Towards Static Modular Software Verification

Marcus Mews, Steffen Helke

Department of Software Engineering
Technische Universität Berlin
Ernst-Reuter-Platz 7
10587 Berlin
{mews, helke}@cs.tu-berlin.de

Abstract: The paper presents our first work in progress results of an approach to verify the correct use of software libraries in target projects. Therefor the project's source code is analyzed and checked against the library's behavior specification, called interface grammar. This grammar is formalized using annotated state diagrams, and the verification analysis is based on static control flow, data flow and alias analyses. The paper illustrates the presented approach using a small-sized Java library example. In the end, we give a brief outlook to necessary enhancements.

1 Introduction

When developing software, in many cases software engineers include and reuse software libraries. But reusing third party's libraries necessitates a thorough understanding of the software library. Without proper care, misused libraries can lead to errors and exceptions at runtime, and can thus endanger the safety of the developed software. Hence the question arises, whether the included software libraries are utilized correctly and how to get prove.

In our context, utilizing a software library means nothing else but calling a library's interface methods. Usually, most software libraries provide a documentation including e. g. its methods, which are intended to be called in a specific order. This grammar is part of the interface specification and its violation can cause the library and/or its caller to fail.

We address the issue of wrong calling orders of library methods and present a static source code analysis for modular software verification. Inputs to this analysis are the interface grammar and the complete source code which utilizes the library. In this paper, we use state machines to specify the interface grammar. As a result of the analysis, two succeeding library calls are detected which may lead to a violation of the library's specification at runtime. Since the analysis relies on naturally imprecise control flow, data flow and alias analyses, its results can contain false positives. Nevertheless, the presented analysis can give sound evidence that a library is utilized correctly, if no errors are detected.

2 Static Software Utilization Verification

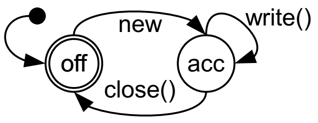
Our static modular software verification is presented in two steps: First, we show how to derive possible misuses from the interface grammar. Then, we explain how we verify whether the software source code contains any of these misuses. But before, have a look at Java Listing 1: Our approach finds the two FileOutputStream misuses: accessing the same file twice at the same time and omitting to close the second file stream.

```
public class FileOutputStream_Error {
    public static void main(String[] args) throws IOException {
        File file = new File("c:/line.txt");
        FileOutputStream fos1 = new FileOutputStream(file);
        FileOutputStream fos2 = new FileOutputStream(file);
        fos2.write("Hallo Welt".getBytes());
        fos1.close();
} }
```

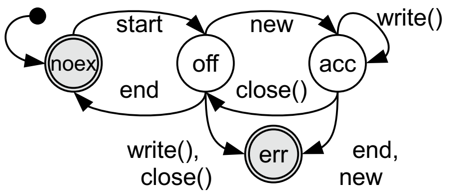
Listing 1: This compiling code contains two library misuses (one throwing a runtime exception)

2.1 Step 1: Find Error Paths

Misusing a software library means that the library's interface methods are called in a wrong order, or the library is not shut down appropriately before the program terminates. We call a sequence of succeeding interface events (method calls or program start/termination) leading to an error state an error path. In this section, we outline how to derive an error path from the interface grammar.



(a) Original interface specification L_{FOS}



(b) Extended specification L_{FOS_X}

Figure 1: Reduced interface grammar of the Java FileOutputStream library

We use state machines to specify the interface grammar: $L = (Q_L, \Sigma_L, \Delta_L, q_{0_L}, F_L)$ (Fig. 1). Q_L contains all states, Σ_L contains all input symbols and $\Delta_L \subseteq Q_L \times \Sigma_L \times Q_L$ contains all transitions. Δ_L maps from a start state and an input symbol ($dom(\Delta_L) \subseteq Q_L \times \Sigma_L$) to a target state in Q_L. Σ_L is the set of qualified interface constructor and method names (abbreviated in Fig. 1). q_{0_L} is the initial state and F_L contains all finite states. We simplify the task and derive error paths with length of two, only. Therefor, we restrict L so that all transitions with the same input symbols lead to the same state:

$$\frac{(q_1, \sigma_1, {q'}_1), (q_2, \sigma_2, {q'}_2) \in \Delta_L}{(\sigma_1 = \sigma_2) \Rightarrow (q'_1 = q'_2)}$$

Next, we derive a second state machine $L_X = (Q_X, \Sigma_X, \Delta_X, q_{0_X}, F_X)$ based on L (Fig. 1(b)). The purpose of L_X is to enrich L with information about the program start and terminate events, and an error state and its attached transitions. Therefor, in exchange for start and finite state markings we add a no execution state q_{noex}, the input symbols σ_{start}, σ_{end}, and transitions Δ_{noex} to and from q_{noex}. We also add an error state q_{err} and transitions Δ_{err} from every state to q_{err}: If a state lacks an outgoing transition that fires on an input symbol σ_i, a new transition to q_{err} is added. The state machine L_X remains deterministic and Δ_X still has only one target state for every tuple in its domain. The initial state now is $q_{0_X} = q_{noex}$, and the and the finite states are $F_X = \{q_{noex}, q_{err}\}$.

$$Q_X \triangleq Q_L \cup \{q_{err}, q_{noex}\},\ \Sigma_X \triangleq \Sigma_L \cup \{\sigma_{start}, \sigma_{end}\},\ \Delta_X \triangleq \Delta_L \cup \Delta_{noex} \cup \Delta_{err}$$

$$\Delta_{noex} \triangleq \{(q_{noex}, \sigma_{start}, q_{0_L})\} \cup \{(q_f, \sigma_{end}, q_{noex}) \mid q_f \in F_L\}$$

$$\Delta_{err} \triangleq \{(q_i, \sigma_i, q_{err}) \mid q_i \in Q_L \wedge \sigma_i \in (\Sigma_L \cup \{\sigma_{end}\}) \wedge (q_i, \sigma_i) \notin\ dom(\Delta_L \cup \Delta_{noex})\}$$

At last, we calculate error paths using L_X. As a benefit of the state machine restriction mentioned above, we can reduce complexity and length of the error paths. An error path $p \in P$ is a list of succeeding interface events, and in our case defined as $P \subseteq \Sigma_X \times \Sigma_X$, containing only two events in a row. P_{FOS} shows all error paths of the Java File Stream library of L_{FOS_X}, and $P_{Listing}$ shows the two error paths that can be found in Listing 1.

$$P \triangleq \{(\sigma_i, \sigma_j) \mid \delta_m, \delta_n \in \Delta_X \wedge \delta_m = (q_i, \sigma_j, q_{err}) \wedge \delta_n = (q_k, \sigma_i, q_i)\}$$

$$\begin{aligned} P_{FOS} \triangleq\ & \{(\sigma_{start}, \sigma_{write()}), (\sigma_{start}, \sigma_{close()}), (\sigma_{close()}, \sigma_{write()}), (\sigma_{close()}, \sigma_{close()})\} \\ & \cup \{(\sigma_{new()}, \sigma_{new()}), (\sigma_{new()}, \sigma_{end}), (\sigma_{write()}, \sigma_{new()}), (\sigma_{write()}, \sigma_{end})\} \end{aligned}$$

$$P_{Listing} \triangleq \{(\sigma_{new()}, \sigma_{new()}), (\sigma_{write()}, \sigma_{end})\}$$

2.2 Step 2: Check Project

With the error paths at hand, we analyse the program and detect possible library misuses. The library interface methods can be either static or bound to receiver objects. Since we support multiple library instances, library misuses have to be checked for every library instance and its aliases. Thus, aliasing and control flow problems are tackled now.

2.2.1 Alias Analysis

The flow insensitive may alias analysis respects the following assignments: ordinary variable assignments, parameter assignments of method calls, assignments from return statements to method declarations, and from method declaration to all possibly bound method calls. The analysis uses symbols $s \in S$ for variables and methods calls/declarations. We refer to every kind of assignment from symbol s_1 to s_2 with the fact notation $assigned_d(s_1, s_2)$. We then specify transitive assignments with $assigned(s_1, s_2)$, and define that two symbols s_x and s_y do alias when they both have an assignment symbol o in common.

$$assigned(s_1, s_2) \triangleq \exists s_i \in S \mid assigned_d(s_1, s_2) \vee (assigned_d(s_1, s_i) \wedge assigned(s_i, s_2))$$

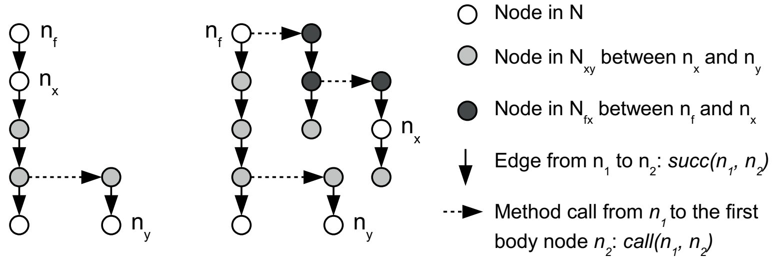
$$alias(s_x, s_y) \triangleq \exists o \in S \mid assigned(s_x, o) \wedge assigned(s_y, o)$$

2.2.2 Control Flow Analyses

The goal of the control flow analysis is to find two directly succeeding interface events n_x and n_y in the source code. This means that other library events n_B may not be fired in between those two events. More precisely: There exists at least one control flow path from n_x to n_y so that no other n_B is in between. In this subsection, we first describe how we abstract from the source code, and then give a specification of our control flow analysis.

We transform the source code to a data structure $G = (M, B, N, E, C, n_0, F_M, F_P)$ with M as methods, N as nodes, $n_0 \in N$ as the program start node, and $F_P \subset N$ as the program terminal nodes. $E \subseteq N \times N$ is a relation that represents edges from one node to other nodes, and $C \subseteq N \times M$ is a relation that maps method calls from nodes to methods and respects polymorphism by mapping each node to all possible called methods. $B \subseteq M \times N$ is a function that maps every method to its first node, and $F_M \subseteq M \times N$ is a relation that maps every method to all its exit nodes. Additionally, $M_{LE} \subset M$ references all methods that invoke library events like methods of the analyzed library or methods that exit the program. In other words, G contains ordinary control flow graphs for every method of the program, and all Java statements/expressions are abstracted to nodes. Further, the following rules apply: (1) We begin at the first node of every method; (2) every node points to its predecessor(s) (except the last node in a method); (3) every method call node relates additionally to all possibly bound methods (C); (4) every `switch` condition statement node points to all of its conditional bodies and the next mandatory node if no default body was declared; and (5) every `if` condition statement points either to its two conditional bodies, or to its single conditional body and to the next mandatory node. To free G from loops, (6) there are no edges that point to previous nodes. Further, (7) the bodies of loop statements are copied once so that the loop statement node points to both, the original loop body b and a copy bb which is a concatenated version of two times b. Unrolling loop bodies to bb suffices since the error paths only have a length of two. Additionally, (8) conditional loops point to the next mandatory node, since they are not necessarily executed.

Additionally, the methods $start$ and end (representing the symbols σ_{start} and σ_{end} of L) are added to M. As a predecessor we insert a new first node that calls the method $start \in M$. And complementary, we add after every node that can be the last node of a regular program execution, a new succeeding last node that calls the method $end \in M$.


n_f
n_x
n_y
n_f
n_x
n_y
Node in N
Node in N_{xy} between n_x and n_y
Node in N_{fx} between n_f and n_x
Edge from n_1 to n_2: $succ(n_1, n_2)$
Method call from n_1 to the first body node n_2: $call(n_1, n_2)$

(a) direct (b) indirect (c) Legend

Figure 2: There are two error path types from node n_x to n_y

Fig. 2 depicts two control flow structures that could be specified in G. In the following, we call nodes connected by edges of G *succeeding nodes*. If a node calls a method, we say that the control flow path *descends*. Note that the control flow path between n_x and n_y in Fig. 2(a) is directly constructable by succeeding or descending to the next nodes. In contrast, the control flow path in Fig. 2(b) first needs to return to the previous call site n_f from where it can reach n_y (and even n_x) directly. Two nodes (n_x, n_y) can be connected by arbitrary control flow paths. Each control flow path has a set of nodes N_{xy} that contains all nodes in between.

With error tuples like $(\sigma_{write()}, \sigma_{end}) \in P$ from Sec. 2.1 we call the code analysis method $Path_{Lib}(\sigma_{write()}, \sigma_{end})$. To ensure that there are no library events between n_x and n_y, we detect on one path all nodes N_{xy} in between (using $Path(n_x, N_{xy}, n_y)$) and demand that they do not invoke library events (using $noLib(N_{xy})$). In the case $Path_{Lib}(\sigma_a, \sigma_b) \wedge (\sigma_a, \sigma_b) \in P$ is true, we successfully detected a possible error path in the source code.

$$
\begin{aligned}
&Path_{Lib}(n_x, n_y) \triangleq \exists N_{xy} \subseteq N \mid Path(n_x, N_{xy}, n_y) \wedge noLib(N_{xy})\\
&Path(n_x, N_{xy}, n_y) \triangleq \exists n_f \in N, \exists N_{fx}, N_{fy} \subseteq N \mid\\
&\qquad AllPaths_{Desc}(n_f, N_{fx}, n_x) \wedge Path_{Desc}(n_f, N_{fy}, n_y) \wedge N_{xy} = N_{fy} \setminus (N_{fx} \cup \{n_x\})\\
&Path_{Desc}(n_x, N_B, n_y) \triangleq Path_{Succ}(n_x, N_B, n_y) \dot{\vee} Path_{Call}(n_x, N_B, n_y)\\
&Path_{Succ}(n_x, N_B, n_y) \triangleq n_x \notin dom(C) \wedge ((succ(n_x, n_y) \wedge N_B = \emptyset)\\
&\qquad \vee(\exists n_i \in N, \exists N_{B*} \subseteq N \mid succ(n_x, n_i) \wedge Path_{Desc}(n_i, N_{B*}, n_y) \wedge N_B = \{n_i\} \cup N_{B*}))\\
&Path_{Call}(n_x, N_B, n_y) \triangleq (call(n_x, n_y) \wedge N_B = \emptyset) \vee (\\
&\qquad \exists n_t, n_i, n_j \in N, \exists m \in M, \exists (n_x, m) \in C, \exists (m, n_t) \in F_M, \exists N_{B*}, N_{B**} \subseteq N \mid\\
&\qquad\quad (call(n_x, n_i) \wedge Path_{Desc}(n_i, N_{B*}, n_y) \wedge N_B = \{n_i\} \cup N_{B*}) \vee\\
&\qquad\quad (call(n_x, n_i) \wedge Path_{Desc}(n_i, N_{B*}, n_t) \wedge ((succ(n_x, n_y) \wedge N_B = \{n_i, n_t\} \cup N_{B*})\\
&\qquad\qquad \vee(succ(n_x, n_j) \wedge Path_{Desc}(n_j, N_{B**}, n_y) \wedge N_B = \{n_i, n_j, n_t\} \cup N_{B*} \cup N_{B**}))))\\
&noLib(N_B) \triangleq \forall n_i \in N_B \mid (n_i, m) \in C \wedge m \in (M \setminus M_{LE})\\
&call(n_1, n_2) \triangleq \exists m \in M \mid (n_1, m) \in C \wedge (m, n_2) \in B\\
&succ(n_1, n_2) \triangleq (n_1, n_2) \in E
\end{aligned}
$$

Descending the control flow path is easy using G, but ascending again is only possible if one keeps track with the call sites: Only if the call sites in a generic path are known, the next node after a return node can be determined. To keep track with call sites, the analysis specifies a generic path from n_x to n_y based on two descending paths. Both of them start at the same node n_f that precedes n_x and n_y, and that is located at a higher level in the call graph hierarchy. Since the control flow graphs *may* be forked at a node n_f (as shown in Fig. 2(b)), we call n_f fork node. The nodes $n \in N_{xy}$ can then be specified using the difference of two descending path node sets: The minuend is the set of nodes N_{fy} between the n_f and n_y (Fig. 2: grey/dark nodes and n_x); and the subtrahend is the set of nodes N_{fx} between n_f and n_x, including n_x (Fig. 2: dark nodes and n_x). But N_{fx} and N_{fy} are of different kind: While both of them contain nodes between n_f and n_x or n_y, respectively, N_{fx} contains the nodes of all paths between n_f and n_x (specified in $AllPaths_{Desc}$). In contrast, N_{fy} only contains the nodes of one single path between n_f and n_y (specified in $Path_{Desc}$). In the formalization above, $Path_{Desc}$ is stated in detail, and $AllPaths_{Desc}$ is omitted, but can be specified analogously.

The specification $Path_{Desc}$ always respects methods calls when determining next nodes. If a node n_x does not call a method, then $Path_{Desc}$ is based on $Path_{Succ}$. Otherwise – if n_x calls a method – $Path_{Desc}$ is based on $Path_{Call}$. With regard to all possible locations of n_x and n_y in a descending control flow path, $Path_{Succ}$ and $Path_{Call}$ are defined. $Path_{Succ}$ first considers the case that n_x and n_y follow each other directly and hence have no nodes in between. The second case is that n_y follows n_x at some point later in the control flow graph, and a recursive definition is used. Hence, the nodes in between are the union of the directly succeeding node n_i and all the following nodes in N_{B*}. In style of $Path_{Succ}$, $Path_{Call}$ is specified similarly.

The remainder of the specification above states a succession and a call relation. $succ(n_1, n_2)$ is true when the node n_2 succeeds n_1. $call(n_1, n_2)$ is true when n_1 calls a method and n_2 is the first node of this method's body.

2.3 Evaluation

For evaluation we implemented our approach using JTransformer [KHR07] as a meta programming and analysis tool for Java. To verify the implementation we used a test suite that tests every possible correct and incorrect library use of our example, and Java language features like program calls, conditional blocks and loops. To evaluate performance and scalability[1], we extended the Soot framework's analysis source code [VRHS$^+$99] that has a big connected call graph, using polymorphy etc. Table 1 indicates that the performance does not depend on the code size but on the call graph size due to its depth and numerous calls to the same methods.

Project	Lines of Code	Performance (sec.)
Single test case	22	0.004
Test suite (22 test cases)	959	0.312
Soot	12515	4874

Table 1: Scaling performance of the analysis

3 Conclusion

Like Ball et al. [BR02] and others before, we use an API grammar to specify correct behavior. Our work also is related to the work of Hughes et al. [HB07], Tkachuk et al. [TD03], and Jin [Jin07], but for verification we use static code analyses instead of model checking or formal methods.

[1]Tested on an Intel i5 Processor, 4GB RAM; JTransformer's fact building time not included.

Our implementation currently supports libraries that use static and instance methods, and parameters. In addition, language features like polymorphy, condition and loop statements are respected. On the downside, the implementation ignores threads and exception handling, permits recursion and poorly scales to large programs. Nevertheless, our approach as presented here is capable of analysing simple but essential libraries like file stream or socket libraries based on static analyses, and identifies their misuses.

In the future, we will work on supporting error paths of length greater than two and extend the interface grammar to provide additional features to express method parameter constraints or even dependencies of multiple library instances. Regarding the implementation, we concentrate on switching to Soot as an analysis tool, and use collapsed call graphs and more precise static code analyses that take object or control flow context information into account [Mil05].

Acknowledgements

This work is carried out as part of the VirtuOS project. The VirtuOS project is financed by TSB Technologiestiftung Berlin – Zukunftsfonds Berlin Co-financed by the European Union – European fund for regional development.

References

[BR02] Thomas Ball and S K Rajamani. SLIC: A Specification Language for Interface Checking (of C). *Techn Report MSRTR2001*, 21(MSR-TR-2001-21), 2002.

[HB07] Graham Hughes and Tevfik Bultan. Interface grammars for modular software model checking. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 39–49, New York, NY, USA, 2007. ACM.

[Jin07] Ying Jin. Formal Verification of Protocol Properties of Sequential Java Programs. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 475 –482, july 2007.

[KHR07] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd workshop on Linking aspect technology and evolution*, LATE '07, New York, USA, 2007. ACM.

[Mil05] Ana Milanova. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol*, 14:2005, 2005.

[TD03] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 188–197, New York, NY, USA, 2003. ACM.

[VRHS$^+$99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.