On Development of Static Analysis Tools for String-Embedded Languages

Marat Khabibullin
St. Petersburg Academic University
194021, Khlopina Str 8/3
St. Petersburg, Russia
maratx387@gmail.com

Andrei Ivanov
St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia
ivanovandrew2004@gmail.com

Semyon Grigorev
St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia
rsdpisuy@gmail.com

ABSTRACT

Some programs can produce string expressions with embedded code in other programming languages while running. This embedded code should be syntactically correct as it is typically executed by some subsystem. A program in Java language that builds and sends SQL queries to the database it works with can be considered as an example. In such scenarios, languages like SQL are called string-embedded and ones like Java – host languages.

In spite of the fact such an approach of programs building is being replaced by alternative ones, for example by ORM and LINQ, string-embedding is still used in practice. Development and reengineering of the programs with string-embedded languages is complicated because the IDE and similar tools process the code embedded in strings as host language string literals and cannot provide the functionality to work with this code. To facilitate the development process, string-embedded code highlighting, completion, navigation and static errors checking would be useful. For the purposes of reengineering, embedded code metrics computation would be helpful.

Currently existing tools to string-embedded languages support only operate with one host language and a fixed set of string-embedded ones. Their functionality is often limited. Moreover, it is almost impossible or requires a substantial amount of work to add a support for both new host and string-embedded language. Attempts to extend their functionality often result in the same problem.

In this paper we present the platform which can be used for relatively fast and easy building of endpoint tools that provide a support for different string-embedded languages inside different host languages. The tools built for T-SQL and arithmetic expressions language embedding in C# are demonstrated as the examples of how the platform can be used.

ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

CCS Concepts

•**Software and its engineering** → **Automated static analysis; Software maintenance tools;** •**Theory of computation** → *Program analysis; Parsing;*

Keywords

String-embedded language, integrated development environment, IDE, approximation, control flow graph, CFG

1. INTRODUCTION

When writing certain kinds of programs on some programming language one frequently needs to construct a code on some other language. The program written on Java, C# or PHP constructing SQL queries and sending them to the database can be considered as an example. In such a program SQL queries can be constructed as string literals, both by writing the entire query inside the literal or by forming final query dynamically, i.e. by combining parts using string operations (concatenation, replace) and common language constructions (loops, conditional expressions). It is important to note that in the later case the parts are no need to be correct SQL expressions. Other examples of the approach in question include forming JavaScript code inside Java when writing web-applications, building of dynamic SQL queries using Dynamic-SQL, xml-files generation, etc. The language that manipulates strings containing code is called the host language. The language which code is written inside string literals is called string-embedded language.

It is useful to have the ability to perform static analysis on string-embedded languages. On the one hand, such analysis would make it possible to support string-embedded languages in IDEs by making syntax highlighting, static errors checking and other functions, previously only available for host languages, become available for string-embedded ones right inside string literals. On the other hand, programs writing approach under consideration is more and more frequently replaced by more advanced approaches. For example, speaking about embedded SQL such approaches include ORM (object-relational mapping) and LINQ (language integrated query). In spite of this fact, there exist many programs where string-embedded languages are used. These programs are still in use so they need a support and maintenance. In particular reengineering can be performed for them.

Static analysis can be useful during the reengineering pro-

cess at least in two ways. The first one is extracting some information about string-embedded code. Examples include estimating embedded program's structural complexity using such metrics as cyclomatic complexity [7], or, in the case of embedded SQL, estimating tables usage frequency in queries to check the possibility of restructuring the database the embedded program works with. The second way is automated transformation of the embedded code to move from the string embedding approach to alternative ones (for example, to move from string-embedded SQL to LINQ). It is important to note that the possibility of such transformations in general case is questionable even in theory. However, in particular cases such transformations can be performed automatedly.

The problem of string-embedded languages static analysis poses a number of challenges. Firstly, not all string expressions in a program contain embedded code and analyzer must be able to differentiate one expressions from another. Secondly, as it was mentioned before, string expressions can be formed dynamically. Thus, we need to construct them according to the operations they are formed with before we can start the analysis itself. Thirdly, the program in the host language in general case may produce the set of strings with embedded code so the analyzer must be able to build this set and represent it in a way convenient for the following analysis. Finally, the lexical and syntax analysis algorithms that can work not only with a single string but with the set of strings are needed.

Despite the utility of string-embedded languages static analysis, existing tools are mostly intended to support embedded code in IDEs and can hardly be used for reengineering problems solving. Moreover, the majority of the tools work only with specific host and string-embedded languages.

In this paper we present the platform for string-embedded languages support, which is a part of the YaccConstructor [6] project. The YaccConstructor is devoted to the experiments in the field of static analysis. The platform under discussion is intended as a basis for endpoint tools, making the process of its creation relatively easy and fast. Endpoint tools can be created for different host and string-embedded languages. The platform is designed to be extensible, so it is able to add different functions based on string-embedded language static analysis (from the syntax highlighting to metrics computation). This paper describes the platform and shows how to create endpoint tools.

2. EXISTING TOOLS

There are several tools that can process dynamically generated expressions. Many of them are oriented to support string-embedded languages in IDE. Such tools implement one of two basic approaches.

- Language inclusion checking. This approach answers the question if the strings generated by a program is included in the reference language described by user. This approach can be used to check expressions correctness, but other kinds of analysis cannot be performed based on the approach under consideration.

- Approximation of dynamically generated expression set followed by lexical and syntax analysis. By expressions set approximation one means the process of building a subset or a superset of the initial one. In the case of superset building it is called upper approximation. The advantages of the approach in question include flexibility: each step is performed independently, so existing algorithms implementations can be used for each step and can be replaced by other implementations if needed. As a result new tools can be created based on the existing ones.

Below a brief description of currently available tools is given.

2.1 JSA

The Java String Analyzer[1] [2] is a tool to analyze the flow of strings and string operations in Java programs. JSA checks if the regular approximation of the embedded language is included in the context-free description of the reference language. For every string expression JSA computes a finite-state automaton (FSA). It represents the approximate set of string expression's possible values that may be produced during the program execution. To build the FSA, the context-free grammar is constructed from the program's data-flow graph. This grammar is obtained by replacing every string variable with nonterminal, every string literal with terminal and every string operation with production rule. Then the context-free language the grammar produces is approximated by the regular one. As a result the tool returns the strings that are not in the reference language but that can be produced during the program execution.

2.2 PHPSA

PHP string analyzer[2] [8] is a static program analyzer that supports HTML and XML code embedded in PHP. The approach is based on the ideas used in JSA but to increase the analysis accuracy the context-free approximation is used instead of the regular one.

2.3 Alvor

Alvor[3] [1] is a plugin for Eclipse IDE that is intended for a static validation of SQL expressions embedded into Java code. Alvor performs the interprocedural code analysis, processes conditional statements, concatenation and assignment operations and reports lexical and syntax errors. If more than one error is detected only the first one is reported. The example of such a case is represented in Figure 1, with typos in lines 16 and 18. Only the typo in the line 16 is indicated by underlining. However, it should be mentioned that the plugin lacks loops and string operations other than concatenation support.

2.4 IntelliLang

IntelliLang[4] is a plugin for IntelliJ IDEA[5] IDE that extends its functionality in the field of string-embedded languages support. Plugin can highlight embedded code, perform code completion and for some languages (JavaScript,

[1]Java String Analyzer project site:http://www.brics.dk/JSA/

[2]PHP String Analyzer project site:http://www.score.cs.tsukuba.ac.jp/~minamide/phpsa/

[3]Alvor project site:https://bitbucket.org/plas/alvor

[4]IntelliLang is a plugin offering a number of features related to the processing of embedded languages. Site: https://www.jetbrains.com/idea/help/intellilang.html

[5]IntelliJ IDEA is an IDE for JVM-based development. Site: http://www.jetbrains.com/idea/

```
11 public static void executeSQL(Connection connection)
12   throws SQLException{
13     String sql = "select id, first_name from person where ";
14     int a = 2;
15     if(a > 3){
16       sql += " b => 1 ";
17     }else{
18       sql += " c => 1 ";
19     }
20     sql += " order by first_name";
21     PreparedStatement preparedStatement =
22             connection.prepareStatement(sql);
23     ResultSet result = preparedStatement.executeQuery();
24 }
```

Figure 1: Eclipse's text editor window with Alvor plugin

XML) detect errors. IntelliLang does not perform strings with embedded code search: user should manually specify the strings to be analyzed. This makes it inconvenient to use the plugin for some tasks especially for reengineering. IntelliJ IDEA's text editor window is illustrated in Figure 2. The method getHtml is marked by `@Language("HTML")` attribute which means this method returns the string containing html code as a result. Despite the closing tag `</body>` is missing in the 12 line, IntelliLang does not report the error as the plugin is not able to perform error checking for html language.

```
10     @Language("HTML")
11     public String getHtml(){
12         String body = "<body>" + "Hello";
13
14         String html = "<html>" + body + "</html>";
15         return html;
16     }
```

Figure 2: IntelliJ IDEA's text editor window with IntelliLang plugin

2.5 PhpStorm

PhpStorm[6] is an IDE for web-applications creating in PHP. PHP programs often contain embedded code in HTML, CSS, JavaScript, and SQL, and PhpStorm performs highlighting and code completion for it. However PhpStorm is not able to handle dynamically generated strings. Such an example can be seen in Figure 3. The variable `$string` contains html code, but it is not highlighted as its value is dynamically constructed. Moreover, PhpStorm does not report errors: even though the IDE highlighted the SQL code in the line 11, it did not indicate the error the query contains.

2.6 Varis

Varis [9] is a plugin for Eclipse that provides support of HTML, CSS, and JavaScript code embedded into PHP. The functionality includes code highlighting, completion, "jump to declaration", call graphs building for embedded JavaScript. Figure 4 illustrates code highlighting and completion functions.

[6]IDE for PHP programming language. Site: http://www.jetbrains.com/phpstorm/

```
1  <?php
2      $usualString = 'simple string';
3      $hello ='<html><body>Hello, world!</body></html>';
4      $string = '<';
5      if (cond)
6          $string .= 'html';
7      else
8          $string .= 'body';
9      $string .= '>';
10
11     $error = 'SELECT * FROM table1 FROM table1';
12 ?>
```

Figure 3: PHPStorm's text editor window

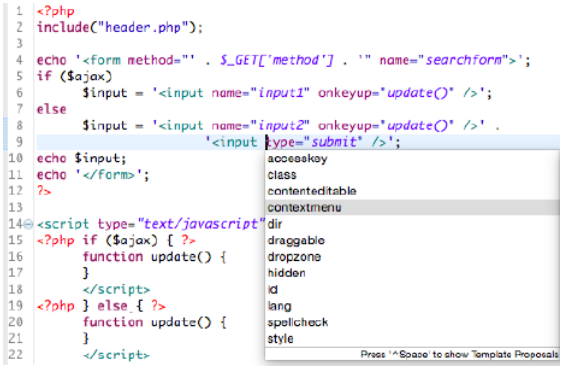


Figure 4: Eclipse's text editor window with Varis plugin

In conclusion one should note that along with specific drawbacks described tools either have limited functionality (JSA, PHPSA) which is difficult to extend, or support limited number of embedded languages (Alvor, PhpStorm and Varis). Furthermore, almost all the tools work with one specific host language.

3. THE PLATFORM

Any language can be used both as string-embedded or host language. For each combination it could be necessary to solve different tasks: from errors detection to additional information extraction and metrics computation. As there are substantial number of languages combinations and different types of tasks, our research is not aiming to create a tool that handles all possible scenarios. The main goal is to create the platform that simplifies the process of building endpoint tools for the certain languages and tasks. The similar approach can be seen in the tools for compilers development. Usually such tools include lexical and syntax generators and utility functions libraries to simplify the process of specific compiler creating.

Being complex and self-contained task, the host language analysis is assumed to be performed by some external tool. Moreover, this tool is assumed to return a source code's AST which contains the information needed for further steps as a result of it's work.

The platform is to perform the whole process of string-embedded languages handling. It includes the following steps:

- dynamically generated expressions can have a set of possible values so the approximation of the set is to be

constructed;

- lexical analysis is to be applied to the FSA that has been obtained after approximation;
- syntax analysis is to be performed resulting in the parse forest;
- parse forest analysis and semantics calculation is to be performed.

3.1 Architecture

The platform's architecture is shown in Figure 5.

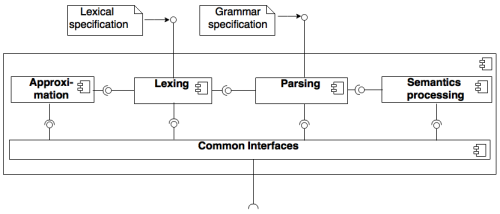


Figure 5: The platform's architecture

The Approximation component is responsible for building the set of strings with embedded code that can be produced by the host language program. The elements of that set are to be statically analysed. This set can be considered as a language where the words are the strings with embedded code. In general the language can be recursively enumerable. Many problems are undecidable for this class of languages that makes it hard to work with. One possible solution is to build an upper regular approximation of the language we have. In other words we build the regular language that contains all the words of the language we have and, possibly, some other words. It is important to note that the regular language must be as close as possible to the language it approximates. It means the language must keep the number of "extra" words to a minimum. Despite the fact we end up with the approximation of the initial set, working with a regular language makes the further analysis much easier. Moreover, regular languages can always be represented as FSA that is convenient to manipulate.

The Approximation component accepts a generalized CFG with some additional information as an input. As an output it produces the FSA that encodes the approximated set of strings with embedded code.

The lexing component consist of two parts. The first one is the lexer's generator which generates finite-state transducer(FST, [5]) from the provided lexical specification of the string-embedded-language. The second one is the interpreter which analyzes the data structure passed using generated FST. The lexing component accepts FSA over symbols alphabet and produces the FSA over alphabet of processed language tokens.

The parser generator is based on the RNGLR algorithm [11]. Using the grammar of the language processed the generator builds parse tables. Then the analyzer, which is implemented as a separate library, parses the FSA that was obtained after the lexical analysis. The result is a shared packed parse forest (SPPF [10]) which is a compact structure that allows to reuse common nodes along different ASTs. This structure can be used for the further processing: for additional information extraction or to implement such IDE functionality as "go to definition".

3.2 Approximation

The process of building the set of strings with embedded code can be divided into two steps:

- finding the string expressions with embedded code in the source code;
- building the set of possible values for found string expressions.

The first step is not implemented in our platform. This functionality is assumed to be implemented by the platform's user. The second step is performed by the platform and described in more detail below.

3.2.1 Upper regular approximation building

This section briefly describes the regular approximation building algorithm, presented in [13]. The paper [13] is devoted to the analysis of string expressions produced by PHP code. The analysis is aimed to find the expressions that can be used to perform malicious actions. We are interested in the algorithm presented because it guarantees building the upper approximation. Moreover, it handles most host language operations that can be used to form string expressions.

To begin with we introduce some definitions. *Target expression* – the string expression in the source code which possible values are to be built. *Target node* – the node in the control flow graph (CFG) containing target expression. *Data-dependency graph (DDG)* – directed graph that represents dependencies between instructions in the source code. Graph nodes correspond to expressions in the source code. The edge from the node `X` to the node `Y` indicates we need to calculate the expression in the node `X` before we can calculate the expression in the node `Y`.

As an example consider Listing 1 where the reference to the `"sql"` variable in the line 10 is chosen to be the target expression. Figure 6 shows the CFG for the code fragment from the listing where the node 16 is a target node. DDG for the `"sql"` reference is shown in Figure 7.

```
 1  Entries GetEntries(boolean cond)
 2  {
 3      string logMsg = "Selected";
 4      string sql = "SELECT * FROM ";
 5      if(cond)
 6        sql = sql + "Table1";
 7      else
 8        sql = sql + "Table2";
 9      Console.WriteLine(logMsg);
10      return Db.Execute(sql);
11  }
```

Listing 1: C# code fragment

Further the following agreement about finite-state automata will be used. It is assumed every automaton has an additional state that is not shown explicitly. This state is called a sink state. The sink state is not a final one and transitions for all symbols of the automaton's alphabet lead to the sink state itself. When some automaton is mentioned in the text or illustrated in a picture transitions will be defined only for some symbols of the alphabet. It is assumed that if the

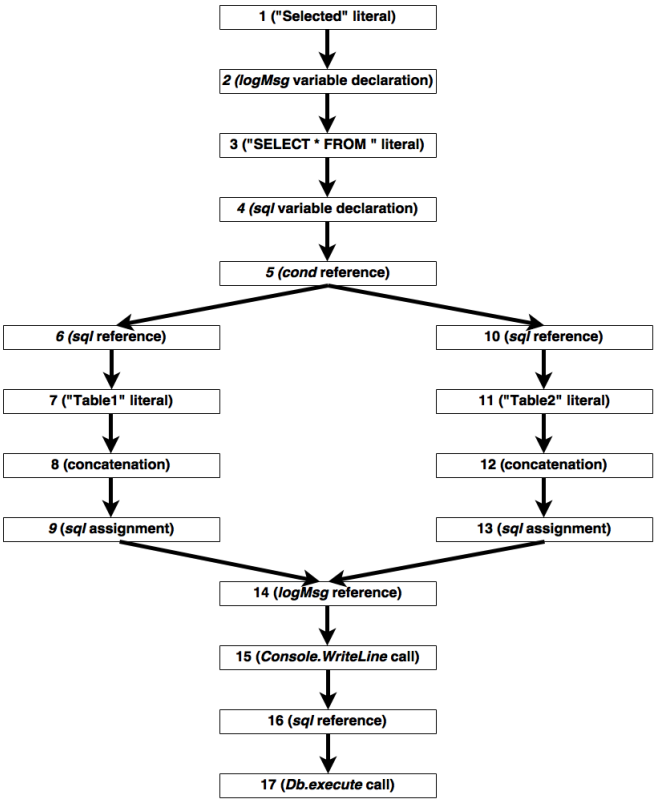


Figure 6: Control flow graph for the code fragment from the Listing 1

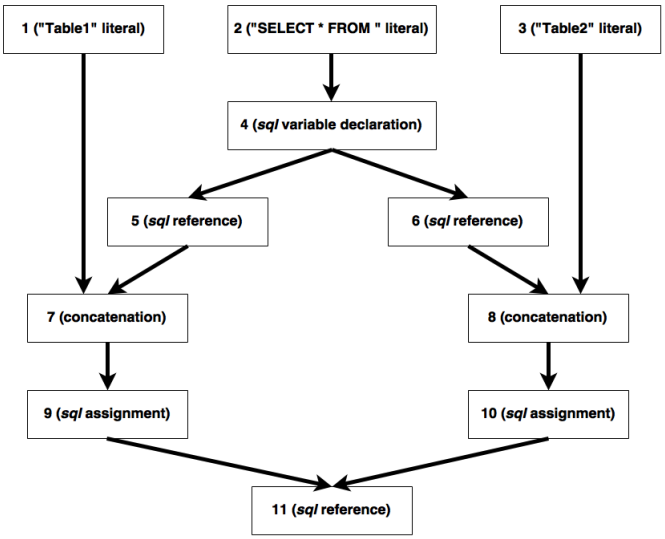


Figure 7: Data-dependency graph for the `"sql"` variable reference from the Listing 1

transition for the symbol `'a'` from the state `X` is not defined explicitly it leads to the sink state (see Figure 8). Sink states are omitted for simplicity.

Now we describe the regular approximation building algorithm presented in [13]. It consists of the following steps:

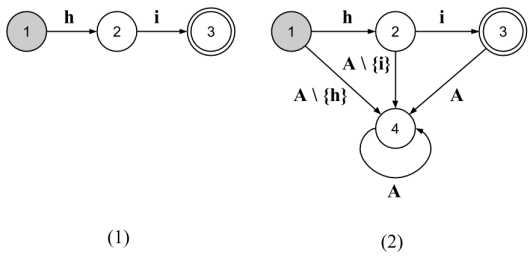


Figure 8: Simplified illustration of the automaton over the alphabet A (1) and corresponding fully determined one with sink state (2)

- control flow graph is created for the source code;
- based on the CFG and some target node in it the data-dependency graph is computed;
- finite state automata are constructed for the expressions in the DDG nodes. Automata are the upper approximations of the expressions' possible values sets. Automaton corresponding to the target node is returned as a result.

Let us see how exactly automata are built for the expressions in the nodes during the DDG traversal.

String literals. The FSA for the string literal is an automaton that accepts only this literal. The example for the literal `"hi"` is shown in Figure 8.

Strings concatenation. The concatenation operation is performed on two automata that correspond to the operation arguments. Based on these automata the resulting one is built. It accepts the strings that consist of prefix and suffix. The prefix is any string accepted by the first argument automaton and the suffix is any word accepted by the second argument automaton (see Figure 9).

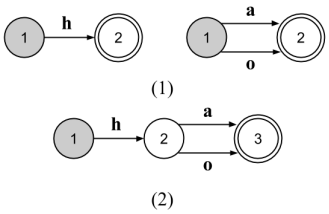


Figure 9: Argument automata (1) and their concatenation (2)

Replace in strings. Replace operation expects three automata as the arguments (see Figure 10). The first one contains strings where the replace must be performed (Figure 10(1)). The second one contains strings to be searched as substrings in the strings of the first automaton (Figure 10(2)). The third one encodes the set of strings to be used as a replacement(Figure 10(3)).

Conditional expressions (if, if-else). Suppose there is node `A` in the DDG. If the expression in the node `A` depends on some expression that can be constructed in different branches of the conditional operator there are several (two in case of if-else) paths leading to the node `A`. Same

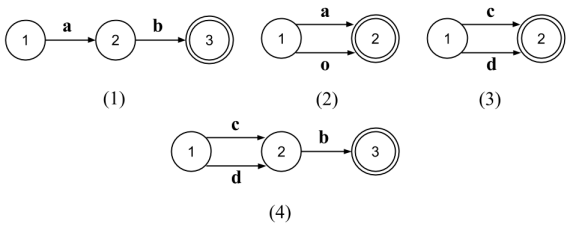


Figure 10: Replace operation arguments (1), (2) and (3) and the operation result (4)

type expressions are constructed along all these paths. To collect the information about all these possible expressions one must union all the automata obtained after traversing each path leading to the node `A`.

Loops. Loops processing is challenging. Firstly, during the static analysis it is often impossible to define how much iterations of the loop will be executed. So in general the loop produces an infinite language. Secondly, the language can be non regular and the example of such a case is shown on Listing 2. The fragment generates strings from the set $\{a^n b^n |\ n = 1,\ 2 \ldots\}$, that is a classic example of a nonregular language.

```
1  String str = "ab";
2  for(...) {
3    str = str.Replace ("ab", "aabb");
4   }
5  Db.Execute(str);
```

Listing 2: Loop with a replace operation in C#

To overcome the problems described the widening operator is used. The operator accepts two automata as the arguments and defines a special equivalence relation on the set of their states united. The equivalence classes formed by this equivalence relation are used as states for the resulting automaton, transitions are created based on the transitions of the argument automata. This resulting widened automaton accepts all the words accepted by the argument automata and generalizes these automata in some sense.

The operator is used for two automata that are the results of two successive loop iterations as follows:

$$A'_i \ = \ \nabla(A_i \cup A_{i-1}, A_{i-1})$$

where A_i, A_{i-1} – are the automata generated after current and previous iterations respectively, ∇ – widening operator. The A'_i automaton is used as a new result of the current iteration. What is important is that the sequence A'_i, $i = 1, 2 \ldots$ is always converge [13], i.e. beginning from some m:

$$\forall i > m \colon A'_i \ = \ A'_{i+1} \ = \ R$$

The notation $A'_i = A'_{i+1}$ means the automata are equivalent, i.e. they define the same regular language. The automaton R is set to be the result of the loop's approximation. In fact, here we solve the least fixed point computation problem.

Let us consider the example of widening operator usage for the loop in the Listing 3. The process of automata construction is presented in Figure 11. For each iteration the following automata are shown: the initial one – constructed as a result of loop body processing; the united automaton – constructed as a union operation result of the initial automaton and the automaton obtained in the previous iteration; the widened automaton – constructed as a result of applying widening operator to the united automaton and the automaton obtained in the previous iteration. If there is a dashed arrow from the automaton `A` to automaton `C` and from `B` to `C` the `C` is constructed from `A` and `B` using the operation which name is written in the same line the automaton `C` is placed (for example, union operation).

```
1  String str = "a";
2  for(...) {
3    str += "b";
4  }
5  Db.Execute(str);
```

Listing 3: Loop with concatenation on C#

It can be seen from the example the sequence of widened automata is converge on the third iteration. That is the widened automaton of the third iteration, accepting words from the set $\{ab^*\}$, is set to be the approximation result of the loop from the Listing 3.

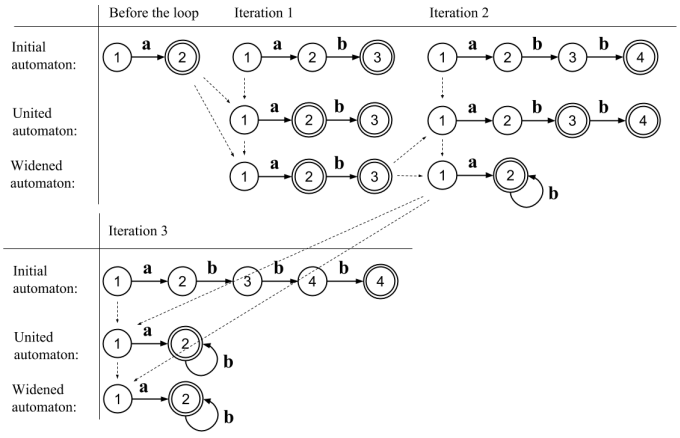


Figure 11: Approximation building for the loop from Listing 3

User input functions. User input functions like reading from a console or file can return any strings and it is impossible to predict these strings values during the static analysis. Therefore, it is assumed the automaton for the return values of such functions is an automaton accepting any words.

Formal definitions of concatenation, replace and widening operations for the FSA are given in [13].

Let us consider the example of FSA building for the DDG in Figure 7. The following steps will be performed.

1. Nodes 1, 2 and 3 contain literals, so for each of them the FSA accepting corresponding literal will be constructed.

2. After node 4 is processed the FSA constructed for node 2 will be bound to the name `"sql"`.

3. For node 7 the automaton will be constructed which is the result of concatenation of the automaton named `"sql"` and the automaton from node 1. The same is

for node 8 but the second argument of the concatenation is obtained from node 3. The resulting automaton for node 7 will accept the literal "SELECT * FROM Table1" and the automaton for node 8 – literal "SELECT * FROM Table2".

4. In nodes 9 and 10 the automata from nodes 7 and 8 respectively will be bound to name "sql".

5. The final node 11 is a node where alternative branches of conditional operator are merged. So the automata with the same names obtained from the alternative branches will be united here. That is, the resulting automaton for node 11 is the one that accepts both literals "SELECT * FROM Table1" and "SELECT * FROM Table2".

3.2.2 Implementation details of the approximation algorithm

One should note the implementation details of the described algorithm in our platform.

On the first step of the algorithm we create so called generalized CFG instead the usual one. The generalized CFG has the same structure as the usual one, but instructions in nodes are replaced with abstract ones. So in fact the generalized version represents the control flow of a pseudocode of the initial program. It is assumed that to build the approximation for some specific language one should convert language specific CFG to general one. This makes the platform able to support multiple host languages.

The approximation is built for some function (or method) where the strings with embedded code are constructed. During the processing of this function, the ones called from it are processed too. So the interprocedural approximation is built. To process the function calls the whole approximation algorithm is repeated recursively with the called function's return statement being a target expression. The recursion depth is constrained by some number that is defined as the algorithm's parameter and can be changed if needed. If the recursion depth becomes equal to the constraining parameter value the subsequent function calls are not processed anymore. The FSA accepting any words is used as the approximation result of this subsequent function calls.

The function called in source code under processing can be recursive itself. In this case the algorithm will always achieve constrained recursion depth and the approximation will be inaccurate because the FSA accepting any words will be used. In the case when called function is tail recursive we overcome described problem with special preprocessing. The generalized control flow subgraph corresponding to tail recursive function call is replaced with the subgraph corresponding to the loop equivalent to this tail recursive function. This loop is processed then with widening operator as any other loop. For non tail recursive functions there is no special treatment for now.

3.3 Lexical and syntax analysis

The purpose of the lexical analysis is to extract tokens defined by the language specification from the input stream of symbols. Also the correspondence between lexical units and the source code must be preserved. The result of lexical analysis of the dynamically generated expression is a finite-state automaton over the alphabet of tokens. In the classic lexical analysis the token can be represented by some identifier and the subsequence of input symbols. In the case of dynamically generated expressions analysis the input symbols subsequence is replaced by the FSA representing the subset of all possible values of dynamically generated expression. Moreover, in the latter case for each symbol we must preserve the information about its position in the source code.

Lexical analysis consists of four steps. Firstly, the input FSA that was build by the approximator is transformed into the deterministic one. After that the FST is constructed based on the FSA. The next step is to build the composition of the obtained FST and the FST that was built based on the language specification. This second FST produces tokens as an output. The composition is a result of the consecutive application of these two transducers, i.e. an output of first FST is used as an input for the second. After that we obtain either a set of lexical errors or a FST with a pair on each edge. The first item in the pair is the symbol with the binding to the source code, the second one is a function that returns either token or nothing. If the previous step end up with no errors, the last step is an interpretation of the FST resulting in a finite-state automaton over the tokens alphabet.

This automaton is processed by the syntax analyzer. The analyzer is based on RNGLR [11] algorithm which is a modification of Tomita's GLR-algorithm[12]. Tomita's GLR algorithm allows to handle arbitrary context-free grammars. GLR handles Shift/Reduce and Reduce/Reduce conflicts – situations when an available data are not enough to choose the correct way of parsing. Therefore, GLR processes all possible ways in case of conflict. So several derivation trees for a single input sequence can be obtained. The parser used in our platform extends this approach. The algorithm processes a token graph as an input. Herewith we meet a new situation when one node has several outgoing edges. We call this a "Shift/Shift" conflict despite the fact it is not an actual conflict because each way should be processed. More detail about the parser algorithm can be found in [4].

3.4 SPPF analysis and semantics calculation

The result of dynamically generated expressions parsing is a parse forest represented by the structure called SPPF [10]. This structure allows to reuse common subtrees for different trees.

Based on SPPF it is possible to build a so called compound control flow graph and try to solve classic static analysis tasks using it. When building compound CFG we assume that the string-embedded language grammar is a grammar of some programming language and that a mapping between non-terminals and CFG blocks is defined. For instance the "if_stmt" non-terminal derivation should be treated as the conditional block in CFG. Other example is shown in Figure 12, where the "assignment" non-terminal derivation is treated as the assignment operation in CFG.

Because of the fact that SPPF can contain several trees, the compound CFG can contain several usual CFGs. To make this possible an intermediate nodes are introduced. Further intermediate nodes will be called *"nodes"* (in figures they will have oval shape) and CFG elements will be called *"blocks"* (in the figures they will have rectangle shape). The derivation of "assignment" non-terminal corresponding to sequences "X = 1;" and "Y = 1;" is shown in Figure 13.

Using obtained compound CFG one can solve the static

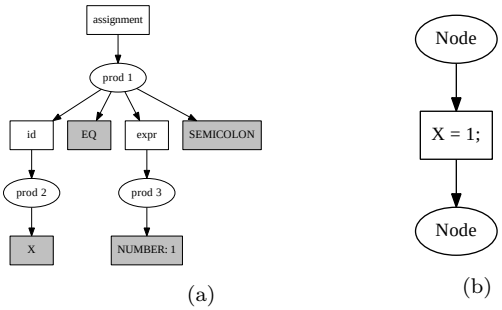


Figure 12: `assignment` non-terminal derivation (12a) and corresponding CFG block (12b)

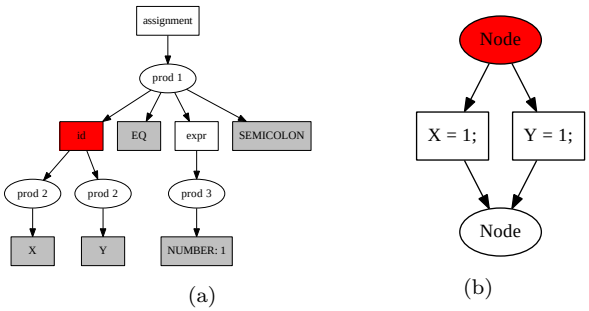


Figure 13: `"assignment"` non-terminal derivation (13a) and corresponding CFG blocks (13b)

analysis tasks. Consider definite assignment analysis as an example. Here we suppose that variable is defined in the statement α if and only if there is a variable definition along each of the evaluation paths from the program's start to the α statement.

For each block α the following transitions are valid:

$$before(\alpha) = \bigcap_{\beta \in pred(\alpha)} after(\beta)$$

$$after(\alpha) = before(\alpha) \cup gen(\alpha)$$

where

- $before(\alpha)$ set contains variables defined before the statement α;
- $after(\alpha)$ set contains variables defined after the statement α;
- $gen(\alpha)$ set is empty if α is not an assignment block and contains left-hand operand of the assignment otherwise.

As in case of usual CFG, the algorithm traverses the graph beginning from the start node for which the empty set is associated with. The only difference is that the $pred(\alpha)$ set contains nodes, not CFG blocks. But it is easy to define the required sets:

$$before(\beta) = \bigcap_{\alpha \in pred(\beta)} after(\alpha)$$

$$after(\beta) = before(\beta)$$

The compound CFG is shown in Figure 14. In the red colored node only variable `X` is defined. This node have two child blocks and the list containing variable `X` is passed to each block. The `"Y = X + 2;"` block adds the variable `Y` to the list (because `Y` is defined in this block), and the `"Z = X * 3;"` block adds the variable `Z`. These two blocks have the same child node, therefore the intersection of parents' lists is associated with it, i.e. the list containing the only variable `X`. The list is passed when the block `"X = Y * Z;"` is processed. Since passed list does not contain `Y` and `Z`, they will be reported as undefined variables.

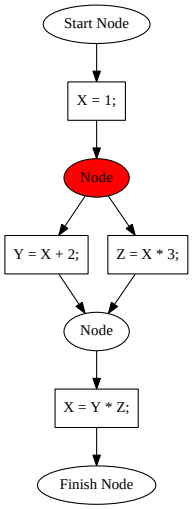


Figure 14: Compound control flow graph

4. USING THE PLATFORM

In order to use the platform a developer have to perform two steps.

- Add the host language support. To do this it is necessary to provide the module which for a source code on the host language can build a generalized CFG and mark target nodes in it. Generalized control flow graph specification is defined by the platform. The result of this module's work will be passed to the Approximation component as an input.
- Add the support of the string-embedded language through the implementation of the common interface. To achieve this it is necessary to provide the lexical specification (needed for lexer generation) and the grammar (needed for parser generation) of the embedded language. After the lexer and parser are created developer can use the standard implementation of the interface functions.

After these steps are performed a developer obtains the tool which supports the desired string-embedded language inside the desired host one. An embedded code syntax highlighting, matching parentheses highlighting, and static errors detection becomes available for these pair of languages. Microsoft Visual Studio with the ReSharper[7] plugin is used

[7]ReSharper is plugin to Microsoft Visual Studio, extend-

as a target IDE. The functionality currently available is not very rich, but the platform's architecture is designed for further extension.

Currently available functions can be configured by the user. Syntax highlighting can be configured by specifying the mapping between each token type and the color it should be highlighted with. Also user can choose which tokens are considered as paired: for each pair "left" and "right" elements are to be set (opening and closing parentheses, for instance). If a caret in the editor is located near one of the paired elements the other element corresponding to it will be highlighted.

4.1 Evaluation

To test the platform the tools for embedding T-SQL and arithmetic expressions language (called Calc) in C# and JavaScript were implemented. The tools are designed to work on Microsoft .NET platform. The most popular IDE for the .NET platform is Microsoft Visual Studio. By default it has no string-embedded languages support. The implemented tools were integrated as extensions to ReSharper – a plugin to Microsoft Visual Studio, which extends the the default functionality of the IDE.

ReSharper has a freely available SDK containing most of the ReSharper's functionality, in particular the capability to build control flow graphs for the source code. The modules for building C# and JavaScript specific CFGs and converting them to generalized ones were implemented using the SDK.

For strings with embedded code searching, an algorithm based on the special predefined set of functions (or methods) usage was implemented. These functions are called hotspots. What is special about these functions is that we know they expect a string with embedded code as an argument. In the source code we search calls of these hotspot functions and the arguments of the calls are set to be target expressions. The main idea of the approach is based on the fact, that the programs that construct the strings with embedded code often use some subsystem to execute it. The method `"execute"` of the `java.SQL.Statement` interface that is used in Java programs to execute SQL queries via JDBC technology can be considered as an example. In general the described approach does not always work correctly, but relative implementation simplicity is it's undoubted advantage.

The information about hotspots is stored in the configuration .xml file. For each hotspot it contains corresponding function's signature and the name of the embedded language the hotspot corresponds to. In the case of T-SQL a hotspot is a method or function named `ExecuteImmediate`, in case of Calc – named `Eval`. It should be noted that hotspots for different embedded languages can occur in the same analyzed file with the source code.

Lexical and parser specifications were built for Calc and a subset of T-SQL languages. The lexer and parser were generated using these specifications. Xml files for syntax highlighting and set of classes for ReSharper integration were created during the parser generation.

Syntax highlighting. The example of T-SQL and Calc languages support in C# is shown in Figure 15. Each language is highlighted separately: numbers in T-SQL query and in Calc expressions have different colors. Note the value of the `"expr"` variable is formed in a loop using conditional statement and *TrueCaseStr* function invocation. Plugin handles all these host language constructions for strings building so highlighting is performed even inside *TrueCaseStr* method.

```
 8  static int Calculate(bool cond)
 9  {
10    var query = "insert into y(u, v)";
11    query += "values(1,2)";
12    Program.ExecuteImmediate(query);
13
14    string expr = "(10";
15    for (int i = 0; i < 10; ++i)
16    {
17      expr += " + 1";
18      if (cond)
19        expr += TrueCaseStr();
20    }
21    return Program.Eval(expr + ") /2");
22  }
23
24  static string TrueCaseStr()
25  {
26    return "*3";
27  }
```

Figure 15: T-SQL and Calc syntax highlighting

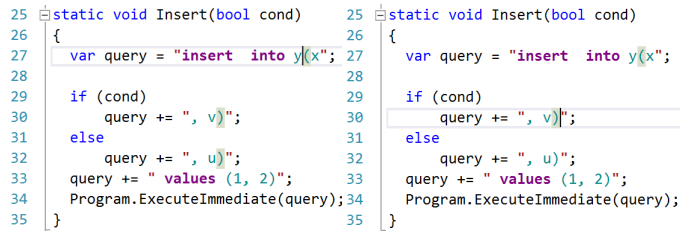
Static errors detection. Plugin can find lexical errors and errors related to language semantics. In figure 16 `"varY"` variable (line 22) is underlined as undefined because there exists the path that does not contain `"varY"` declaration (if `cond` parameter value is false).

```
17  public static void Execute(bool cond)
18  {
19      string query = "varX = 1;";
20      if (cond)
21          query += "varY = 2;";
22      query += "varZ = varX + varY;";
23      Program.ExtEval(query);
24  }
```

Figure 16: Static errors detection

Matching parenthesis highlighting. Matching parenthesis highlighting improves the code readability. If a caret in the text editor is located near the one of paired symbols then the plugin should highlight it and it's matching symbol. The situation with dynamically generated expressions is different because one symbol in a pair may have several matching ones. Such a case is shown in Figure 17a. Opening brace (line 27) has two matching closing braces (lines 30 and 32) so three elements are highlighted. At the same time each closing brace has only one matching opening brace (Figure 17b).

ing standard IDE functionality. Site: http://www.jetbrains.com/resharper/



(a) Multiple matches (b) Single match

Figure 17: Parenthesis highlighting

5. FUTURE WORK

The future work will include the platform and plugin improvement. At the platform level the available functionality for embedded code processing will be extended. Particularly we plan to implement the mechanisms that will make string-embedded code transformations possible. Code transformations can be used to perform migration from one particular DBMS to another [3] or to new embedding technology such as LINQ. This task is related to two problems: possibility of non-trivial transformations and proof of transformations correctness. Also we plan to implement type checking. For SQL it may be both a type checking inside the dynamically constructed queries and the consistency of the returned and expected types. In the later case we should check whether the type of the query result equals to the type of the variable in the host language the result is assigned to.

Performing semantic actions over SPPF is another important direction of the research. It is needed for refactoring, translation quality improvement, automation of the transition to a more reliable meta-programming tools.

6. REFERENCES

[1] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene. An interactive tool for analyzing embedded sql queries. In *APLAS'10 Proceedings of the 8th Asian conference on Programming languages and systems*, pages 131–138. Springer-Verlag Berlin, Heidelberg, November 2010.

[2] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS'03 Proceedings of the 10th international conference on Static analysis*, pages 1–18. Springer-Verlag Berlin, Heidelberg, June 2003.

[3] S. Grigorev. Automated transformation of dynamic sql queries in information system reengineering, 2012. Master's thesis, Saint-Petersburg State University.

[4] S. Grigorev and I. Kirilenko. Glr-based abstract parsing. In *CEE-SECR '13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*. ACM New York, NY, USA, October 2013.

[5] T. Hanneforth. Finite-state machines: Theory and applications unweighted finite-state automata, 2008. Institut fur Linguistik Universitat Potsdam.

[6] I. Kirilenko, S. Grigorev, and D. Avdiukhin. Syntax analyzers development in automated reengineering of informational system. *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, 174(3):94–98, June 2013.

[7] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[8] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05 Proceedings of the 14th international conference on World Wide Web*, pages 432–441. ACM New York, NY, USA, May 2005.

[9] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Varis: Ide support for embedded client code in php web applications. In *Proceedings of the 37th ACM/IEEE International Conference on Software Engineering (ACM/IEEE ICSE 2015)*. IEEE CS Press, May 2015.

[10] J. Rekers. Parser generation for interactive environments, 1992. Ph.D. thesis, University of Amsterdam.

[11] E. Scott and A. Johnstone. Right nulled glr parsers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):577–618, July 2006.

[12] M. Tomita. Lr parsers for natural languages. In *ACL '84 Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics*, pages 354–357. Association for Computational Linguistics Stroudsburg, PA, USA, July 1984.

[13] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, February 2014.