Research Article

Endre Fülöp and Norbert Pataki*

A DSL for Resource Checking Using Finite State Automaton-Driven Symbolic Execution

https://doi.org/10.1515/comp-2020-0120
Received Mar 31, 2020; accepted May 28, 2020

Abstract: Static analysis is an essential way to find code smells and bugs. It checks the source code without execution and no test cases are required, therefore its cost is lower than testing. Moreover, static analysis can help in software engineering comprehensively, since static analysis can be used for the validation of code conventions, for measuring software complexity and for executing code refactorings as well. Symbolic execution is a static analysis method where the variables (*e.g.* input data) are interpreted with symbolic values.
Clang Static Analyzer is a powerful symbolic execution engine based on the Clang compiler infrastructure that can be used with C, C++ and Objective-C. Validation of resources' usage (*e.g.* files, memory) requires finite state automata (FSA) for modeling the state of resource (*e.g.* locked or acquired resource). In this paper, we argue for an approach in which automata are in-use during symbolic execution. The generic automaton can be customized for different resources. We present our domain-specific language to define automata in terms of syntactic and semantic rules. We have developed a tool for this approach which parses the automaton and generates Clang Static Analyzer checker that can be used in the symbolic execution engine. We show an example automaton in our domain-specific language and the usage of generated checker.

Keywords: static analysis, Clang, finite state automata, domain-specific language

1 Introduction

Compilers play an essential role in the early detection of software problems regarding many aspects of the source code. Compilers validate the syntactic elements, referred variables, called functions to name a few. However, many problems may remain undiscovered.

Static analysis is a widely-used method which is by definition the act of uncovering properties and reasoning about software without observing its runtime behaviour, restricting the scope of tools to those which operate on the source representation, the code written in a single or multiple programming languages. While most static analysis methods are designed to detect anomalies (called bugs) in software code, the methods they employ are varied [1]. One major difference is the level of abstraction at which the code is represented [2]. Because static analysis is closely related to the compilation of the code, the formats used to represent the different abstractions are not unique to the analysis process, but can be found in the compilation pipeline as well [3].

Validation of resource management is an important process because resource problems (*e.g.* memory leak) may occur in C and C++ programs. Their validation is not part of the usual compilation process [4]. This kind of problems typically is not analysed by unit tests, therefore they are not found and the problems may exist for a long time. Special runtime environment can detect these problems, such as Valgrind [5].

Finite state automaton (FSA) is a handy tool for modeling usage of resources [6]. Taint analysis also takes advantage of finite state automata [7]. In this paper, we argue for a generic, resource-oriented static analysis approach. One can define a special resource-oriented checker with our domain-specific language (DSL) and our tool generates a static analysis method to detect the problems regarding the proposed resource.

This paper is an extended version of [8]. We have improved the DSL, especially an error handling sublanguage has been proposed. We have developed and presented a more precise and more detailed example how to use our DSL. Moreover, we have added an overview of related works.

The rest of this paper is organized as follows. First, we give details on the different levels of static analysis in Section 2. In Section 3, we present some typical resource-

Endre Fülöp: Department of Programming Language and Compilers, Eötvös Loránd University, E-mail: gamesh411@gmail.com
***Corresponding Author: Norbert Pataki:** ELTE Eötvös Loránd University, Budapest, Hungary, Faculty of Informatics, 3in Research Group, Martonvásár, Hungary, E-mail: patakino@elte.hu

oriented problems. We show how finite state automata help detecting resource problems in Section 4. The related work is discussed in Section 5. In Section 6 and 7, we define our domain-specific language regarding its syntactic and semantic elements. An integration of our approach with the Clang infrastructure can be seen in Section 8. We present a typical use-case in Section 9. Details on future work are given in Section 10. Finally, this paper concludes in Section 11.

2 Static analysis methods

Static analysis methods are different and their outcome may be based on the their capabilities. In this section, we present how the methods differ.

2.1 Textual representation

Some analysis techniques can be run on the source code text that can be seen in Listing 1. This representation is natural for the developer to read, and can be used for review purposes, the code style and the applicability of certain guidelines can be analyzed. The main limitation, however, is that for tools performing automated analysis on the stream of tokens, this representation can prove to be an insufficient source of information. The tokenized source code does not reveal the structure of the software which is normally added by the parser. Solutions formulated at this abstraction level are also sensitive to reformatting and refactoring transformations applied to the source code. As a result of these properties, it is infeasible to implement checks for complex properties in a stable, reusable fashion.

```
#define Z 0
#define DIV(a, b) ((a)/(b))

int main()
{
  int a = 1 / 0; // found trivially
  int b = 1 / Z; // needs preprocessor information
  int c = DIV(1, 0); // same as above

  // not detected by naive implementations
  int d = 1 / (1 - 1);
  int e = DIV(1, 1) / (DIV(1, 1) - DIV(1, 1));
}
```

Listing 1: The capabalities and shortcomings of textual representation

2.2 Abstract Syntax Tree

Compared to the textual representation, more elaborate analysis problems can be solved if the structure—dictated by the formal grammar of the programming language—is provided. In addition to revealing structure, type information also becomes accessible. The parser is the most common source of the abstract syntax tree (AST), as it is also used for code generation. This also means parsing of the source code is a prerequisite for static analysis using the AST [9]. Other drawbacks of this representation are that the dynamic properties of the software are still hidden, the possible execution paths cannot be taken into account when defining the detection criteria of an analysis solution. This level of abstraction can be utilized to detect errors similar to the problem in Listing 2. In the Clang infrastructure, there is an existing embedded domain-specific language for matching the AST, called the ASTMatchers library (Listing 3), which provides a declarative interface for selecting and traversing nodes of the AST and also allows one to bind them to symbols, which can later be used as input for other matchers, or for generating diagnostics [8].

```
// Clang Tidy check: cppcoreguidelines-slicing
struct B
{
  int a;
  virtual int f();
};

struct D : B
{
  int b;
  int f() override;
};

void use(B b) // Missing reference intended?
{
  b.f(); // Calls B::f.
}
//...
D d;
use(d); // Slice.
```

Listing 2: A check based on the syntactic information provided by the AST

```
// Clang ASTMatchers example
// This binds the CXXRecordDecl with name "::MyClass"
// to "myClassDecl".

recordDecl(hasName("::MyClass")).bind("myClassDecl")
```

Listing 3: Declarative matching of AST

2.3 Program flow

The AST alone is generally not sufficient to effectively detect problems like in Listing 4. If a static analysis method produces many false positives, it becomes harder to find real programming errors in the code, which in the long run reduces the effectiveness of the method. In order to reason the program flow, various data-flow analysis methods exist, which can also be found inside the compilation pipeline—often inside the code optimizer as well, but using their results for analysis purposes can be challenging.

```
enum Choice { YES, NO };

int f(Choice c)
{
  if (c == YES) { return 1; }
  else { return 2; }
}

int main()
{
  int a = f(getInputChoice());

  // safe to assume that a cannot be greater than 2
  // no matter what the initial choice was
  if (a > 2)
  {
    // unreachable
    return 1 / 0;
  }
}
```

Listing 4: False positives can occur due to insufficient information

2.4 Program path

Clang Static Analyzer uses an even more detailed model to aid the formulation of static analysis solutions. The program path analysis explores all possible execution paths of the program (Listing 5), and—in case of the Clang Static Analyzer—uses symbolic execution in order to reason about the values along a possible path [10]. The drawback is that the number of execution paths to explore is exponential in the number of branches. This means that practical program path analysis solutions involve some heuristics in order to select a subset of possible paths to be analyzed [11].

```
int main()
{
  // the return value of 'getInputChoice()'
  // is assumed to be one of the 2 enum values
  // or unknown
  int a = f(getInputChoice());
}
```

Listing 5: Values are modelled along execution paths

3 Resource-like problems

Resource-like problems involve a service of the running environment, which can be used by the program via calls to an API [12]. For instance, Linux systems have a standard way of interfacing file system. This API describes the correct usage of descriptors and function calls that can be seen in Fig. 1. Generally, these APIs also dictate some rules as for the usage of the individual parts. These can include restrictions on the multiplicity of service objects, the immediate order of triggering events concerning these objects, or eventual state of the service infrastructure. If these rules are not followed by the developer, usually some negative consequences occur, like resource-overuse and race conditions [13]. The summed complexity of programming language constructs, the problem domain logic, and the resource handling may cause errors which are not easily detectable or reproducible [14]. Static analysis can be used to detect violations of such contracts [15].

State machine representation of resource-like problems has another benefit. The level of abstraction can be raised, and many users of static analysis tools can better formulate the error conditions by specifying the abstract machine and its working logic, than by starting with the highly detailed view. This applies to solutions implemented in functional, as well as in procedural frameworks, as the level of technical detail to implement a checker is high.

In order to alleviate the implementation burden, several tools and libraries have already been implemented by the Clang Static Analyzer community. Most notably, there is the ASTMatchers library which generalizes the selection of interesting nodes from the translation unit, by giving a declarative embedded domain specific language. Another tool is under development, which would allow the implementer to specify interesting patterns inside the Control Flow Graph (CFG) of the examined software. Even though this would allow for matching against more dynamic aspects of the program execution, however, the authors consider this solution more in-line with the aforementioned ASTMatchers library, providing an extra level of precision without the burden of verbosity. It is important to note that both tools mentioned come with a cost. ASTMatchers uses dynamic memory allocation with reference counting, thus

if misused, the memory handling cost could prove to be significant compared to the execution of matching relevant nodes.

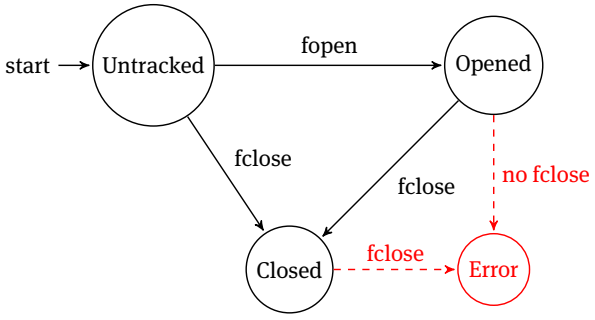

Figure 1: File handling API model

4 Generalizing error detection with FSA

As discussed above, the finite state automata model is used in the solution proposed by the authors to give a solution for describing error conditions in a program. This solution is in the form of an external DSL. The language lets its user specify a labelled state machine whose transitions are governed by the relevant events occuring during the symbolic execution paths. The exploration of said paths, and the generation of base events are assumed to be done by the framework. The authors implemented a concrete solution using Clang Static Analyzer as the backend which provides these facilities. This separation of concerns would allow this DSL to be used with other static analysis frameworks, and even other languages (besides C and C++, relevant to the example).

The main idea behind the error detection solution is to separate the matching or filtering into three distinct levels. These three are: *syntax-level*, *value-level*, and *model-level* matching of the relevant events. Also the priority of handling these levels is important. The traditional way of implementing checkers in case of Clang Static Analyzer is to implement callback functions which handle events that fire when symbolic execution reaches certain syntactic categories. This would make the traditional approach *syntax-first*. This means it is more fit for describing problems that are syntax-oriented, and would have to be more verbose in order to implement state-related checks. The proposed solution is *model-first* in contrast.

5 Related work

There are various code-analysis related works that employ the finite state machines as the fundamental abstraction. The work by Zhang et al uses dynamic symbolic execution to guide the static analysis to explore program paths which have a specific class of properties [16]. These properties are called regular, and the most crucial aspect of them is that the decision problem of whether a program path satisfies given property can be formulated with an FSM. The authors suggest the possibility of developing a guiding algorithm for a broader class of properties, which they call context-free properties. The main goal of the approach is the enhancement of the static analysis process, favouring paths that can lead to the detection events that are implemented independently of the guiding logic itself. From the viewpoint of finding bugs in a code, this approach is more general as the properties defined need not be erroneous executions, but also in itself insufficient as the coding error must also be detected and this is not done by the guiding algorithm. In contrast to the technique introduced in that paper, the current work proposes a DSL for detecting the error-condition and does not modify the exploration strategy of the analyzer engine in any way. Clang SA provides a method for client code both to check and to build the representation on which the analysis is performed. However, the DSL currently does not support this. Clang SA can also employ various exploration strategies, but providing the user of the language with the option of configuring these parameters is deemed out of scope for the complexity aims of the current DLS.

The work by Starynkevitch provides a generic DSL called Melt, which provides access to the internal data structures of the GNU C Compiler (GCC). The language is compiled, Lisp-like and high-level [17]. It provides an accessible interface to the GCC compiler infrastructure. The motivation behind this DSL is that the internal API of the GCC compiler infrastructure is not stable, and directly consuming that representation in client code comes with a considerable maintenance burden. Extending GCC with plugins implemented in MELT can be used for code-generation, optimization as well as code refactoring, coding standard validation and detecting security threats. The language is broad in scope, provides functional style program constructs that are later transpiled to C code, and packaged to use the standard GCC plugin API. The Checkerlang DSL aims at a narrower scope. Since its main goal is to describe erroneous program constructs as concisely as possible without leaking the internal representation of the Clang infrastructure, there are no facilities that can be

used in any other stage of the compilation pipeline. This simplicity provides accessibility to a wider audience and also more flexibility when implementing the runtime infrastructure of the DSL.

A hybrid implementation of FSM-based error detection is given by Slabý et al [15]. The authors use a multiphase analysis technique which employs flow-sensitive analysis and points-to analysis in the first phase to produce an instrumented version of the source code, which has the same semantics concerning the original problem it was specified for but initializes and triggers transition in a state machine that describes the problem to be detected. Then a slicing technique is used to prune execution paths which are irrelevant with respect to the errors to be detected. The instrumented and sliced source is then analyzed with symbolic execution. The result of symbolic execution analysis is a set of program paths along which the state machine reached an error-state. The solution shown by the authors is custom implementation, not necessarily building on a compiler infrastructure to provide the necessary intermediate representation; however, the details are not discussed. Checkerlang DSL does not itself provide a detailed implementation plan for checking the properties specified, only the declarative means for specifying the error-condition belonging to a specific problem domain, and the implementation is currently tied to the Clang infrastructure, more specifically Clang SA.

An important aspect of static analysis is the feedback to the programmers who typically use integrated development environments (IDEs) [18]. The result of static analysis should be displayed in this tool. However, Microsoft proposed a language-independent protocol called Language Server Protocol (LSP) that many programming environments support (*e.g.* Visual Studio Code) [19]. LSP is an emerging open-source protocol, but compilation diagnostics are supported comprehensively, so our idea is LSP-based approach for emit warnings into the IDEs since. LSP's decoupling approach is essential to reach a significant amount of developers.

6 Checkerlang DSL

The implementer could define the state machine (modelling Fig. 1 example) by explicitly specifying the transitions as seen in Listing 6. This example excludes the error states and transitions. The example defines a checker named `stream_checker`, which has a state machine (this is implicit, currently a single state machine kind can be defined for a checker, out of which multiple instances can be instantiated). The machine is defined by a list of transitions. Transitions have a head, a body and a tail part. The head part signifies the starting state, the tail part the destination state. The body part defines the guard conditions of the transaction, as well as the extraction of symbolic values, and diagnostics to be emitted. The extraction of symbolic values has a form of traditional assignment expressions, but with a predefined set of extractor functions allowed on the right hand side. The definition of states is implicit and the set of defined states is said to be the union of states appearing in the head and body parts of the transactions. The machine described in the example has three transitions, the first two of which are describing how to construct a machine. The start state of the conceptual machine is not modelled, instead transitions with the keyword `construct` in their head signify how to start tracking a symbolic value with a machine. Every constructor transition must have an extractor expression specifying the ID label. This value is then used later to track the state, and perform the necessary transitions. The body part consists of an arbitrarily ordered list of either extractor expressions, predicate expressions or diagnostic expressions. Extractor expressions define the labels of the state machine. These labels are persistent across transitions, so they do not follow lexical scoping. This also means, that label ID is readily available in every non-constructor transition. Predicate expressions govern the execution of the transition and the success of a transaction is dependent on their runtime evaluation. Diagnostic expressions help formulate the output of the analysis.

```
checker stream_checker

{constructor}
  ID = RESULT_OF(fopen)
{file_opened}

{constructor}
  ID = NTH_PARAM_OF(fclose, 1)
{file_closed}

{file_opened}
  fd = NTH_PARAM_OF(fclose, 1)
  fd == ID
{file_closed}
```

Listing 6: DSL to describe FSA

```
(* main language *)
checker_name = identifier;
checker_specification = "checker", checker_name;
head_id = identifier;
transition_head = "{" , head_id, "}";
transition_body = (sym_definition | sym_predicate |
     diag_expr)*;
```

```
tail_id = identifier;
transition_tail = "{", tail_id, "}";
transition_specification = transition_head,
    transition_body, transition_tail;
transitions = transition_specification*;
grammar = checker_specification, transitions;
```

Listing 7: EBNF of Checkerlang main language

```
(* sym_definition sublanguage *)
result_of_function_name = identifier;
result_of_expr = "RESULT_OF(", result_of_function_name,
    ")";
nth_param_of_function_name = identifier;
nth_param_of_argument_position = digit;
nth_param_of_expr = "NTH_PARAM_OF(",
    nth_param_of_function_name, ",",
    nth_param_of_argument_position, ")";
sym_expr = result_of_expr | nth_param_of_expr;
sym_definition_lhs = identifier;
sym_definition_rhs = sym_expr;
sym_definition = sym_definition_lhs, "=",
    sym_definition_rhs;
```

Listing 8: EBNF of Checkerlang extractor sublanguage

```
(* sym_predicate sublanguage *)
sym_predicate_lhs = identifier;
sym_predicate_rhs = identifier;
op_kind_eq = "==";
op_kind_neq = "!=";
sym_predicate_op = op_kind_eq | op_kind_neq;
sym_predicate = sym_predicate_lhs, sym_predicate_op,
    sym_predicate_rhs;
}
```

Listing 9: EBNF of Checkerlang predicate sublanguage

```
(* diagnostic sublanguage *)
leak_diag = diag, string;
diag = diag, string;
diag_expr = diag | leak_diag ;
}
```

Listing 10: EBNF of Checkerlang diagnostic sublanguage

7 Semantics of Checkerlang DSL

The transitions defined by Checkerlang are to be executed on a best-efforts basis. This means that the transition described by the language (Listing 7) should be executed like this: try to execute all statements of the transition body in their original order. If a state extractor statement (Listing 8) is encountered, evaluate the expression on the right hand side, then update value of the label indicated by the left hand side. In case of a symbolic predicate statement, evaluate the left and the right hand side in no particular order, then evaluate the whole expression. If the value of the whole symbolic predicate statement (Listing 9) is true, then proceed to the next statement. If the value is false, cancel the transaction. This signifies an aborted transaction, however, any labels updated so far are not rolled back to their original state. Besides a symbolic predicate expression evaluating to false, any errors during any evaluation also causes abortion. If the statement list is processed without abortion, the transition is deemed successful, and the machine transitions into the destination state. When a diagnostic expression is met (Listing 10) depeding on the type of the diagnostic expression one of the following would happen: immediate diagnostic expression triggers the emission of a bug report immediately. There is no way to cancel an immediate diagnostic. In case of leak diagnostics, the diagnostic message is saved for later use. If the machine goes out scope during symbolic execution (this means that the symbol associated with its ID label is considered dead by the execution framework), this diagnostic is emitted. The emission of such diagnostic is however cancelled if any later transaction is successfully executed. These rules help formulate the checking logic in high abstraction level.

8 Practical usage within the Clang infrastructure

We have implemented a parser to the Checkerlang DSL. This parser uses the PEG parser combinator header-only library PEGTL [8]. The library is written in the C++ programming language, and allows the assembly of parser from subparsers. The composition mechanic is implemented by inheriting from class templates, and makes heavy use of template metaprogramming [20]. The generated parser is a top-down recursive descent parser, which does not have a separate scanner. The lexing phase is completely omitted, and built in the parser using whitespace-accepting combinators. As the code Example 11 shows, implementing the DSL via parser combinators produces a succint and readable parser implementation.

```
struct ws : one< ' ', '\t', '\n', '\r' > {};
struct wss : star< ws > {};
struct wsp : plus< ws > {};

struct checker_name : identifier {};
```

```
struct checker_specification : seq< S("checker"), wsp,
    checker_name > {};
struct head_id : identifier {};
struct transition_head : seq< one< '{' >, wss, head_id,
    wss, one< '}' >> {};
struct sym_definition : sym_extractor_lang::rules::
    sym_definition {};
struct sym_predicate_expr : sym_predicate_lang::rules::
    sym_predicate_expr {};
struct transition_body : list< sor<sym_definition,
    sym_predicate_expr, diag_expr>, wsp > {};
struct tail_id : identifier {};
struct transition_tail : seq< one< '{' >, wss, tail_id,
    wss, one< '}' >> {};
struct transition_specification : seq< transition_head,
    wss, transition_body, wss, transition_tail > {};
struct transitions : list< transition_specification, wsp
     > {};
struct grammar : seq< wss, checker_specification, opt<
    wsp, transitions >, wss > {};
```

Listing 11: Implementating with parser combinators

We have created a project where this header-only implementation of the parser is used to create a Clang plugin. The plugin itself is a dynamically-linkable file that implements a Clang Static Analyzer checker. This checker can be parametrized with a filename, which contains the specification of the static analysis problem written in Checkerlang, parses it during execution, and directs the analysis and the emission of diagnostics during runtime. This solution enables the rapid development of static analysis solutions as no recompilation is needed to test independent checks, only parametrizing the Clang invocation differently can result in injecting a brand new checker logic.

9 Implementation example

We have implemented a checker using the Checkerlang DSL, which makes use of the diagnostic capabilities mentioned in order to find bugs in the C standard library file handling API.

The code snippet on Listing 12 is not perfect regarding the resource usage, the opened file is not closed. We present the application of the generated checker on this source code.

```
#include <stdio.h>
#include <string.h>

int f(const char* path, const char* content)
{
    FILE* outfile = fopen(path, "w");
    fwrite(content,
           sizeof(char),
           strlen(content),
           outfile);
    return 0;
}

int main(int argc, char** argv)
{
    return f(argv[1], argv[2]);
}
```

Listing 12: Code with file handling error

```
checker stream_checker

{constructor}
  ID = RESULT_OF(fopen)
  ID != 0
{file_opened}

{constructor}
  ID = NTH_PARAM_OF(fclose, 1)
{file_closed}

{file_opened}
  fd = NTH_PARAM_OF(fclose, 1)
  fd == ID
  LEAK_DIAG("File left open!")
{file_closed}

{file_closed}
  fd = NTH_PARAM_OF(fclose, 1)
  fd == ID
  DIAG("File closed multiple times!")
{file_closed}
```

Listing 13: Extended example in Checkerlang DSL

An extended example can be seen on Listing 13. The diagnostics are extended to handle both the error cases of active misuse and negligence in upholding the API contracts. There is an alternative flow of checking the usage even before actively checking for an fclose event, or handling the leak condition. In the first constructor of the implementation, the `ID` parameter must also be checked for not being null. This is because in the opening of the filed, the symbolic value of the `fopen` expression does not represent a real file. In case the opening failed, the return value is a null pointer. Checking this return value should be done in the user code, and is not deemed to be the responsibility of this checker. There exist many other checkers within the core infrastructure of most static analysis engines which handle the generic null pointer usage error condition.

The second constructor allows the analysis to start the modeling from an intermediate state. If the analysis engine happens to not know anything about the symbolic value of

the first parameter of an `fclose` call, we assume that the file handle was used correctly beforehand. This assumption extends the possible range of erroneous usages found, but remains on the conservative side of modeling, as the unknown past usages of the file handler are assumed to be correct.

```
file.c:11:5: File left open! [stream_checker]
    return 0;
    ^
```

Listing 14: Diagnostic of the generated checker (inside function f)

The generated checker finds the incorrect resource management regarding the Listing 12. The checker emits the diagnostics that can be seen on Listing 14.

10 Future work

The extractor language has some limitations at the moment. In order to harness the power of the tools already developed, namely the ASTMatchers library, the extractor language should be extended. A trivial extension would be to create an expression extractor sublanguage as well and integrate it into the current symbolic extractor language. The expression extractor sublanguage could be the ASTMatchers language itself. Another extension would be to allow multiple kinds of machines to be defined in a checker. The instances of multiple machines could share state, or pass messages between each other to coordinate for better solutions. The semantics of these extensions are to be defined later on, as well as the performance cost of using dynamic linking and interpreting the checking logic.

11 Conclusion

Resource problems in C/C++ programs are quite typical. Compilers cannot detect many problems regarding resources, like double free, use-after-free, leakage, etc. In this paper, we presented our generic approach that aims at static analyses of improved resource management. The proposed approach involves a new domain-specific language, called Checkerlang in which FSA can be specified. This paper described the definition of this language. We have developed a tool that is able to parse Checkerlang source code and it generates Clang Static Analyzer checker from the specified automaton. The checker is utilized during symbolic execution. We presented how this approach can be applied with a use-case.

Acknowledgement: The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications)

References

[1] Johnson B., Song Y., Murphy-Hill E., Bowdidge R., Why don't software developers use static analysis tools to find bugs?, In: D. Notkin, B. H. C. Cheng, K. Pohl (Ed.), Proceedings of the 2013 International Conference on Software Engineering (18–26 May 2013, San Francisco, California, USA), IEEE Computer Society, 2013, 672–681

[2] King, C., Symbolic execution and program testing, Commun. ACM, 1976, 19, 385–394

[3] Nagappan N., Ball T., Static analysis tools as early indicators of pre-release defect density, In: G. Roman, W. G. Griswold, Ba. Nuseibeh (Ed.), Proceedings of the 27th International Conference on Software Engineering (15–21 May 2005, St. Louis, Missouri, USA), ACM, 2005, 580–586

[4] Meyers S., Effective C++, 3rd ed., Addison-Wesley, 2005

[5] Nethercote N., Seward J., Valgrind: A framework for heavyweight dynamic binary instrumentation, In: J. Ferrante, K. S. McKinley (Ed.), Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (10–13 June 2007, San Diego, California, USA), ACM, 2007, 89–100

[6] Novitzká V., Mihályi D., Slodicak V., Finite automata in the mathematical theory of programming, In: E. Kovács, P. Olajos, T. Tómács (Ed.), Proceedings of the 7th International Conference on Applied Informatics vol. 2 (28–31 January 2007, Eger, Hungary), 2007, 91–98

[7] Arroyo M., Chiotta F., Bavera F., An user configurable clang static analyzer taint checker, In: C. Cubillos, H. Astudillo (Ed.), Proceedings of the 2016 35th International Conference of the Chilean Computer Science Society (10–14 October 2016, Valparaiso, Chile), IEEE, 2016, 1–12

[8] Fülöp E., Pataki N., Symbolic Execution with Finite State Automata, In: W. Steingartner, Ŝ. Korečky, A. Szakál (Ed.), Proceedings of the 2019 IEEE 15th International Scientific Conference on Informatics (20–22 November 2019, Poprad, Slovakia), IEEE, 116–120

[9] Babati B., Horváth G., Májer V., Pataki N., Static analysis toolset with Clang, In: G. Kusper, G. Kovásznai, R. Kunkli, S. Király, T. Tómács (Ed.), Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary), 2017, 23–29

[10] Horváth G., Pataki N., Source language representation of function summaries in static analysis, In: Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (17–22 July 2016, Rome, Italy), ACM, 2016, 6(1)–6(9)

[11] Szabó Cs., Kotul M., Petruš R., A closer look at software refactoring using symbolic execution, In: E. Kovács, G. Kusper, R. Kunkli, T. Tómács (Ed.), Proceedings of the 9th International Conference on Applied Informatics vol. 2 (29 January–1 February 2014, Eger, Hungary), 2014, 309–316

[12] Dewhurst S. C., C++ gotchas avoiding common problems in coding and design, Pearson Education, 2003
[13] Stroustrup B., The C++ programming language, 4th ed., Addison-Wesley, 2013
[14] Papp D., Pataki N., Bypassing memory leak in modern C++ realm, Annales Mathematicae et Informaticae, 2018, 48, 43–50
[15] Slabý J., Strejček J., Trtík M., Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution, In: M. Stoelinga, R. Pinger (Ed.), Formal Methods for Industrial Critical Systems (27–18 August 2012, Paris, France), 2012, 207–221
[16] Zhang Y., Chen Z., Wang J., Dong W., Liu Z., Regular property guided dynamic symbolic execution, In: A. Bertolino, G. Canfora, S. G. Elbaum (Ed.), Proceedings of the 37th IEEE International Conference on Software Engineering (Vol. 1) (16–24 May 2015, Florence, Italy), 2015, 643–653
[17] Starynkevitch B., MELT – a Translated Domain Specific Language Embedded in the GCC Compiler, In: O. Danvy, C. Shan (Ed.), Proceedings DSL 2011: IFIP Working Conference on Domain-Specific Languages (6–8 September 2011, Bordeaux, France) Electronic Proceedings in Theoretical Computer Science, 2011, 66, 118–142
[18] Sulír M., Bačíková M., Chodarev S., Porubän J., Visual augmentation of source code editors: A systematic mapping study, Journal of Visual Languages & Computing, 2018, 49, 46–59
[19] Mészáros M., Cserép M., Fekete A., Delivering comprehension features into source code editors through LSP, In: K. Skala (Ed.), Proceedings of the 42nd internal convention MIPRO 2019 (20–24 May 2019, Opatija, Croatia), 1581–1586
[20] Porkoláb Z., Sinkovics Á., Domain-specific language integration with compile-time parser generator library, In: E. Visser, J. Järvi (Ed.), Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (10–13 October 2010, Eindhoven, The Netherlands), ACM SIGPLAN Notices, 2010, 46(2), 137–146