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.

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

```java
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();
    }
}
```

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.

We use state machines to specify the interface grammar: $L = (Q_L, \Sigma_L, \Delta_L, q_{0L}, F_L)$ (Fig. 1). $Q_L$ contains all states, $\Sigma_L$ contains all input symbols and $\Delta_L \subseteq Q_L \times \Sigma_L \times Q_L$ contains all transitions. $\Delta_L$ maps from a start state and an input symbol ($dom(\Delta_L) \subseteq Q_L \times \Sigma_L$) to a target state in $Q_L$. $\Sigma_L$ is the set of qualified interface constructor and method names (abbreviated in Fig. 1). $q_{0L}$ 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:

$$(q_1, \sigma_1, q'_1), (q_2, \sigma_2, q'_2) \in \Delta_L$$

$$\sigma_1 = \sigma_2 \Rightarrow (q'_1 = q'_2)$$

Figure 1: Reduced interface grammar of the Java FileOutputStream library
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 $\sigma_{\text{start}}, \sigma_{\text{end}}$, and transitions $\Delta_{\text{noex}}$ to and from $q_{noex}$. We also add an error state $q_{err}$ and transitions $\Delta_{\text{err}}$ from every state to $q_{err}$. If a state lacks an outgoing transition that fires on an input symbol $\sigma_i$, a new transition to $q_{err}$ is added. The state machine $L_X$ remains deterministic and $\Delta_X$ 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 $F_X = \{q_{noex}, q_{err}\}$.

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

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

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

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 $P \subseteq \Sigma_X \times \Delta_X$, 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.

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

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

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

### 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 $\text{assigned}_d(s_1, s_2)$. We then specify transitive assignments with $\text{assigned}(s_1, s_2)$, and define that two symbols $s_x$ and $s_y$ do alias when they both have an assignment symbol $o$ in common.

$$\text{assigned}(s_1, s_2) \triangleq \exists s_i \in S | \text{assigned}_d(s_1, s_2) \lor (\text{assigned}_d(s_1, s_i) \land \text{assigned}(s_i, s_2))$$

$$\text{alias}(s_x, s_y) \triangleq \exists o \in S | \text{assigned}(s_x, o) \land \text{assigned}(s_y, o)$$
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 \( F_P \subseteq N \) 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, \( M_{LE} \subseteq M \) 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 \textit{start} \ and \textit{end} \ (representing the symbols \( \sigma_{\text{start}} \) and \( \sigma_{\text{end}} \) of \( L \)) are added to \( M \). As a predecessor we insert a new first node that calls the method \( \textit{start} \in M \). 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 \( \textit{end} \in M \).

![Figure 2: There are two error path types from node \( n_x \) to \( n_y \)](image)
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 \((\sigma_{\text{write}}(), \sigma_{\text{end}}) \in P\) from Sec. 2.1 we call the code analysis method \( \text{Path}_{\text{Lib}}(\sigma_{\text{write}}(), \sigma_{\text{end}}) \). 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 \( \text{Path}(n_x, N_{xy}, n_y) \)) and demand that they do not invoke library events (using \( \text{noLib}(N_{xy}) \)). In the case \( \text{Path}_{\text{Lib}}(\sigma_a, \sigma_b) \land (\sigma_a, \sigma_b) \in P \) is true, we successfully detected a possible error path in the source code.

\[
\begin{align*}
\text{Path}_{\text{Lib}}(n_x, n_y) &\equiv \exists N_{xy} \subseteq N \mid \text{Path}(n_x, N_{xy}, n_y) \land \text{noLib}(N_{xy}) \\
\text{Path}(n_x, N_{xy}, n_y) &\equiv \exists n_f \in N, \exists N_{fx}, N_{fy} \subseteq N \\
&\quad \bigwedge \text{AllPaths}_{\text{Desc}}(n_f, N_{fx}, n_x) \land \text{Path}_{\text{Desc}}(n_f, N_{fy}, n_y) \land N_{xy} = N_{fy} \setminus (N_{fx} \cup \{n_x\}) \\
\text{Path}_{\text{Desc}}(n_x, N_B, n_y) &\equiv \text{Path}_{\text{Succe}}(n_x, N_B, n_y) \lor \text{Path}_{\text{Call}}(n_x, N_B, n_y) \\
\text{Path}_{\text{Succe}}(n_x, N_B, n_y) &\equiv \exists n_r \notin \text{dom}(C) \land ((\text{succ}(n_x, n_y) \land N_B = \emptyset) \\
&\quad \lor (\exists n_i \in N, \exists N_{B^*} \subseteq N \mid \text{succ}(n_i, n_i) \land \text{Path}_{\text{Desc}}(n_i, N_{B^*}, n_y) \land N_B = \{n_i\} \cup N_{B^*})) \\
\text{Path}_{\text{Call}}(n_x, N_B, n_y) &\equiv (\text{call}(n_x, n_y) \land N_B = \emptyset) \lor (\exists n_t, n_i, n_j \in N, \exists m \in M, \exists (n_x, m) \in C, \exists (m, n_t) \in F_M, \exists N_{B^*}, N_{B^{**}} \subseteq N \\
&\quad \land (\text{call}(n_x, n_t) \land \text{Path}_{\text{Desc}}(n_t, N_{B^{*}}, n_y) \land N_B = \{n_i\} \cup N_{B^*}) \lor \land (\text{call}(n_x, n_i) \land \text{Path}_{\text{Desc}}(n_i, N_{B^{**}}, n_t) \land ((\text{succ}(n_x, n_y) \land N_B = \{n_i, n_t\} \cup N_{B^*}) \\
&\quad \lor (\text{succ}(n_x, n_j) \land \text{Path}_{\text{Desc}}(n_j, N_{B^{**}}, n_y) \land N_B = \{n_i, n_j, n_t\} \cup N_{B^*} \cup N_{B^{**}})))) \\
\text{noLib}(N_B) &\equiv \forall n_i \in N_B \mid (n_i, m) \in C \land m \in (M \setminus M_{\text{LE}}) \\
\text{call}(n_1, n_2) &\equiv \exists m \in M \mid (n_1, m) \in C \land (m, n_2) \in B \\
\text{succ}(n_1, n_2) &\equiv (n_1, n_2) \in E
\end{align*}
\]

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 graph 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 \( n_y \)). 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 \( \text{AllPaths}_{\text{Desc}} \)). 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 scalability1, we extended the Soot framework’s analysis source code [VRHS+99] that has a big connected call graph, using polymorphism 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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Lines of Code</th>
<th>Performance (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single test case</td>
<td>22</td>
<td>0.004</td>
</tr>
<tr>
<td>Test suite (22 test cases)</td>
<td>959</td>
<td>0.312</td>
</tr>
<tr>
<td>Soot</td>
<td>12515</td>
<td>4874</td>
</tr>
</tbody>
</table>

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.

---

1Tested 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 polymorphism, 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


