Supporting Proofs for Control-Flow Recovery from Partial Failure Reports

Peter Ohmann Alexander Brooks Loris D'Antoni Ben Liblit
University of Wisconsin–Madison, USA
{ohmann,albrooks,loris,liblit}@cs.wisc.edu

Abstract

Debugging post-deployment failures is difficult, in part because failure reports from these applications usually provide only partial information about what occurred during the failing execution. We introduce approaches that answer control-flow queries about a failing program's execution based on failure constraints given as formal languages. A key component of our approach is the introduction of a new class of subregular languages, the unreliable trace languages (UTL), which allow us to answer many common queries in polynomial time. This report supplements the description of these new approaches with formal proofs. Specifically: we prove completeness for our context-insensitive query problem, tightly bind polynomial-time decidability of query recovery to the UTL class, and prove partial correctness for our approach to answering user queries with UTL constraints.

1. Introduction

This report provides supplemental proofs for the techniques and language classes introduced by Ohmann et al. [3]. The report is not meant to be read in isolation. Rather, it should be read alongside the full conference paper [3], as this report glosses over many details that are fully expounded in the original paper.

Debugging is a tedious and difficult task, particularly when failures occur after software is deployed to end users. In our full paper [3], we formalize and evaluate analysis techniques that can answer control-flow queries as *Possible* or *Impossible* given a program's control-flow graph (CFG) and formal descriptions of constraints obtained from a failure report for that program. These queries can be posed with or without including calling context sensitivity in our analysis; we define these two problems as the *context-sensitive query recovery problem* and the *context-insensitive query recovery problem* respectively (see Ohmann et al. [3, section 3.3]).

At a high level, each approach first encodes the program's CFG, each failure constraint, and the query as formal languages whose accepted strings are sequences of edges from the CFG. Both query recovery problems are then defined as intersection-emptiness checks over these languages: if a string exists in all of the provided languages, then the query is satisfiable (i.e., *Possible*) given the provided CFG and failure report. In the full paper [3], we also introduce a new subclass of regular languages, the unreliable trace languages (UTL), for which we are able to solve the context-insensitive query recovery problem in polynomial time. In this paper, we provide supplemental proofs for these approaches, showing the relationship between the context-sensitive and context-insensitive query recovery problems, and proving properties of the class UTL.

This paper is organized as follows. Section 2 briefly summarizes definitions from the full paper [3]. The following sections provide supplementary proofs. Section 3 proves that the context-insensitive query problem provides a complete approximation of the context-sensitive problem. Section 4 demonstrates the bounds of UTL by proving that two small extensions to this language class result in NP-hard query problems. Section 5 provides partial proofs of correctness for the intersection-emptiness algorithms for UTL from Ohmann et al. [3, section 4.2]. We conclude in section 6.

2. Background

This section briefly reiterates some of the key definitions from the full paper [3]. All definitions and descriptions are taken from Ohmann et al. [3], where they are discussed in much greater detail.

2.1 Control-Flow Queries

Per Ohmann et al. [3, definition 1], a control-flow graph (CFG) is a tuple $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$ where:

- N is a finite set of nodes;
- n_0 is the entry node;
- $\mathbb{L}$ is a finite set of function names;
- $E_i \subseteq N \times N$ is a set of internal (intraprocedural) edges;
- $E_c \subseteq N \times (N \times \mathbb{L}) \times N$ is a set of function-call edges, such that for every $(n, (n_1, \alpha), n') \in E_c$, $n = n_1$; and
- $E_r \subseteq N \times (N \times \mathbb{L}) \times N$ is a set of function-return edges.

From a CFG, G, we use $N_{\mathbb{L}}$ to denote the alphabet $N \cup \underline{N \times \mathbb{L}}$ and $\widehat{N_{\mathbb{L}}}$ to denote the tagged alphabet $N \cup \widehat{(N \times \mathbb{L})}$. A path through G is given as a sequence of edges $\pi = \langle e_1 \cdots e_j \rangle$, such that $\pi \in (E_i \cup E_c \cup E_r)^*$. Informally, a path, π is a context-insensitive path in G if, for every adjacent pair of edges from π, (e_1, e_2), the target node of e_1 equals the source node of e_2. Further, π is a context-sensitive path in G if every edge $e \in E_r$ from π is paired with a preceding edge from E_c with an identical function label, and these pairings are strictly nested. Then, *proj*(π) is a context-insensitive (resp. context-sensitive) trace in G if π is a context-insensitive (resp. context-sensitive) path in G via the definition of *proj*() from Ohmann et al. [3, definition 2]. Informally, we translate the sequence of edges from π to a corresponding sequence of symbols from $\widehat{N_{\mathbb{L}}}$. Finally, define languages $L_s(G)$ and $L_i(G)$ as the set of all context-sensitive and context-insensitive traces in G, respectively. Per Ohmann et al. [3, theorem 1], $L_s(G) \subseteq L_i(G)$ for any CFG, G.

Symbolic visibly-pushdown automata (s-VPAs) describe nested-word languages, where words may have hierarchical structure in addition to their linear encoding and contain *symbolic* transition functions to better support large or infinite alphabets [1]. Most importantly, s-VPAs can compactly and precisely encode a trace of program control flow. An s-VPA that operates over ordinary words without hierarchical structure (e.g., a program's control flow, ignoring the program stack) is a Symbolic Finite Automaton (s-FA) [5, 6]. In Ohmann et al. [3, section 3.2], we describe methods for encoding $L_s(G)$ and $L_i(G)$ as s-VPA and s-FA respectively. An individual element of a failure report is a *constraint*, given as a language. A constraint C is s-VPA-definable (resp. s-FA-definable) if there exists an s-VPA (resp. s-FA) A_C such that $L(A_C) = C$.

The inputs to the query recovery problem are:

- G, a control-flow graph
- $\{FP_1, \ldots, FP_n\}$, a set of s-VPA-definable failure constraints defined over $\widehat{N_{\mathbb{L}}}$
- R, an s-VPA-definable query constraint defined over $\widehat{N_{\mathbb{L}}}$

Then, per Ohmann et al. [3, definition 5], the *context-sensitive query recovery problem* is to check whether

$$L_s(G) \cap \bigcap_i FP_i \cap R \neq \emptyset.$$

If all FP_i and R are s-FA-definable, then, per Ohmann et al. [3, definition 6], the *context-insensitive query recovery problem* is to check whether

$$L_i(G) \cap \bigcap_i FP_i \cap R \neq \emptyset.$$

In section 3, we prove that the context-insensitive recovery problem provides an unsound but complete approximation for the context-sensitive recovery problem.

2.2 Unreliable Trace Languages

The unreliable trace languages (UTL) are a class of sub-regular languages. Formally, for an alphabet Σ, the following characterizes the class UTL:

$$UTL = \left\{\Sigma^* \sigma_1 \Sigma^* \sigma_2 \Sigma^* \ \ldots \ \Sigma^* \sigma_n \Sigma^* \quad \text{for } n \geq 0 \text{ and such that all } \sigma_i \in \Sigma\right\}$$

Note that, since UTL is a subclass of the regular languages, all languages in UTL are s-FA-definable.

UTL is an important class for the context-insensitive query recovery problem from Ohmann et al. [3, definition 6]. Specifically, if all $FP_i \in UTL$ and $R \in UTL$, then we can answer user queries in polynomial time via the algorithm from Ohmann et al. [3, section 4.2]. In section 4, we prove that two small extensions to UTL result in NP-hard context-insensitive query recovery problems, and thereby provide evidence for the tightness of the UTL class. In section 5, we prove partial correctness for our intersection algorithm from Ohmann et al. [3, section 4.2].

3. Proof of Completeness For Context-Insensitive Recovery

Theorem 1 *Given*

- *a control flow graph G with nodes in N and labels in $\mathbb{L}$,*
- *a set of s-VPA-definable* failure constraints $\{FP_1, \ldots, FP_n\}$ *of nested words over the alphabet $N_{\mathbb{L}}$, and*
- *an s-VPA-definable* query constraint R *of nested words over the alphabet $N_{\mathbb{L}}$,*

if $\{FP'_1, \ldots, FP'_n\}$ and R' are s-FA-definable languages of words over the alphabet $\widehat{N_{\mathbb{L}}}$, such that for all i, $FP_i \subseteq FP'_i$ and $R \subseteq R'$, and

$$L_i(G) \cap \bigcap_i FP'_i \cap R' = \emptyset$$

then

$$L_s(G) \cap \bigcap_i FP_i \cap R = \emptyset.$$

Proof. Assume, by contradiction that $L_i(G) \cap \bigcap_i FP'_i \cap R' = \emptyset$, but $L_s(G) \cap \bigcap_i FP_i \cap R \neq \emptyset$. This means that there exists a word $w \in \widehat{N_{\mathbb{L}}}^*$ such that $w \in L_s(G)$, $w \in FP_i$ for all i, and $w \in R$. Since, for all i, $FP_i \subseteq FP'_i$ and $R \subseteq R'$, we have that $w \in FP'_i$ for all i, and $w \in R'$. From Ohmann et al. [3, theorem 1], we know that $L_s \subseteq L_i$, therefore $w \in L_i$. Therefore, $w \in L_i(G) \cap \bigcap_i FP'_i \cap R'$, which means that this set cannot be empty. Hence, a contradiction. □

4. Proofs of NP-hardness for Two Generalized Trace Language Classes

Unfortunately, the unreliable trace languages (described in Ohmann et al. [3, section 4.1]) are a very tight class. Specifically, the useful property of polynomial-time decidability for queries in unreliable trace languages appears not to generalize beyond this restrictive class. Here, we prove that two small extensions of the unreliable trace languages (to more generalized language families describing program trace properties) result in language intersection problems with NP-hard complexity.

Throughout this section we will often use regular expressions to define regular languages, though most of our techniques are defined over automata. The equivalence of regular expressions and finite-state automata is well-known, and, for each of the trace languages we define, this conversion is trivial and requires no more than a constant-factor increase from the number of regular expression terms to the number of FSA transitions.

4.1 Allowing Ambiguity

The first generalization that we consider relaxes the requirement that each constraint be composed of a sequence of *characters*, instead allowing a sequence of *character classes*. Specifically, we consider the class of languages:

$$\mathcal{A} = \big\{\Sigma^* \, C_1 \, \Sigma^* \, C_2 \, \ldots \, \Sigma^* \, C_n \, \Sigma^* \\ \text{for } n \geq 0 \text{ and such that all } C_i \subseteq \Sigma\big\}$$

Theorem 2 *The context-insensitive recovery problem from Ohmann et al. [3, definition 6] is NP-hard if all $FP_i \in \mathcal{A}$ and $R \in \mathcal{A}$.*

Proof. The proof is via a straightforward reduction from Boolean SAT on formulae in conjunctive normal form (CNF) [2].

We are given a CNF formula:

$$f = (p_{1,1} \vee p_{1,2} \vee \ldots) \wedge (p_{2,1} \vee p_{2,2} \vee \ldots) \wedge \ldots$$

We begin by converting each conjunct into a regular expression as follows:

$$\begin{aligned} f_i &= p_{i,1} \vee p_{i,2} \vee \cdots \vee p_{i,n} \\ &\Downarrow \\ r_i &= \Sigma^* \, [p_{i,1} \, p_{i,2} \, \ldots \, p_{i,n}] \, \Sigma^* \end{aligned}$$

Our alphabet, Σ, is comprised of all literals in all of these conjunct formulae, along with their negations. That is, from the single conjunct

$$\pi_1 \vee \overline{\pi_2} \vee \pi_3$$

we would add the following characters to Σ:

$$\pi_1, \; \overline{\pi_1}, \; \pi_2, \; \overline{\pi_2}, \; \pi_3, \; \overline{\pi_3}$$

Note that all $r_i \in \mathcal{A}$; that is, all r_i are generalized trace languages by our extended definition. Further, all r_i are languages over Σ.

Next, we need to construct a language to enforce the principle of excluded middle. In terms of our trace language, this corresponds to recognizing only strings with exactly one of each literal (from our original CNF formula) or its negation. To do so, we construct our principle of excluded middle regular expression:

$$L(M) = [\sigma_1 \, \overline{\sigma_1}][\sigma_2 \, \overline{\sigma_2}] \ldots [\sigma_{|\Sigma|} \, \overline{\sigma_{|\Sigma|}}]$$

from each $\sigma_i \in \Sigma$. This language recognizes those strings assigning "true" or "false" to each literal appearing in the original CNF formula. Note that $L(M)$ corresponds exactly to $L_i(G)$ for a control-flow graph, G, that is a sequence of branches. Now, recall that each r_i enforces one of the original conjuncts of f. Thus, if

$$\bigcap_i (r_i) \cap L(M) \neq \emptyset$$

the original formula (f) is satisfiable. Each of the above steps is a straightforward linear transformation from our original formula into a regular language, and, hence, the whole transformation is clearly polynomial. Therefore, determining intersection-emptiness for this generalized class of trace languages is NP-hard. □

4.2 Constrained Paths

The second generalization that we consider relaxes the requirement that each constraint have no detail on what happens during execution between observation points. Specifically, if our original alphabet is Σ_o, we consider the class of languages:

$$\mathcal{B} = \big\{C^* \, \sigma_1 \, C^* \, \sigma_2 \, C^* \, \ldots \, C^* \, \sigma_p \, C^* \\ \text{for } p \geq 0, \; C \subseteq \Sigma_o \text{ and such that all } \sigma_i \in \Sigma_o\big\}$$

For the proof, we require a slightly extended alphabet:

$$\Sigma = \Sigma_o \cup \{\#\}$$

where the symbol "#" is not present in Σ_o, and is used only as a marker in the proof.

Theorem 3 *The context-insensitive recovery problem from Ohmann et al. [3, definition 6] is NP-hard if all $FP_i \in \mathcal{B}$ and $R \in \mathcal{B}$.*

Proof. The proof is via reduction from the decision version of the Shortest Common Supersequence Problem (SCSP) on any alphabet (including a binary alphabet), which is proven NP-complete by Räihä and Ukkonen [4]. The following input characterizes a SCSP over an arbitrary alphabet:

- a size, z
- a set of sequences $S = s_1, s_2, \ldots, s_n$ such that for all $s_i \in S$, all characters s_{i_j} are members of alphabet $\Sigma_o = \Sigma \setminus \{\#\}$.

Solving the SCSP requires that one find a string R such that $|R| \leq z$ and for all $s_i \in S$, s_i is an ordered subsequence of R (that is, s_i is obtained by deleting zero or more elements from R).

We begin by converting each s_i to a regular expression:

$$s_i = s_{i_1}\, s_{i_2}\, \ldots\, s_{i_m}$$
$$\Downarrow$$
$$r_i = \Sigma^*\, s_{i_1}\, \Sigma^*\, s_{i_2}\, \Sigma^*\, \ldots\, \Sigma^*\, s_{i_m}\, \Sigma^*$$

Note that all $r_i \in \mathcal{B}$—that is, all r_i are generalized trace languages by our extended definition (with $C = \Sigma$)—and this conversion is a simple enumeration of the sequence (and, hence, is clearly polynomial). The intersection

$$\bigcap_i r_i$$

is always non-empty, as it always contains the concatenation of sequences $s_1\, s_2\, \ldots\, s_n$. However, any string in this intersection that contains no more than z characters from Σ_o serves as a witness for the original SCSP. Now, consider the language

$$L(Z) = (\Sigma_o^*\, \#)^z\, \Sigma_o^*$$

This language recognizes exactly those strings with z "#" characters. Further, $L(Z) \in \mathcal{B}$ (it is a generalized trace language with $C = \Sigma_o$). Finally, consider the language

$$L(V) = (\#\, \Sigma_o^?)^*$$

This language requires that every character from Σ_o be immediately preceded by a "#" character. Further, note that $L(V)$ corresponds exactly to $L_i(G)$ for a control-flow graph, G, that contains an infinite loop containing a large switch branching to nodes labeled with each $\sigma \in \Sigma_o$, all branching back together into the single entry node labeled "#". More concretely, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$ where:

$$N = \Sigma \qquad n_0 = \# \qquad \mathbb{L} = \emptyset$$
$$E_i = \{(\#, \sigma)\, \forall \sigma \in \Sigma\} \cup \{(\sigma, \#)\, \forall \sigma \in \Sigma\}$$
$$E_c = \emptyset \qquad E_r = \emptyset$$

The language

$$L(Z) \cap L(V)$$

contains those strings with exactly z "#" characters, where each "#" is optionally followed by a single character from Σ_o. Hence, this language contains at most z characters from Σ_o.

Now, if the language

$$L(R) = \bigcap_i (r_i) \cap L(Z) \cap L(V)$$

is non-empty, then a supersequence of size no larger than z exists for the original input sequences. This supersequence is comprised of the ordered sequence of characters from Σ_o (i.e., excluding all instances of "#") in any witness for $L(R)$. Note that $L(R)$ is an intersection of generalized trace languages (i.e., languages from $\mathcal{B}$) with a feasible control-flow graph language $L_i(G)$, and the above procedure checks this intersection for emptiness to obtain a solution to the original SCSP. All transformations in generating $L(R)$ are polynomial. Therefore, determining intersection-emptiness for this generalized class of trace languages is NP-hard. □

5. Proofs of Correctness For Unreliable Trace Language Intersection

In this section, we present partial correctness proofs for our approach to checking intersection-emptiness from Ohmann et al. [3, section 4.2]. Specifically, we prove that our procedure—given a control-flow graph, $G = (N, n_0, \mathbb{L}, E_i, E_c, E_r)$, and a vector of unreliable trace constraint vectors V that correspond to a set of failure report elements FP and a query R—answers *Possible* if and only if the language intersection $L_i(G) \cap \bigcap_j FP_j \cap R$ (by Ohmann et al. [3, definition 6]) is non-empty.

Theorem 4 *If our data-flow analysis procedure reports Possible, then the context-insensitive intersection $L_i(G) \cap \bigcap_j FP_j \cap R$ is non-empty.*

Proof. Note that for any node $n \in sccG$, either $in(n) = \bot$ or $in(n) = out(m)$ for some immediate predecessor m of n. If our procedure reports *Possible*, then there exists a path p through $sccG$ where $n_0 \in p_1$.nodes, $in(p_i) = out(p_{i-1})$ for all $i \in 2 \ldots |p|$, and $out(p_{|p|})$ is a vector of empty vectors. We will use this path through $sccG$ to form a witness to the non-empty intersection.

Consider the difference between $in(p_i)$ and $out(p_i)$. This difference contains the set of constraint observations that were consumed when passing through p_i. For each constraint FP_j, the difference corresponds to a sequence of consumed symbols, c_j. We then form the string $s_i = c_1 \shortparallel c_2 \ldots \shortparallel c_{|c|}$. This string is not necessarily a substring of any trace in $L_i(G)$. Fortunately, all symbols in s_i correspond to nodes or edge labels from G that are mutually reachable from one another (i.e., they are in the same strongly-connected component). We can form a new string, s'_i, that corresponds to a partial trace from $L_i(G)$, and contains s_i as a subsequence. Therefore, s'_i contains, in order, all symbols consumed by every constraint at p_i. If the resulting s'_i is empty, we update s'_i to contain a single arbitrary node from p_i.nodes.

For every adjacent pair (s'_i, s'_{i+1}), the final symbol in s'_i, α, belongs to strongly-connected component p_i (from our path p) and the first symbol from s'_{i+1}, β, belongs to strongly-connected component p_{i+1} (also from the path p). Thus, there exists some partial trace w_i from $L_i(G)$ where $w_{i_1} = \alpha$ and $w_{i_{|w_i|}} = \beta$. The following string is a witness to the non-empty intersection $L_i(G) \cap \bigcap_j FP_j \cap R$:

$$s'_1 \shortparallel w_1 \shortparallel s'_2 \shortparallel w_2 \shortparallel \ldots \shortparallel s'_{|s|}$$

□

Theorem 5 *If $L_i(G) \cap \bigcap_j FP_j \cap R$ is non-empty, then our data-flow analysis procedure reports Possible.*

Proof. If we know that the intersection $L_i(G) \cap \bigcap_j FP_j \cap R$ is non-empty, then there must exist some witness string w over $\widehat{N_{\mathbb{L}}}$ within the intersection. The context-insensitive condensation of G, $sccG$, induces a mapping from symbols in $\widehat{N_{\mathbb{L}}}$ to nodes in $sccG$. We can thus, given a witness string w, construct a sequence of substrings $w_1, w_2, \ldots, w_k$ where each w_i consists of exactly those symbols in w that mapped to the same strongly-connected component p_i.

The induced sequence of strongly-connected components $p = \langle p_1, p_2, \ldots, p_k \rangle$ describes a path through $sccG$. Note that all symbols appearing in all unreliable trace language vectors V_j appear in the witness w, so the `consume()` function will make no progress in any node $n \in sccG$ where n does not occur in p. That is, `consume`$(n, f) = f$ for n that do not occur in p. This means that for all $i \in 2 \ldots |p|$, $in(p_i) = out(p_{i-1})$. Further, no `merge()` operation ($\sqcap$) can possibly return $\perp$.

Since our data-flow problem iteratively computes the meet-over-all-paths solution, we simply need to show that, given path p over $sccG$, our data-flow approach consumes all of the observations so that $out(p_{|p|})$ is a vector of empty vectors. Concretely, $F_{p_{|p|}}(F_{p_{|p|-1}}(\ldots F_{p_1}(V)))$ is the vector of $|V|$ empty vectors. Consider a single constraint $V_j = \langle V_{j_1}, V_{j_2}, \ldots, V_{j_m} \rangle$. Each symbol V_{j_k} appears in exactly one strongly-connected component, p_i, from the path p. For a given pair of observations V_{j_a}, V_{j_b} with $a < b$, V_{j_a} appears in some p_α, and V_{j_b} appears in some p_β. Note that V_{j_a} appears before V_{j_b} in w, because the sequence of symbols in V_j is a subsequence of w. Since $sccG$ is a condensation of G, we therefore know that $\alpha \leq \beta$.

Now, all symbols from V occur somewhere within the path p, and moreover, they occur in order (as demonstrated). Recall (from Ohmann et al. [3, section 4.2]) that `consume`$(p_i, in(p_i))$ will greedily consume the longest prefix, x, of each vector in $in(p_i)$ where each symbol $x_n \in p_i$.nodes. Each requirement vector is consumed independently in parallel as we call `consume()` over each p_i, and, as previously demonstrated, our approach will never merge facts to $\perp$ in the given problem instance. This allows us to conclude that the data-flow procedure consumes all symbols from each $V_j \in V$, and $out(p_{|p|})$ is a vector of empty vectors as desired. We then report "Possible". □

6. Conclusion

Interpreting partial failure reports from post-deployment applications is challenging. Our system [3] allows users to pose control-flow queries to answer questions about a program's failing execution. We use formal-language representations of a program's control flow and various failure report constraints to check if a query is *Possible* to satisfy on any run consistent with the failure constraints. If all constraints and the user query are expressible as unreliable trace languages, we can answer queries remarkably efficiently (in polynomial time).

In this document, we provide supporting proofs for this system. Specifically, we first show that our calling-context-insensitive query recovery approach safely approximates the context-sensitive result. Then, we prove that the unreliable trace languages are a bounded class, by proving that two small extensions to UTL result in NP-hard query recovery problems. Finally, we prove partial correctness for our approach to query recovery using UTL constraints from Ohmann et al. [3, section 4.2].

Acknowledgments

This research was supported in part by DoE contract DE-SC0002153 and NSF grants CCF-0953478, CCF-1217582, CCF-1318489, CCF-1320854, and CCF-1420866. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] L. D'Antoni and R. Alur. Symbolic visibly pushdown automata. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria*, volume 8559 of *Lecture Notes in Computer Science*, pages 209–225. Springer, 2014. ISBN 978-3-319-08866-2. URL `http://dx.doi.org/10.1007/978-3-319-08867-9_14`.

[2] R. M. Karp. Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. ISBN 0-306-30707-3. URL `http://www.cs.berkeley.edu/~luca/cs172/karp.pdf`.

[3] P. Ohmann, A. Brooks, L. D'Antoni, and B. Liblit. Control-flow recovery from partial failure reports. In M. Vechev, editor, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona, Spain, June 2017.

[4] K. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theor. Comput. Sci.*, 16:187–198, 1981. URL `http://dx.doi.org/10.1016/0304-3975(81)90075-X`.

[5] M. Veanes. Applications of symbolic finite automata. In *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada*, volume 7982 of *Lecture Notes in Computer Science*, pages 16–23. Springer, 2013. ISBN 978-3-642-39273-3. URL `http://dx.doi.org/10.1007/978-3-642-39274-0_3`.

[6] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *3rd International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France*, pages 498–507. IEEE, 2010. ISBN 978-0-7695-3990-4. URL `http://dx.doi.org/10.1109/ICST.2010.15`.