetition is already specified as part of the input. If the language is generated procedurally, or provided as an automaton, the transformation is no longer viable. Instead, it may be worthwhile to develop algorithms that examine the automaton directly for relaxation opportunities.

Finally, it would be interesting to apply the same technique to the testing of emptiness of context-free language intersection, by iteratively refining regular overapproximations to the languages involved. Chaki *et al.* [2] do this in a different context, namely the verification of concurrent C programs; an implementation was incorporated into the model checker MAGIC [3].

Acknowledgments. We wish to thank Pieter Hooimeijer for providing both software and valiant support on short notice. We acknowledge support of the Australian Research Council through Discovery Project Grant DP110102579.

References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing Context-Free Grammars Using an Incremental SAT Solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying Concurrent Message-Passing C Programs with Recursive Calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
3. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Transactions on Software Engineering 30(6), 388–402 (2004)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
5. Craig, W.: Linear reasoning: A new form of the Herbrand-Gentzen theorem. Journal of Symbolic Logic 22(3), 250–268 (1957)
6. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8, 1–27 (2012)
8. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011)
9. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Proc. 2009 ACM SIGPLAN Conf. Programming Language Design and Implementation, pp. 188–198. ACM (2009)

10. Hooimeijer, P., Weimer, W.: Solving string constraints lazily. In: Proc. IEEE/ACM Conf. Automated Software Engineering, pp. 377–386 (2010)
11. Hooimeijer, P., Weimer, W.: StrSolve: Solving string constraints lazily. Automated Software Engineering 19(4), 531–559 (2012)
12. Ilie, L., Solis-Oba, R., Yu, S.: Reducing the Size of NFAs by Using Equivalences and Preorders. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 310–321. Springer, Heidelberg (2005)
13. Ilie, L., Yu, S.: Reducing NFAs by invariant equivalences. Theoretical Computer Science 306(1-3), 373–390 (2003)
14. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA 2009), pp. 105–116. ACM (2009)
15. Li, N., Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Reggae: Automated test generation for programs using complex regular expressions. In: Proc. 24th IEEE/ACM Int. Conf. Automated Software Engineering, pp. 515–519 (2009)
16. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
17. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science 345(1), 101–121 (2005)
18. Minamide, Y.: Static approximation of dynamically generated web pages. In: Proc. 14th Int. Conf. World Wide Web, pp. 432–441. ACM Press (2005)
19. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. Journal of Symbolic Logic 62(2), 981–998 (1997)
20. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proc. IEEE Symp. Security and Privacy, pp. 513–528. IEEE Computer Society (2010)
21. Veanes, M., de Halleux, P., Tillman, N.: Rex: Symbolic regular expression explorer. Microsoft Research Technical Report MSR-TR-2009-137, Microsoft Research, Redmond, WA (2009)
22. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proc. Third Int. Conf. Software Testing, Verification and Validation, pp. 498–507. IEEE Comp. Soc. (2010)
23. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proc. ACM SIGPLAN 2007 Conf. Programming Language Design and Implementation, pp. 32–41 (2007)