On Strings in Software Model Checking

Hossein Hojjat[1], Philipp Rümmer[2], and Ali Shamakhi[1]

[1] University of Tehran, Iran
[2] Uppsala University, Sweden

Abstract. Strings represent one of the most common and most intricate data-types found in software programs, with correct string processing often being a decisive factor for correctness and security properties. This has led to a wide range of recent research results on how to analyse programs operating on strings, using methods like testing, fuzzing, symbolic execution, abstract interpretation, or model checking, and, increasingly, support for strings is also added to constraint solvers and SMT solvers. In this paper, we focus on the verification of software programs with strings using model checking. We give a survey of the existing approaches to handle strings in this context, and propose methods based on algebraic data-types, Craig interpolation, and automata learning.

1 Introduction

The analysis of program operating on strings has received a lot of attention in the past years, motivated by the observation that correct string handling is crucial to achieve functional correctness, and that even innocent-looking mistakes related to strings (for instance, incorrect input validation or sanitisation) can open severe security vulnerabilities in programs [10]. In this paper, we consider the analysis of software programs with the help of model checking, and provide a survey of the methods used in model checkers to handle strings. We observe that several bounded model checkers and tools for symbolic execution use "native" methods for solving string constraints, in particular inbuilt string support in SMT solvers, whereas unbounded model checkers tend to represent strings using data-types like arrays and stay closer to the runtime implementation of strings. We then outline ongoing work to handle strings natively in the Horn clause-based software model checker JayHorn.

1.1 Strings in Programming Languages

Given a finite, non-empty alphabet Σ, strings are elements of the set Σ^* or finite sequences of characters over Σ. In practice, alphabets are, e.g., ASCII or Unicode. Relevant operations on strings include functions to access individual characters or substrings, to concatenate strings, to split strings, to compute the length of strings or the number of character occurrences, to check membership in regular or context-free languages, to replace all or some occurrences of substrings, or more generally transformations like sanitisation or encoding/decoding.

In programming languages, strings are partly given the status of a primitive data-type with inbuilt notation for literals (`"..."`), but the full set of string operations is typically provided through libraries, such as `string.h` in C, and `java.lang.String` and related classes in Java. The internal representation of strings as a character array is fully exposed in C, but usually hidden in more high-level languages.

Strings are in programs often used to store *data* such as addresses, usernames, or passwords, whose correct processing is critical. Strings can also represent *code,* for instance when interfacing databases (SQL commands) or in the context of the web (JavaScript embedded in HTML), leading to the possibility of injection attacks when a programs fails to correctly isolate code from data [10].

2 Survey of Existing Methods for String Analysis

In this section, we focus on strings in model checking while only touching upon some of the methods in other areas. For a more complete survey of string methods we refer the reader to the recent book [10].

2.1 Bounded Methods

Bounded analysis methods, for instance, bounded model checking or symbolic execution, typically only have to check *satisfiability (SAT)* of constraints extracted from a program, usually testing path feasibility. In our case, such constraints will contain variables ranging over strings. SAT checks on string constraints are at this point supported relatively well by existing constraint and SMT solvers (as a result of extensive research over the past years) and string theories have in particular been added to state-of-the-art SMT solvers like Z3 [16] and CVC4 [7]. There is also a larger number of dedicated string solvers, for instance ABC [6], Hampi [20], Kaluza [37], Norn [3], Ostrich [11], Sloth [26], Trau [1], Z3-str [41]. Scalability to handle real-world constraints and support for more complex string operations (e.g., transduction) are still a concern with the existing solvers, however.

As a representative set of state-of-the-art software model checkers, we survey the tools that participated at SV-COMP 2019 [8], the most recent competition of automatic software verifiers. In the competition, 31 tools participated, of which 27 were verifiers for C and 4 for Java. It is observable that purely bounded analysis is applied by 7 of the C verifiers[3] and 3 of the Java verifiers,[4] while the other tools attempt exhaustive verification without imposing any bound on the number of execution steps (Section 2.2).

Following the actual C semantics of strings, the predominant approach applied by the bounded C verifiers is to consider strings as zero-terminated arrays,

[3] CBMC, CBMC-Path, Map2Check, Pinaka, VeriFuzz, Yogar-CBMC, Yogar-CBMC-Parallel

[4] JBMC, JPF, SPF

and rely on decision procedures for the theory of arrays (for instance, via encoding to Boolean SAT) to perform feasibility checks. There exists some initial work in CBMC to target native string solvers, but does not seem to be used in the competition versions.

The situation is different in the bounded Java verifiers, where the symbolic tools use native string solvers to analyse path constraints. JBMC [14] comes with its own string solver that works through an encoding to Boolean SAT, while SPF [36] can use multiple different string solvers as its back-end.

2.2 Unbounded Methods

In addition to SAT checks, unbounded (infinite-state) program verification methods also require artefacts like loop invariants or function summaries, which can be provided manually or be computed automatically. For the latter purpose, a wider range of techniques has been proposed that could be listed here; to get a full picture, we refer to the recent handbook on model checking [13]. A general observation, however, is that strings can be handled only by few of the existing invariant generation methods; in particular, to the best of our knowledge, no interpolation procedures are known for any (relevant) theory of strings.

The few invariant generation methods specifically supporting strings include the randomised search approach in [38], and the SAT-based automata learning approach in [2], which we employ in Section 3.3.

Software Model Checking. We survey again the tools that participated at SV-COMP 2019 [8]: 20 tools performing unbounded verification for C programs, and one model checker for unbounded Java verification. Like in the bounded case, most of the C model checkers see strings as character arrays, and execute string operations as code; this means that invariant generation relies on existing methods for the theory of arrays. 2LS encodes data structures (including string) using invariants describing heap configuration [33], PredatorHP models memory using Symbolic Memory Graph (SMG) and defines certain manipulations of zero-terminated strings over SMG [18], and SMACK models the behaviour of `string.h` functions. The handling of strings in the Java model checker JayHorn, an ongoing implementation effort, is discussed in Section 3.

Deductive Verification. In deductive verification systems, invariants and method contracts usually have to be provided manually, but their correctness is verified automatically. To be able to handle strings, deductive verification systems include axiomatic models of strings. For instance, in Dafny [31] strings are encoded as sequences, which are in turn mapped to arrays, together with a set of operations modelled using quantified axioms that are heuristically instantiated by the underlying SMT solver. The KeY system [4], a verification tool for Java programs, includes a formalisation of Java strings and the Java string constant pool in terms of algebraic data-types [9]. This formalisation partly inspires the techniques discussed in Section 3.2. Like [9], we propose to represent strings using ADTs while targeting the fully-automatic setting of a software model checker, including automatic invariant inference.

Static Analysis and Abstract Interpretation. A number of abstract domains have been proposed to analyse programs with strings, see for instance [10] for an overview. A related approach [12] translates Java programs to (data) flow graphs, extracts context-free grammars characterising the possible strings in the program, and then over-approximates those sets using regular languages.

Dedicated Analysis Methods. Several approaches exist to specifically analyse loops that iterate over or manipulate strings; such methods are usually restricted to loops of a particular syntactic shape, or to loops written in domain-specific languages. Bek is a language and system to write and analyse string sanitisers that internally uses symbolic transducers [27]. An extended version of Bek, named Bex, targets the more general case of string decoders [39]. A summarisation (or acceleration) method for string-manipulating loops is given in [40].

3 Towards String Handling in a Java Model Checker

We now describe ideas and techniques to handle strings natively in an unbounded model checker for Java. The work is inspired by the implementation of JayHorn tool [29], a Java verifier that works by translating Java bytecode to sets of constrained Horn clauses [23].

As observed in the previous section, and as with any other theory in software model checking, one of the main challenges with strings is the inference of inductive invariants. This aspect is particularly pronounced with strings, for which already decidability of SAT checks is sometimes open (depending on the precise set of operations considered [21]), and implementation even of known decision procedures can be hard. Logical methods used in other domains for invariant generation, for instance Craig interpolation [34] or abduction [17], have so far not been carried over for strings, to the best of our knowledge.

We consider two main paradigms to compute invariants in this setting:

- A *reduction-based* approach, in which string constraints are translated to algebraic data-types (ADTs), which can then be handled using known techniques, and are in particular amenable to Craig interpolation [24]. The reduction also requires an encoding of the string operations, which is in our setting done by formulating constrained Horn clauses, i.e., through an operational encoding. This is possible for all computable operations on strings, but does not always make it easy for an ADT solver to discover sufficiently general invariants.
- A *learning-based* approach, which performs an exhaustive search for Craig interpolants (as building blocks of inductive invariants) through SAT-based construction of finite-state automata [2].

Those two approaches have quite complementary properties. Reduction to other theories can in principle support all string operations, and handle the combination of strings with other theories (e.g., integers, arrays, or bit-vectors), but might not lead to useful predicates or invariants. The reduction approach is also similar in flavour to the representation of strings as arrays in existing software model checkers, though considering a different target theory, and using tailor-made operational encodings also of the string operations. Learning and systematic search can find concise predicates that pinpoint the reason why a program behaves correctly, but the approach might be computationally expensive, restricted to invariants of particular syntactic shape, and (depending on the algorithm used) difficult to combine with other theories. In our case, the learning procedure attempts to construct Craig interpolants that are regular expression membership constraints, which means that formulas like equality of two strings cannot be expressed or found.

3.1 Dealing with Implementation Artefacts

As a prerequisite for applying native string solving technology, it is necessary to bridge the gap between the programming language semantics of strings (in Java, the view of strings being instances of the class `java.lang.String`, and the string constant pool [22]) and the algebraic view on strings (strings constituting the set Σ^* of finite sequences over some alphabet Σ). The architecture of JayHorn offers a natural solution for this: deviating from the standard runtime implementation, object references in JayHorn are treated as *tuples* that consist of the object address (an integer), but also include other (immutable) information about an object [28]. For instance, a reference can store the precise dynamic type of the referenced object, the allocation site, constructor parameters, or values of immutable fields. The additional information contained in a reference has the purpose of increasing the expressive power of the class invariants used to represent heap data-structures.

This approach turns out to be particularly useful for boxed data-types like `java.lang.Integer`, since those classes are immutable and their contents do not change after object creation. This means that a reference to an object of `java.lang.Integer` can be defined to store the actual value (the boxed integer number) as well, using the native data-type for integers; since the boxed data can now be retrieved directly from the reference, without having to access any fields of the object, verification with boxed data becomes very similar to the handling of native data-types and local variables.

The same encoding can be used for strings: the reference tuple pointing to a `java.lang.String` object can be defined to contain the actual string contents as one of the components, represented using a native data-type, for instance, an ADT as in Section 3.2. Since the semantics of most of the string operations (for instance, `String.equals` and `String.concat`) can be modelled purely in terms of the string contents, this means that programs can then be analysed treating strings as a native data-type, assuming the idealised algebraic semantics of the string stored in the reference tuple.

3.2 Strings as an Algebraic Data-type

Algebraic data-types (with fully-free constructors) is a theory increasingly supported by Horn solvers, for instance by Eldarica [25], Spacer [30], and a version of VeriMAP [15]. While ADT support in the mentioned solvers is still somewhat limited and an active area of research (e.g., in case of Eldarica, only quantifier-free solutions are computed), ADTs are significantly simpler to handle than a full theory of strings, since methods like Craig interpolation and quantifier elimination are available.[5]

We define the theories of recursive algebraic data types (ADTs) as it is done in [24]. The signature of an ADT is defined by a sequence $\sigma_1, \ldots, \sigma_k$ of sorts and a sequence $f_1, \ldots, f_m$ of constructors. The type of an n-ary constructor is of the form $f_i : \sigma_1 \times \cdots \times \sigma_n \to \sigma_0$. Zero-ary constructors are also called constants. In addition to constructors, formulas over ADTs can use *variables* (with some type from the sorts $\{\sigma_1, \ldots, \sigma_k\}$); *selectors* f_i^j (which extract the j^{th} argument of an f_i-term) and *testers* is_{f_i} (which determine whether a term is an f_i-term).

ADTs enable a natural representation of strings as lists of characters. Here, nil is a constant, cons is a binary constructor, and $\mathsf{Character}$ is a sort:

$$\mathsf{String} \ ::= \ \mathsf{nil} \mid \mathsf{cons}(\mathsf{Character}, \mathsf{String})$$

This representation still leaves a number of choices open; exploration of this space is ongoing work, so that we only discuss the parameters in the scope of this paper, without evaluating the implications experimentally.

Encoding choice 1: *The character domain.* In our current implementation in JayHorn, the Character is a synonym for the mathematical integers, which are handled well by most Horn solvers. This domain does obviously not model ASCII or Unicode characters accurately, and might lead to spurious verification counterexamples; a more precise encoding could be using bit-vectors or an interval of the integers.

Encoding choice 2: *The character order.* The encoding of lists leaves open in which order the characters of a string should be stored: starting with the first character or starting with the last (or choosing an order individually for each string variable). The current JayHorn implementation uses the more natural order of storing the first string character as the first element of a list; but given that it is more common in Java programs to *append* to strings, it is quite possible that reverse order would perform better for static analysis.

After choosing the string representation, the Java API string operations have to be defined. One approach for this would be to execute the bytecode implementing the methods, for instance the methods of `java.lang.String`. This would not yield the most efficient definition for the purpose of model checking, however, since the bytecode would assume the internal representation of strings as character arrays, running counter to the chosen algebraic list representation. In the context of JayHorn, a more efficient path is to encode each string operation

[5] In this sense, ADTs also have better properties than the theory of arrays.

```
package java.lang;
public class String {
  [...]
  public String concat(String that) { [...] };
  [...]
}
```

Fig. 1. The Java string concatenation method

Recursive encoding $\mathcal{H}_{rec}$:

$$\begin{aligned} C_{rec}(\mathsf{nil}, x, x) &\leftarrow \mathit{true} \\ C_{rec}(\mathsf{cons}(c,x), y, \mathsf{cons}(c,z)) &\leftarrow C_{rec}(x,y,z) \end{aligned}$$

Recursive encoding with pre-condition $\mathcal{H}_{prec}$:

$$\begin{aligned} C^{post}_{prec}(\mathsf{nil}, x, x) &\leftarrow C^{pre}_{prec}(\mathsf{nil}, x) \\ C^{pre}_{prec}(x, y) &\leftarrow C^{pre}_{prec}(\mathsf{cons}(c,x), y) \\ C^{post}_{prec}(\mathsf{cons}(c,x), y, \mathsf{cons}(c,z)) &\leftarrow C^{pre}_{prec}(\mathsf{cons}(c,x), y) \wedge C^{post}_{prec}(x,y,z) \end{aligned}$$

Iterative encoding $\mathcal{H}_{it}$:

$$\begin{aligned} C^{1}_{it}(\bar{z}, x, \mathsf{nil}, y) &\leftarrow C^{entry}_{it}(\bar{z}, x, y) \\ C^{1}_{it}(\bar{z}, a, \mathsf{cons}(c,b), y) &\leftarrow C^{1}_{it}(\bar{z}, \mathsf{cons}(c,a), b, y) \\ C^{2}_{it}(\bar{z}, b, y) &\leftarrow C^{1}_{it}(\bar{z}, \mathsf{nil}, b, y) \\ C^{2}_{it}(\bar{z}, a, \mathsf{cons}(c,b)) &\leftarrow C^{2}_{it}(\bar{z}, \mathsf{cons}(c,a), b) \\ C^{exit}_{it}(\bar{z}, r) &\leftarrow C^{2}_{it}(\bar{z}, \mathsf{nil}, r) \end{aligned}$$

Table 1. Different Horn encodings of concatenation of two strings

using a set of Horn clauses tailored to static analysis. As a case study in the scope of this paper, we consider the method to perform concatenation of two strings (Fig. 1). Other Java string operations can be handled in a similar way.

Encoding choice 3: *The concatenation function.* Table 1 shows some of the different encodings of the concatenation function as a set of constrained Horn clauses, operating on the ADT string representation: using a total function defined recursively, and represented using a summary predicate C_{rec}; using a partial function defined recursively by a summary predicate C^{post}_{prec} and a domain predicate C^{pre}_{prec}; and using a purely iterative encoding with entry predicate C^{entry}_{it} and exit predicate C^{exit}_{it}.

A clause with a concatenation constraint on natively represented strings,

$$H \leftarrow z = \mathsf{concat}(x, y) \wedge B(\bar{a})$$

can then be translated using the different encodings, leading to three different but equisatisfiable sets of clauses:

$$\{H \leftarrow C_{rec}(x, y, z) \wedge B(\bar{a})\} \cup \mathcal{H}_{rec} \tag{1}$$

$$\{H \leftarrow C^{post}_{prec}(x, y, z) \wedge B(\bar{a}),\ C^{pre}_{prec}(x, y) \leftarrow B(\bar{a})\} \cup \mathcal{H}_{prec} \tag{2}$$

$$\{H \leftarrow C^{exit}_{it}(\bar{a}, z),\ C^{entry}_{it}(\bar{a}, x, y) \leftarrow B(\bar{a})\} \cup \mathcal{H}_{it} \tag{3}$$

In the last encoding $\mathcal{H}_{it}$, the arity of the predicates has to be adjusted so that all variables $\bar{a}$ occurring in the clause body $B(\bar{a})$ can be passed through.

Only experiments can tell which of those encodings performs best in a software model checker. Initial results indicate that the iterative encoding, although it requires the largest number of clauses, might be easiest to handle for existing Horn solvers, probably because only linear clauses are generated.

In cases where the length of the left string x is known to be bounded (and small), it is, of course, most efficient to unwind the recursive/iterative definition of the concatenation function sufficiently often.

Encoding choice 4: *Clause sharing.* In the iterative version of concatenation, it is always necessary to introduce fresh predicates and clauses $\mathcal{H}_{it}$ for each occurrence of concatenation `concat` in a program. This is not the case for the recursive versions, however, where the same predicates and clauses could be used for multiple occurrences of `concat`. Whether such clauses sharing has advantages for Horn solving is so far unclear, however.

Encoding choice 5: *Ghost data.* In addition to just working with the string contents represented using an ADT, it can be meaningful to also explicitly pass around ghost data obtained by applying some homomorphism to string values. For instance, the *length* of a string is a feature that is frequently useful for invariants; the function that maps a string to its length is a homomorphism of the concatenation function, and the clauses shown in Table 1 can easily be augmented to keep track of string length as well.

3.3 Learning Invariants over Strings

The encoding of strings using ADTs is quite flexible, and can be expected to work well when the correctness of a program can be shown using invariants on the level of ADTs: that means, using quantifier-free formulas that talk about a finite number of characters of the involved strings. Depending on the applied Horn solver, and the encoding choices with respect to ghost data, also invariants are feasible that can be expressed using recursive functions like the string length function.[6] However, ADTs do not suffice for programs that demand more intricate invariants about strings; for instance, the statement that an unbounded string only contains characters in the range `a-z`.

We propose the use of learning-based interpolation, as defined in [2], to find such more expressive invariants. Interpolation is used by many Horn solvers to

[6] For instance, Eldarica has built-in support for the ADT size function, which corresponds to string length.

construct building blocks for invariants. A *(binary) interpolation problem* is a conjunction of formulas $A[\overline{x}_A, \overline{x}] \wedge B[\overline{x}, \overline{x}_B]$ over disjoint variables $\overline{x}_A$, $\overline{x}_B$ local to A, B and common variables $\overline{x}$. An *interpolant* is a formula $I[\overline{x}]$ over the common variables such that $A[\overline{x}_A, \overline{x}] \Rightarrow I[\overline{x}]$ and $B[\overline{x}, \overline{x}_B] \Rightarrow \neg I[\overline{x}]$ hold.

In [2], it is assumed that $\bar{x} = \langle x_1, x_2, \ldots, x_n \rangle$ only contains string variables; a SAT solver is then used to systematically search for interpolants of the form

$$I[\overline{x}] \quad = \quad x_1 | x_2 | \cdots | x_n \in \mathcal{R} \tag{4}$$

where $|$ is a fresh separator character, and $\mathcal{R}$ a regular expression. $\mathcal{R}$ is for the search represented as a finite-state automaton using a set of Boolean variables. The search procedure itself uses a refinement loop in which the SAT solver guesses interpolant candidates, and a string solver checks the correctness of the candidates. Counterexamples produced by the string solver are used to refine the Boolean constraints. The procedure, therefore, has a lot of similarities with methods in syntax-guided synthesis [5], and could be generalised to interpolant patterns other than (4); it could also be changed to compute inductive invariants instead of just interpolants directly.

SAT-based learning has in the past also been used for a number of related applications, for instance to compute finite-state automata describing regions or strategies of games on infinite graphs [35, 32], or to synthesise transition systems that satisfy given LTL specifications [19]. This illustrates the flexibility of this form of learning; the challenge, however, is usually scalability, since a SAT solver essentially carries out a systematic search over all automata up to a certain size.

In practice, it appears most useful to combine the ADT-based method from Section 3.2 with the learning method. This could be done, for instance, by using the ADT method by default, but switching to the learning method when the computed ADT interpolants start to contain too complex ADT expressions. As a further criterion, when analysing programs that combine strings and other data-types (the most common case), it should be checked prior to starting the learning process whether the conjunction $A[\overline{x}_A, \overline{x}] \wedge B[\overline{x}, \overline{x}_B]$ is unsatisfiable for reasons pertaining to strings. This is a necessary (though not sufficient) criterion for the existence of an interpolant of the form (4). To check whether strings are responsible for any inconsistency, the common non-string variables in $\bar{x}$ can be renamed to local variables x'_i in A and x''_i in B.

4 Conclusions

We have given a survey of string handling in software model checkers, and proposed a combination of methods for model checking of Java programs. The paper presents work in progress, and at the moment the impact of the different design and encoding choices has not been evaluated experimentally yet; we do believe, however, that the outlined combination of string methods can significantly improve the usability of a Java model checker like JayHorn.

Acknowledgements. This research is supported by the Swedish Research Council (VR) under grant 2018-04727, and by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011).

References

1. P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer. Trau: SMT solver for string constraints. In *FMCAD*. IEEE, 2018.
2. P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV*. Springer, 2014.
3. P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In *CAV*. Springer, 2015.
4. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification — The KeY Book — From Theory to Practice*. Springer, 2016.
5. R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Dependable Software Systems Engineering*. IOS Press, 2015.
6. A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *CAV*. Springer, 2015.
7. C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*. Springer, 2011.
8. D. Beyer. Automatic verification of C and Java programs: SV-COMP 2019. In *TACAS*. Springer, 2019.
9. R. Bubel, R. Hähnle, and U. Geilmann. A formalisation of Java strings for program specification and verification. In *SEFM*. Springer, 2011.
10. T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin. *String Analysis for Software Verification and Security*. Springer, 2017.
11. T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. *PACMPL*, (POPL), 2019.
12. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*. Springer, 2003.
13. E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.
14. L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík. JBMC: A bounded model checking tool for verifying Java bytecode. In *CAV*. Springer, 2018.
15. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Solving Horn clauses on inductive data types without induction. *TPLP*, (3-4), 2018.
16. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*. Springer, 2008.
17. I. Dillig, T. Dillig, B. Li, and K. L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA*. ACM, 2013.
18. K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In *SAS*. Springer, 2013.
19. P. Faymonville, B. Finkbeiner, M. N. Rabe, and L. Tentrup. Encodings of bounded synthesis. In A. Legay and T. Margaria, editors, *TACAS*, volume 10205 of *LNCS*, pages 354–370. Springer, 2017.

20. V. Ganesh, A. Kiezun, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A string solver for testing, analysis and vulnerability detection. In *CAV*. Springer, 2011.
21. V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard. What is Decidable about Strings? Technical Report MIT-CSAIL-TR-2011-006, 03 2011.
22. J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.
23. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*. ACM, 2012.
24. H. Hojjat and P. Rümmer. Deciding and interpolating algebraic data types by reduction. In *SYNASC*. IEEE Computer Society, 2017.
25. H. Hojjat and P. Rümmer. The ELDARICA Horn solver. In *FMCAD*. IEEE, 2018.
26. L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, (POPL), 2018.
27. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.
28. T. Kahsai, R. Kersten, P. Rümmer, and M. Schäf. Quantified heap invariants for object-oriented programs. In *LPAR*. EasyChair, 2017.
29. T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. JayHorn: A framework for verifying Java programs. In *CAV*. Springer, 2016.
30. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, (3), 2016.
31. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*. Springer, 2010.
32. A. W. Lin and P. Rümmer. Liveness of randomised parameterised systems under arbitrary schedulers. In S. Chaudhuri and A. Farzan, editors, *CAV*, volume 9780 of *LNCS*, pages 112–133. Springer, 2016.
33. V. Malík, S. Marticek, P. Schrammel, M. K. Srivas, T. Vojnar, and J. Wahlang. 2LS: Memory safety and non-termination - (competition contribution). In *TACAS*. Springer, 2018.
34. K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*. Springer, 2003.
35. D. Neider and U. Topcu. An automaton learning approach to solving safety games over infinite graphs. In M. Chechik and J. Raskin, editors, *TACAS*, volume 9636 of *LNCS*, pages 204–221. Springer, 2016.
36. Y. Noller, C. S. Pasareanu, A. Fromherz, X. D. Le, and W. Visser. Symbolic pathfinder for SV-COMP - (competition contribution). In *TACAS*. Springer, 2019.
37. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE*. IEEE Computer Society, 2010.
38. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, (3), 2016.
39. M. Veanes. Symbolic string transformations with regular lookahead and rollback. In *PSI*. Springer, 2014.
40. X. Xie, Y. Liu, W. Le, X. Li, and H. Chen. S-looper: automatic summarization for multipath string loops. In *ISSTA*. ACM, 2015.
41. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based string solver for web application analysis. In *SIGSOFT*. ACM, 2013.