Precise Analysis of String Expressions

Aske Simon Christensen, Anders Møller[⋆], and Michael I. Schwartzbach

BRICS[⋆⋆], Department of Computer Science
University of Aarhus, Denmark
{aske,amoeller,mis}@brics.dk

Abstract. We perform static analysis of Java programs to answer a simple question: which values may occur as results of string expressions? The answers are summarized for each expression by a regular language that is guaranteed to contain all possible values. We present several applications of this analysis, including statically checking the syntax of dynamically generated expressions, such as SQL queries. Our analysis constructs flow graphs from class files and generates a context-free grammar with a nonterminal for each string expression. The language of this grammar is then widened into a regular language through a variant of an algorithm previously used for speech recognition. The collection of resulting regular languages is compactly represented as a special kind of multi-level automaton from which individual answers may be extracted. If a program error is detected, examples of invalid strings are automatically produced. We present extensive benchmarks demonstrating that the analysis is efficient and produces results of useful precision.

1 Introduction

To detect errors and perform optimizations in Java programs, it is useful to know which values that may occur as results of string expressions. The exact answer is of course undecidable, so we must settle for a conservative approximation. The answers we provide are summarized for each expression by a regular language that is guaranteed to contain all its possible values. Thus we use an upper approximation, which is what most client analyses will find useful.

This work is originally motivated by a desire to strengthen our previous static analysis of validity of dynamically generated XML documents in the JWIG extension of Java [4], but it has many other applications. Consider for example the following method, which dynamically generates an SQL query for a JDBC binding to a database:

```
public void printAddresses(int id) throws SQLException {
  Connection con = DriverManager.getConnection("students.db");
  String q = "SELECT * FROM address";
```

[⋆] Supported by the Carlsberg Foundation contract number ANS-1069/20
[⋆⋆] Basic Research in Computer Science (www.brics.dk),
funded by the Danish National Research Foundation.

```
    if (id!=0) q = q + "WHERE studentid=" + id;
    ResultSet rs = con.createStatement().executeQuery(q);
    while(rs.next()){ System.out.println(rs.getString("addr")); }
  }
```

The query is built dynamically, so the compiler cannot guarantee that only syntactically legal queries will be generated. In fact, the above method compiles but the query will sometimes fail at runtime, since there is a missing space between `address` and `WHERE`. In general, it may range from tedious to difficult to perform manual syntax checking of dynamically generated queries.

Our string analysis makes such derived analyses possible by providing the required information about dynamically computed strings. We will use the term *string operations* when referring to methods in the standard Java library that return instances of the classes `String` or `StringBuffer`.

Outline

Our algorithm for string analysis can be split into two parts:

– a *front-end* that translates the given Java program into a flow graph, and
– a *back-end* that analyzes the flow graph and generates finite-state automata.

We consider the full Java language, which requires a considerable engineering effort. Translating a collection of class files into a sound flow graph is a laborious task involving several auxiliary static analyses. However, only the front-end is language dependent, hence the string analysis can be applied to other languages than Java by replacing just the front-end. The back-end proceeds in several phases:

– The starting point is the flow graph, which gives an abstract description of a program performing string manipulations. The graph only has def-use edges, thus control flow is abstracted away. Flow graph nodes represent operations on string variables, such as concatenation or substring.
– The flow graph is then translated into a context-free grammar with one nonterminal for each node. Flow edges and operations are modeled by appropriate productions. To boost precision, we use a special kind of grammar in which string operations are explicitly represented on right-hand sides. The resulting grammar defines for each nonterminal the possible values of the string expression at the corresponding flow graph node.
– The context-free grammar is then transformed into a mixed left- and right-recursive grammar using a variant of the Mohri-Nederhof algorithm [11], which has previously been used for speech recognition.
– A program may contain many string expressions, but typically only few expressions, called *hotspots*, for which we actually want to know the regular language. For this reason, we introduce the *multi-level automaton (MLFA)*, which is a compact data structure from which individual answers may be extracted by need. Extensive use of memoization helps to make these computations efficient. An MLFA is a well-founded hierarchical directed acyclic graph (DAG) of nondeterministic finite automata.

All regular and context-free languages are over the Unicode alphabet, which we denote Σ. The core of the algorithm is the derivation of context-free grammars from programs and the adaptation of the Mohri-Nederhof algorithm [11], which provides an intelligent means of approximating a context-free language by a larger regular language. Naive solutions to this problem will not deliver sufficient precision in the analysis.

In programs manipulating strings, concatenation is the most important string operation — and in our analysis this operation is the one that we are able to model with the highest precision, since it is an inherent part of context-free grammars. We represent other string operations using less precise automata operations or character set approximations.

The translation from flow graph to multi-level automaton is linear-time. The extraction of a deterministic finite-state automaton (DFA) for a particular string expression is worst-case doubly exponential: one for unfolding the DAG and one for determinizing and minimizing the resulting automaton. In the case of a monovariant analysis, the flow graph obtained from a Java program is in the worst case quadratic in the size of the program, but for typical programs, the translation is linear.

We provide a Java runtime library with operations for selecting the expressions that are hotspots, casting a string expression to the language of a specified regular expression, and for probing regular language membership. This library serves several purposes: 1) It makes it straightforward to apply our analysis tool. 2) In the same way normal casts affect type checking, the "regexp" cast operation can affect the string analysis since the casts may be assumed to succeed unless cast exceptions are thrown. This is useful in cases where the approximations made by the analysis are too rough, and it allows explicit specification of assertions about strings that originate from outside the analyzed program. 3) Even without applying the string analysis, the operations can detect errors, but at runtime instead of at compile-time.

In Section 2, we describe related work and alternative approaches. Section 3 defines *flow graphs* as the connection between the front-end and the back-end of the analysis. In Section 4, a notion of context-free grammars extended with *operation productions* is defined, and we show how to transform flow graphs into such grammars. Section 5 explains how a variant of the Mohri-Nederhof algorithm can be applied to approximate the grammars by strongly regular grammars. These are in Section 6 translated into MLFAs that efficiently allow minimal deterministic automata to be extracted for the hotspots of the original program. Section 7 sketches our implementation for Java, and Section 8 describes examples of string analysis applications and a number of practical experiments.

We describe in what sense the algorithm is sound; however, due to the limited space, we omit proofs of correctness of the translation steps between the intermediate representations.

Contributions

The contributions in this paper consist of the following:

– Formalization of the general framework for this problem and adaptation of the Mohri-Nederhof algorithm to provide solutions.
– Development of the MLFA data structure for compactly representing the resulting family of automata.
– A technique for delaying the approximation of special string operations to improve analysis precision.
– A complete open source implementation for the full Java language supporting the full Unicode alphabet.
– A Java runtime library for expressing regular language casts and checks.
– Experiments to demonstrate that the implementation is efficient and produces results of useful precision.

Running Example

In the following sections we illustrate the workings of the various phases of the algorithm on this tricky program:

```
public class Tricky
{
  String bar(int n, int k, String op) {
    if (k==0) return "";
    return op+n+"]"+bar(n-1,k-1,op)+" ";
  }

  String foo(int n) {
    StringBuffer b = new StringBuffer();
    if (n<2) b.append("(");
    for (int i=0; i<n; i++) b.append("(");
    String s = bar(n-1,n/2-1,"*").trim();
    String t = bar(n-n/2,n-(n/2-1),"+").trim();
    return b.toString()+n+(s+t).replace(']',')');
  }

  public static void main(String args[]) {
    int n = new Random().nextInt();
    System.out.println(new Tricky().foo(n));
  }
}
```

It computes strings of the form `((((((((8*7)*6)*5)+4)+3)+2)+1)+0)` in a manner suitably convoluted to challenge our analysis.

2 Related Work

As far as we know, this straightforward problem of statically determining the possible values of string expressions has not really been explored before. We therefore choose to provide a discussion explaining why it cannot readily be solved using standard techniques: abstract interpretation or set constraints.

In both of those approaches, our work in obtaining a flow graph for string operations in Java programs would essentially have to be duplicated; the differences lie in the subsequent analysis of this flow graph.

Using the standard monotone framework for abstract interpretation [7, 13], the lattice of regular languages would be used to model abstract string values and all string operations would be given an abstract semantics. The standard fixed-point iteration over the flow graph would, however, fail to provide a solution since the lattice of regular languages has infinite height. Thus, we would at some stage be required to perform a widening step. Finding an intelligent way of generalizing a regular language into a useful larger language becomes the stumbling block for this approach. Note that the context-free language defined by a grammar is in fact obtained as the fixed-point of a series of finite approximants. Thus, our application of the Mohri-Nederhof algorithm may be viewed as a technique for jumping directly to a larger regular limit point.

Using set constraints [2], strings would be represented as linear terms with a constructor for each Unicode character. With this encoding, regular tree languages coincide with regular string languages. In the standard approach, each occurrence of an expression in the flow graph would be modeled by a set variable. String operations should then be modeled through appropriate set constraints on these variables. However, several of the operations we consider cannot be captured with any degree of precision by the permitted constraint operators. In particular, concatenation is not allowed: with such an operation, set constraints would no longer define regular tree languages [6]. Thus, we are returned to the problem that we solve in this paper: the flow graph inherently defines a context-free language, which must subsequently be given a regular approximation.

A different approach is described in [17], which introduces the λ^{re}-calculus where string expressions are typed by regular languages. This calculus allows in principle limited type inference (types of recursive functions must be given explicitly), but no algorithm is provided. Intriguingly, the paper refers to the Mohri-Nederhof algorithm as a possible venue for future work. In our approach, we use flow analysis rather than type inference. Thus, λ^{re} compares to our present work as XDuce [10] does to our previous work on JWIG [3].

There is of course much work in speech recognition related to the Mohri-Nederhof algorithm, but we refer to their paper [11] for this discussion.

In our previous work on JWIG [4], we used a simple string analysis that keeps track of finite sets of strings but widens to Σ^* at the slightest provocation. We believe that this simple algorithm has been used in many other places but has not been formally published.

Some work on machine learning is vaguely related to the problem we attack [14]: regular languages are inferred not from a flow graph but from a number of examples and answers to queries. We see no way of applying these techniques to our problem.

Other program analysis techniques also extract context-free grammars from programs [15], however, their grammars usually represent possible execution traces and never string values.

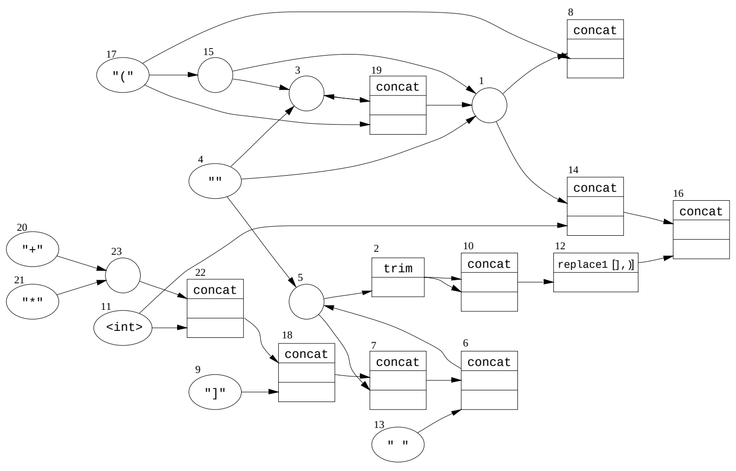
Finally, we note that another well-known combination of strings and program analysis is unrelated to our work. In [8] the problem is to detect memory errors in manipulations of C-like string pointers, and the actual characters occurring in strings are irrelevant to the results.

3 Definition of Flow Graphs

A *flow graph* captures the flow of strings and string operations in a program while abstracting everything else away. The nodes in such a graph represent variables or expressions, and the edges are directed *def-use edges* that represent the possible data flow [1]. More precisely, a flow graph consists of a finite set N of nodes of the following kinds:

– Init: construction of a string value, for instance from a constant or the `Integer.toString` method, and is associated a symbol *reg* that denotes a regular language $[\![reg]\!]$ representing the possible strings.
– Join: an assignment or other join location.
– Concat: a string concatenation.
– UnaryOp: a unary string operation, for instance `setCharAt` or `reverse`, with an associated symbol op_1 denoting a function $[\![op_1]\!] : \Sigma^* \to \Sigma^*$. Non-string arguments to string operations are considered to be part of the function symbols.
– BinaryOp: a binary string operation, for instance `insert`, with an associated symbol op_2 denoting a function $[\![op_2]\!] : \Sigma^* \times \Sigma^* \to \Sigma^*$.

Init nodes have no incoming edges, Join nodes may have an arbitrary number of incoming edges, each UnaryOp node has exactly one incoming edge, and each Concat and BinaryOp node has an ordered pair of incoming edges that represent flow into the respective arguments. Note that our notion of flow graphs is essentially a static single assignment (SSA) form where the Join nodes correspond to Φ functions. The flow graph for the `Tricky` example looks as follows:



The rightmost node corresponds to the single hotspot at `println`.

The semantics of a flow graph is defined as the least solution to a constraint system, similarly to the approach in [4]. The result is a map $F : N \to \Sigma^*$, such that $F(n)$ for every node n contains all possible values of the source program expression or variable that corresponds to n. The constraints are generated according to the following rules:

$F(n) \supseteq [\![reg]\!]$	for each Init node n
$F(n) \supseteq F(m)$	for each edge from a node m to a Join node n
$F(n) \supseteq F(m)F(p)$	for each Concat node n with edges from (m, p)
$F(n) \supseteq [\![op_1]\!](F(m))$	for each $\mathsf{UnaryOp}$ node n with edge from m
$F(n) \supseteq [\![op_2]\!](F(m), F(p))$	for each $\mathsf{BinaryOp}$ node n with edges from (m, p)

4 Construction of Context-Free Grammars

From the flow graph, we construct a special context-free grammar such that each flow graph node $n \in N$ is associated a nonterminal A_n. This grammar has the following property: For each node n, the language $\mathcal{L}(A_n)$ (that is, the language of the grammar with A_n as start nonterminal) is the same as $F(n)$.

First, we define a *context-free grammar with operation productions* as a grammar where the productions are of the following kinds:

$X \to Y$	[unit]
$X \to Y\ Z$	[pair]
$X \to reg$	[regular]
$X \to op_1(Y)$	[unary operation]
$X \to op_2(Y, Z)$	[binary operation]

where X, Y, and Z are nonterminals. The language of such a grammar is defined as one would expect: For a production $X \to reg$, X can derive all strings in $[\![reg]\!]$. For a unary operation $X \to op_1(Y)$, X can derive $[\![op_1]\!](\alpha)$ if Y can derive $\alpha \in \Sigma^*$, and similarly for binary operations. Note that the language is not necessarily context-free because of the operation productions.

The translation from flow graphs to grammars is remarkably simple: For each node n, we add a nonterminal A_n and a set of productions corresponding to the incoming edges of n:

– For an Init node with language reg, add $A_n \to reg$.
– For a Join node, add $A_n \to A_m$ for each node m with an edge to n.
– For a Concat node, add $A_n \to A_m\ A_p$ where m and p are the two nodes that correspond to the pair of incoming edges of n.
– For a $\mathsf{UnaryOp}$ node with operation op_1, add $A_n \to op_1(A_m)$ where m is the node having an edge to n.
– For a $\mathsf{BinaryOp}$ node with operation op_2, add $A_n \to op_2(A_m, A_p)$ where m and p are the two nodes that correspond to the pair of incoming edges of n.

The size of the resulting grammar is linear in the size of the flow graph. For the `Tricky` example it looks as follows:

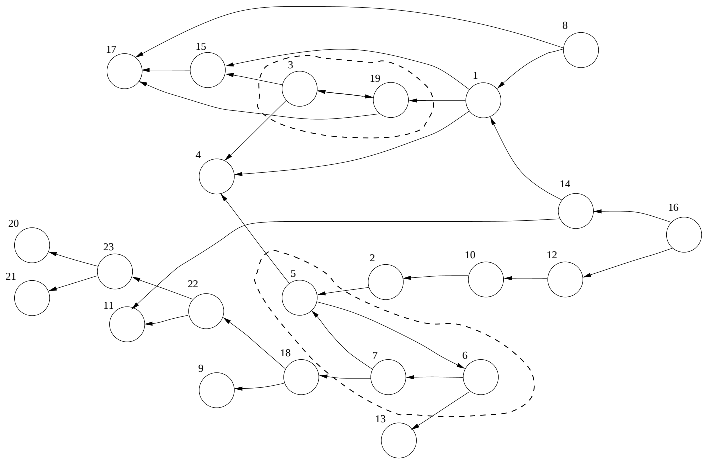
$$
\begin{array}{llll}
X_1 \rightarrow X_4 & X_1 \rightarrow X_{15} & X_1 \rightarrow X_{19} & X_2 \rightarrow \texttt{trim}(X_5) \\
X_3 \rightarrow X_{19} & X_3 \rightarrow X_{15} & X_3 \rightarrow X_4 & X_4 \rightarrow \texttt{""} \\
X_5 \rightarrow X_4 & X_5 \rightarrow X_6 & X_6 \rightarrow X_7\ X_{13} & X_7 \rightarrow X_{18}\ X_5 \\
X_8 \rightarrow X_1\ X_{17} & X_9 \rightarrow \texttt{"]"} & X_{10} \rightarrow X_2\ X_2 & X_{11} \rightarrow \texttt{<int>} \\
X_{12} \rightarrow \texttt{replace[],)]}(X_{10}) & X_{13} \rightarrow \texttt{" "} & X_{14} \rightarrow X_1\ X_{11} & X_{15} \rightarrow X_{17} \\
X_{17} \rightarrow \texttt{"("} & X_{16} \rightarrow X_{14}\ X_{12} & X_{18} \rightarrow X_{22}\ X_9 & X_{19} \rightarrow X_3\ X_{17} \\
X_{20} \rightarrow \texttt{"+"} & X_{21} \rightarrow \texttt{"*"} & X_{22} \rightarrow X_{23}\ X_{11} & X_{23} \rightarrow X_{20} \\
X_{23} \rightarrow X_{21} & & &
\end{array}
$$

The indices correspond to the node numbers in the flow graph. The regular language symbols are defined as expected: For example, $[\![\texttt{"+"}]\!] = \{\texttt{+}\}$ and $[\![\texttt{<int>}]\!]$ is specified by the regular expression `0|(-?[1-9][0-9]*)` (in Unix regexp notation).

5 Regular Approximation

We wish to approximate the grammar generated in the previous section with a regular grammar whose language contains that of the original. The central idea in our approach is based on the well-known fact that left-linear and right-linear context-free grammars effectively define regular languages [9]. This result extends to *strongly regular* grammars, as explained below.

As in the Mohri-Nederhof algorithm [11], we first find the strongly connected components of the grammar by viewing it as a graph with nonterminals as nodes and for each production an edge from the left-hand nonterminal to those on the right-hand side. For the `Tricky` example, the following graph is obtained:



Notice the resemblance with the flow graph shown in Section 3. The two marked node groups correspond to the nontrivial strongly connected components.

The Mohri-Nederhof approximation algorithm requires that for all operation productions, the nonterminals occurring on the right-hand side belong to a component different from the left-hand nonterminal. For this reason, we first eliminate all cycles that contain operation productions: For each unary operation op_1 being used, we require a *character set approximation* $[\![op_1]\!]_C : 2^{\Sigma} \to 2^{\Sigma}$ where $[\![op_1]\!]_C(S)$ contains the set of characters that may occur in $[\![op_1]\!](x)$ for a string $x \in S^*$, and similarly for binary operations. Using these approximations in a simple fixed point computation on the grammar, we find for each nonterminal X a set $C(X) \subseteq \Sigma$ containing all characters that may appear in the language of X. For each cycle we replace one operation production with $X \to r$ where r denotes the regular language $C(X)^*$. After this transformation, the strongly connected components are recomputed. For the `Tricky` example, neither the `trim` nor the `replace` operation occurs on a cycle.

A component M is *right-generating* if there exists a pair production $A \to B\ C$ such that A and B are in M, and M is *left-generating* if there exists a pair production $A \to B\ C$ such that A and C are in M. Each component now has one of four types: *simple* if it is neither right- nor left-generating, *left-linear* if it is right-generating but not left-generating, *right-linear* if it is left-generating but not right-generating, and *non-strongly-regular* otherwise. A context-free grammar is *strongly regular* if it has no non-strongly-regular components. The key observation of Mohri and Nederhof is that the desired approximation of the whole grammar can be obtained by a simple transformation of the non-strongly-regular components.

We adapt the Mohri-Nederhof algorithm to our form of grammar by transforming each non-strongly-regular component M into a right-linear one as follows: For each nonterminal A in M, add a fresh nonterminal A'. If A corresponds to a hotspot or is used in another component than M, then add a production $A' \to e$ where e denotes $\{\epsilon\}$. (Intuitively, A' represents substrings that may be recognized immediately after the substrings that are recognized by A.) Then replace all productions having A as left-hand side as follows:

$$\begin{array}{lll}
A \to X & \rightsquigarrow & A \to X\ A' \\
A \to B & \rightsquigarrow & A \to B, \quad B' \to A' \\
A \to X\ Y & \rightsquigarrow & A \to R\ A', \quad R \to X\ Y \\
A \to X\ B & \rightsquigarrow & A \to X\ B, \quad B' \to A' \\
A \to B\ X & \rightsquigarrow & A \to B, \quad B' \to X\ A' \\
A \to B\ C & \rightsquigarrow & A \to B, \quad B' \to C, \quad C' \to A' \\
A \to \mathit{reg} & \rightsquigarrow & A \to R\ A', \quad R \to \mathit{reg} \\
A \to op_1(X) & \rightsquigarrow & A \to R\ A', \quad R \to op_1(X) \\
A \to op_2(X,Y) & \rightsquigarrow & A \to R\ A', \quad R \to op_2(X,Y)
\end{array}$$

Here, A, B, and C are nonterminals within M, X and Y are nonterminals outside M, and each R is a freshly generated nonterminal. Intuitively, when considering a specific component M we may view the nonterminals outside M as terminals. Since all cycles with operation productions already have been eliminated, the operation arguments cannot belong to M.

As a result of this transformation, the component is now right-linear, its size is proportional to the original one, and it is constructed in linear time. In contrast to Mohri and Nederhof's application where the grammar always has one fixed start nonterminal, our application requires regular approximation for all nonterminals that correspond to hotspots. By construction, the language of a hotspot nonterminal in the original grammar is always a subset of the language of the same nonterminal in the approximated grammar.

We require for each unary operation op_1 being used a conservative regular approximation (e.g. in the form of an automaton operation) $[\![op_1]\!]_R : REG \rightarrow REG$, where REG is the family of regular languages — and similarly for the binary operations. When the operations used in the grammar are replaced by their approximating counterparts, the language of each nonterminal is guaranteed to be regular.

The restriction on adding the $A' \rightarrow e$ productions is essential for our application. As an example, consider the grammar:

$$\begin{array}{l} S \rightarrow T\ S \mid \texttt{a} \\ T \rightarrow S\ \texttt{+} \end{array}$$

which accepts strings of the form `a+a+...+a` and could be constructed from a tiny recursive Java method. Without the restriction, $T' \rightarrow e$ would be added, so the resulting grammar would accept, for example, the string `a+`, which is an unacceptably rough approximation. Instead, the presented algorithm produces an approximation corresponding to the regular expression $\texttt{a}(\texttt{+a})^*$, which is the best we could hope for.

The `Tricky` example contains one non-strongly-regular component consisting of $\{X_5, X_6, X_7\}$, and the approximation algorithm transforms the grammar into the following:

$$\begin{array}{llll}
X_1 \rightarrow X_4 & X_1 \rightarrow X_{15} & X_1 \rightarrow X_{19} & X_2 \rightarrow \texttt{trim(}X_5\texttt{)} \\
X_3 \rightarrow X_{19} & X_3 \rightarrow X_{15} & X_3 \rightarrow X_4 & X_4 \rightarrow \texttt{""} \\
X_5 \rightarrow X_4\ X'_5 & X_5 \rightarrow X_6 & X'_6 \rightarrow X'_5 & X_6 \rightarrow X_7 \\
X'_7 \rightarrow X_{13}\ X'_6 & X_7 \rightarrow X_{18}\ X_5 & X'_5 \rightarrow X'_7 & X_8 \rightarrow X_1\ X_{17} \\
X_9 \rightarrow \texttt{"]"} & X_{10} \rightarrow X_2\ X_2 & X_{11} \rightarrow \texttt{<int>} & X_{12} \rightarrow \texttt{replace[],)](}X_{10}\texttt{)} \\
X_{13} \rightarrow \texttt{" "} & X_{14} \rightarrow X_1\ X_{11} & X_{15} \rightarrow X_{17} & X_{16} \rightarrow X_{14}\ X_{12} \\
X_{17} \rightarrow \texttt{"("} & X_{18} \rightarrow X_{22}\ X_9 & X_{19} \rightarrow X_3\ X_{17} & X_{20} \rightarrow \texttt{"+"} \\
X_{21} \rightarrow \texttt{"*"} & X_{22} \rightarrow X_{23}\ X_{11} & X_{23} \rightarrow X_{20} & X_{23} \rightarrow X_{21} \\
X'_5 \rightarrow \texttt{""} & & &
\end{array}$$

with again X_{16} corresponding to the hotspot. For the `replace[],)]` operation, the regular approximation $[\![\texttt{replace[],)]}]\!]_R$ is defined as an automaton operation that transforms one automaton into another by replacing all '`]`' transitions by '`)`' transitions, where we use automata with partial transition functions. The character set approximation $[\![\texttt{replace[],)]}]\!]_C$ transforms one set of characters into another by removing '`]`' and adding '`)`' if '`]`' occurred originally. However, it is not used in this example since the operation does not occur in a cycle.

Notice the different sources of imprecision in the regular approximation: The Mohri-Nederhof transformation handles concatenation operations that occur in non-strongly-regular components. Other string operations are handled by the regular approximations specified as automata operations. The character set approximation, which is the most rough approximation in use, is used to break cycles of operation productions.

6 Multi-Level Finite Automata

As in [11], we extract automata from strongly regular grammars. However, since we consider the language of more than one nonterminal and have the special operation productions, we use a novel formalism, *multi-level finite automata (MLFA)*, with two important properties: 1) A strongly regular grammar can be translated into an equivalent MLFA in linear time, and 2) from the MLFA, we can efficiently extract a minimal deterministic (normal) automaton for each hotspot.

We define an MLFA to consist of a finite set of states Q and a set of transitions $\delta \subseteq Q \times T \times Q$ where T is a set of labels of the following kinds:

- reg
- ϵ
- (p, q)
- $\mathit{op}_1(p, q)$
- $\mathit{op}_2((p_1, q_1), (p_2, q_2))$

where each p and q are states from Q. There must exist a *level map* $\ell : Q \to \mathbb{N}$ such that:

- $(s, (p, q), t) \in \delta \Rightarrow \ell(s) = \ell(t) > \ell(p) = \ell(q)$,
- $(s, \mathit{op}_1(p, q), t) \in \delta \Rightarrow \ell(s) = \ell(t) > \ell(p) = \ell(q)$, and
- $(s, \mathit{op}_2((p_1, q_1), (p_2, q_2)), t) \in \delta \Rightarrow \ell(s) = \ell(t) > \ell(p_i) = \ell(q_i)$ for $i = 1, 2$.

That is, the states mentioned in a transition label are always at a lower level than the endpoints of the transition, and the endpoints are at the same level. The language $\overline{\mathcal{L}}$ of a single transition is defined according to its kind:

$$\begin{aligned}
&\overline{\mathcal{L}}(\mathit{reg}) = [\![\mathit{reg}]\!]\\
&\overline{\mathcal{L}}(\epsilon) = \{\epsilon\}\\
&\overline{\mathcal{L}}((p, q)) = \mathcal{L}(p, q)\\
&\overline{\mathcal{L}}(\mathit{op}_1(p, q)) = [\![\mathit{op}_1]\!]_R(\mathcal{L}(p, q))\\
&\overline{\mathcal{L}}(\mathit{op}_2((p_1, q_1), (p_2, q_2))) = [\![\mathit{op}_2]\!]_R(\mathcal{L}(p_1, q_1), \mathcal{L}(p_2, q_2))
\end{aligned}$$

Let $\overline{\delta}(q, x) = \{p \in Q \mid (q, t, p) \in T \wedge x \in \overline{\mathcal{L}}(t)\}$ for $q \in Q$ and $x \in \Sigma^*$, and let $\widehat{\delta} : Q \times \Sigma^* \to 2^Q$ be defined by:

$$\begin{aligned}
&\widehat{\delta}(q, \epsilon) = \overline{\delta}(q, \epsilon)\\
&\widehat{\delta}(q, x) = \{r \in Q \mid r \in \overline{\delta}(p, z) \ \wedge \ p \in \widehat{\delta}(q, y) \ \wedge \ x = yz \ \wedge \ z \neq \epsilon\} \quad \text{for } x \neq \epsilon
\end{aligned}$$

The language $\mathcal{L}(s,f)$ of a pair $s,f \in Q$ of start and final states where $\ell(s) = \ell(f)$ is defined as $\mathcal{L}(s,f) = \{x \in \Sigma^* \mid f \in \widehat{\delta}(s,x)\}$. This is well-defined because of the existence of the level map.

A strongly regular grammar produced in the previous section is transformed into an MLFA as follows: First, a state q_A is constructed for each nonterminal A, and additionally, a state q_M is constructed for each strongly connected component M. Then, for each component M, transitions are added according to the type of M and the productions whose left-hand side are in M. For a simple or right-linear component:

$$\begin{array}{lll}
A \rightarrow B & \rightsquigarrow & (q_A, \epsilon, q_B) \\
A \rightarrow X & \rightsquigarrow & (q_A, \Psi(X), q_M) \\
A \rightarrow X\ B & \rightsquigarrow & (q_A, \Psi(X), q_B) \\
A \rightarrow X\ Y & \rightsquigarrow & (q_A, \Psi(X), p),\ (p, \Psi(Y), q_M) \\
A \rightarrow \mathit{reg} & \rightsquigarrow & (q_A, \mathit{reg}, q_M) \\
A \rightarrow \mathit{op}_1(X) & \rightsquigarrow & (q_A, \mathit{op}_1(\Psi(X)), q_M) \\
A \rightarrow \mathit{op}_2(X,Y) & \rightsquigarrow & (q_A, \mathit{op}_2(\Psi(X), \Psi(Y)), q_M)
\end{array}$$

For a left-linear component:

$$\begin{array}{lll}
A \rightarrow B & \rightsquigarrow & (q_B, \epsilon, q_A) \\
A \rightarrow X & \rightsquigarrow & (q_M, \Psi(X), q_A) \\
A \rightarrow B\ X & \rightsquigarrow & (q_B, \Psi(X), q_A) \\
A \rightarrow X\ Y & \rightsquigarrow & (q_M, \Psi(X), p),\ (p, \Psi(Y), q_A) \\
A \rightarrow \mathit{reg} & \rightsquigarrow & (q_M, \mathit{reg}, q_A) \\
A \rightarrow \mathit{op}_1(X) & \rightsquigarrow & (q_M, \mathit{op}_1(\Psi(X)), q_A) \\
A \rightarrow \mathit{op}_2(X,Y) & \rightsquigarrow & (q_M, \mathit{op}_2(\Psi(X), \Psi(Y)), q_A)
\end{array}$$

Each p represents a fresh state. The function Ψ maps each nonterminal into a state pair: If A belongs to a simple or right-linear component M, then $\Psi(A) = (q_A, q_M)$, and otherwise $\Psi(A) = (q_M, q_A)$. The essence of this construction is the standard translation of right-linear or left-linear grammars to automata [9]. The construction is correct in the sense that the language $\mathcal{L}(A)$ of a nonterminal A is equal to $\mathcal{L}(\Psi(A))$. We essentially follow Mohri and Nederhof, except that they construct an automaton for a fixed start nonterminal and do not have the unary and binary operations.

Given a hotspot from the source program, we find its flow graph node n, which in turn corresponds to a grammar nonterminal A_n that is associated with a pair of states $(s,f) = \Psi(A_n)$ in an MLFA F. From this pair, we extract a normal nondeterministic automaton U whose language is $\mathcal{L}(s,f)$ using the following algorithm:

- For each state q in F where $\ell(q) = \ell(s)$, construct a state q' in U. Let s' and f' be the start and final states, respectively.
- For each transition (q_1, t, q_2) in F where $\ell(q_1) = \ell(q_2) = \ell(s)$, add an equivalent sub-automaton from q'_1 to q'_2: If $t = \mathit{reg}$, we use a sub-automaton whose language is $[\![\mathit{reg}]\!]$, and similarly for $t = \epsilon$. If $t = (p,q)$, then the sub-automaton is the one obtained by recursively applying the extraction

algorithm for $\mathcal{L}(p, q)$. If $t = op_1(p, q)$, the language of the sub-automaton is $[\![op_1]\!]_R(\mathcal{L}(p, q))$, and similarly for $t = op_2((p_1, q_1), (p_2, q_2))$.

This constructively shows that MLFAs define regular languages. The size of U is worst-case exponential in the size of F since the sub-automata may be duplicated. Since we subsequently determinize and minimize U, the size of the final DFA is worst-case doubly exponential, however, our experiments in Section 8 indicate that such blowups do not occur naturally. Our implementation uses memoization such that the automaton for a state pair (s, f) is only computed once. This reuse of computations is important for programs with many hotspots, especially if these involve common subcomputations.

We can now see the benefit of representing the unary and binary operations throughout all phases instead of, for instance, applying the character set approximation on all operations at an early stage: Those operations that in the flow graph do not occur in loops are modeled with higher precision than otherwise possible. For example, the `insert` method can be modeled quite precisely with an automaton operation, whereas that is difficult to achieve on the flow graph or grammar level.

To summarize, the whole translation from Java programs to DFAs is sound: Flow graphs are constructed such that they conservatively model the Java programs, and the regular approximation of grammars is also conservative. Both the translation from flow graphs to context-free grammars, the translation from strongly regular grammars to MLFAs, and the extraction of DFAs from MLFAs are exact. Together, this implies that if a Java program at some program point may produce a particular string during execution, then this string is guaranteed to be accepted by the automaton extracted for the program point.

7 Implementation for Java

Our implementation works for the full Java language, which makes the translation to flow graphs quite involved and beyond the scope of this paper. Hence, we settle for a rough sketch.

We use the Soot framework [18] to parse class files and compute interprocedural control flow graphs. We give a precise treatment of `String`, `StringBuffer`, and multidimensional arrays of strings. Using a null-pointer analysis, we limit proliferation of `null` strings. The construction of the flow graphs further requires a constant analysis, a liveness analysis, a may-must alias analysis, and a reaching definitions analysis – all in interprocedural versions that conservatively take care of interaction with external classes.

Our analysis tool is straightforwardly integrated with client analyses, such as the ones described in the next section. Furthermore, it is connected to the runtime library mentioned in Section 1 such that regexp casts are fed into the analysis and the designated hotspots are checked.

8 Applications and Experiments

We have performed experiments with three different kinds of client analyses.

Our motivating example is to boost our previously published tool for analyzing dynamically generated XML in the JWIG extension of Java [4]. This tool uses a primitive string analysis as a black box that is readily upgraded to the one developed in this work.

Another example is motivated by the Soot framework [18] that we use in our implementation. Here a string analysis can be used to improve precision of call graphs for Java programs that use reflection through the `Class.forName` method.

Finally, it is possible to perform syntax checking of expressions that are dynamically generated as strings, as in the example in Section 1.

In all three cases we provide a number of benchmark programs ranging from small to medium sized. Each benchmark contains many string expressions, but only few of those are hotspots. For each benchmark we report the number of lines of Java code, the total number of string expressions, the number of hotspots considered, the number of seconds to compute the MLFA, the total number of seconds to provide automata for all hotspots, and the maximal memory consumption (in MB) during this computation. The timings do not include time used by Soot to load and parse the class files, which typically takes 5-30 seconds for the programs considered. The accuracy of the analysis is explained for each kind of application. All experiments are performed on a 1 GHz Pentium III with 1 GB RAM running Linux.

8.1 Tricky

The `Tricky` benchmark is the example we followed in the previous sections, generating strings of the form: `((((((((8*7)*6)*5)+4)+3)+2)+1)+0)`. The analysis runs in 0.9 seconds and uses 26 MB of memory. The regular approximation that we compute is (in Unix regexp notation) `\(*<int>([+*]<int>\))*` where `<int>` abbreviates `0|(-?[1-9][0-9]*)`. This is a good result, but with a polyvariant analysis, the two calls to the `bar` method could be distinguished and the result further sharpened to `\(*<int>(\*<int>\))*(\+<int>\))*`.

8.2 JWIG Validity Analysis

The five smaller JWIG benchmarks are taken from the JWIG Web site. The three larger ones are a game management system (`MyreKrig`), a portal for a day care institution (`Arendalsvej`), and a system for management of the JAOO 2002 conference (`JAOO`). The hotspots correspond to places where strings are plugged into XML templates.

Example	Lines	Exps	Hotspots	MLFA	Total	Memory
`Chat`	67	86	5	0.597	0.603	34
`Guess`	77	50	4	0.577	0.581	34
`Calendar`	89	116	6	0.712	0.828	34
`Memory`	169	144	3	0.833	6.656	45
`TempMan`	323	220	9	0.845	0.890	33
`MyreKrig`	579	1,248	56	3.700	5.480	51
`Arendalsvej`	3,725	5,517	274	20.767	35.473	102
`JAOO`	3,764	9,655	279	39.721	86.276	107

The time and memory consumptions are seen to be quite reasonable. The precision is perfect for these ordinary programs, where only URL syntax, integers and scalar values must be checked to conform to the requirements of XHTML 1.0. We use the DSD2 schema language [12] which is expressive enough to capture these requirements on string values. Compared to our previous string analysis, we are able to validate more strings, such as dynamically built URL strings. The string analysis typically takes 10-20% of the total JWIG analysis time.

8.3 Reflection Analysis

These benchmarks are culled from the Web by searching for programs that import `java.lang.reflect` and selecting non-constant uses of `Class.forName` which also constitute the hotspots.

Example	Lines	Exps	Hotspots	MLFA	Total	Memory
`Switch`	21	45	1	1.155	1.338	25
`ReflectTest`	50	95	2	1.117	1.220	25
`SortAlgorithms`	54	31	1	0.997	1.214	25
`CarShop`	56	30	2	0.637	0.656	25
`ValueConverter`	1,718	438	4	4.065	4.127	36
`ProdConsApp`	3,496	1,909	3	12.160	13.469	80

Again, the time and memory consumptions are unremarkable. Without a client analysis, it is difficult to rate the precision. In simple cases like `SortAlgorithms` and `CarShop` we find the exact classes, and in some like `ValueConverter` we fail because strings originate from external sources.

8.4 Syntax Analysis

Many Java programs build string expressions that are externally interpreted, a typical example being SQL queries handled by JDBC, as the example in Section 1. At present, no static syntax checking is performed on such expressions, which is a potential source of runtime errors. We can perform such checking by approximating the allowed syntax by a regular subset which is then checked to be a superset of the inferred set of strings. For SQL, we have constructed a regular language that covers most common queries and translates into a DFA with 631 states.

The benchmarks below are again obtained from the Web. Most originate from database textbooks or instruction manuals for various JDBC bindings. The hotspots correspond to calls of `executeQuery` and similar methods.

Example	Lines	Exps	Hotspots	MLFA	Total	Memory	Errors	False Errors
`Decades`	26	63	1	0.669	1.344	27	0	0
`SelectFromPer`	51	50	1	1.442	1.480	27	0	0
`LoadDriver`	78	154	1	0.942	0.981	28	0	0
`DB2Appl`	105	59	2	0.736	0.784	27	0	0
`AxionExample`	162	37	7	0.800	1.008	29	0	0
`Sample`	178	157	4	0.804	1.261	28	0	0
`GuestBookServlet`	344	320	4	1.741	3.167	33	1	0
`DBTest`	384	412	5	1.688	2.387	31	1	0
`CoercionTest`	591	1,133	4	4.457	5.664	42	0	0

As before, the analysis runs efficiently. All hotspots except two are validated as constructing only correct SQL syntax, and encouragingly, the two remaining correspond to actual errors. The `GuestBookServlet` builds a string value with the construction `"'" + email + "'"`, where `email` is read directly from an input field in a Web form. Our tool responds by automatically generating the shortest counterexample:

```
INSERT INTO comments (id,email,name,comment,date) VALUES (0,''','','','')
```

which in fact points to a severe security flaw.

XPath expressions [5] are other examples where static syntax checking is desirable. Also, arguments to the method `Runtime.exec` could be checked to belong to a permitted subset of shell commands.

Finally, we could use our technique to attack the problem of format string vulnerabilities considered in [16]. In our approach, format strings that were tainted by outside values would be recognized possibly to evaluate to illegal strings. The precision of this approach is left for future work. Compared to the use of type qualifiers, our technique is more precise for string operations but it is less flow sensitive.

9 Conclusion

We have presented a static analysis technique for extracting a context-free grammar from a program and apply a variant of the Mohri-Nederhof approximation algorithm to approximate the possible values of string expressions in Java programs. The potential applications include validity checking of dynamically generated XML, improved precision of call graphs for Java programs that use reflection, and syntax analysis of dynamically generated SQL expressions.

Our experiments show that the approach is efficient and produces results of useful precision on realistic benchmarks. The open source implementation together with documentation and all benchmark programs are available at `http://www.brics.dk/JSA/`.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, November 1985.

[2] Alex Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[3] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X, October 2002.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 2003. To appear.

[5] James Clark and Steve DeRose. XML path language, November 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1999. Available from `http://www.grappa.univ-lille3.fr/tata/`.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '77*, pages 238–252, 1977.

[8] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Proc. 8th International Static Analysis Symposium, SAS '01*, volume 2126 of *LNCS*. Springer-Verlag, July 2001.

[9] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, April 1979.

[10] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB '00*, volume 1997 of *LNCS*. Springer-Verlag, May 2000.

[11] Mehryar Mohri and Mark-Jan Nederhof. *Robustness in Language and Speech Technology*, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.

[12] Anders Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.

[13] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, October 1999.

[14] Rajesh Parekh and Vasant Honavar. DFA learning from simple examples. *Machine Learning*, 44:9–35, 2001.

[15] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[16] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. 10th USENIX Security Symposium*, 2001.

[17] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. In *Proc. Workshop on Types in Programming, TIP '02*, 2002.

[18] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot – a Java optimization framework. In *Proc. IBM Centre for Advanced Studies Conference, CASCON '99.* IBM, November 1999.