Program Analysis Course Project on Slicing

David Holtz
Master Program *Computer Science* at University of Stuttgart

***Abstract*—This report presents a novel algorithm based on a data-flow analysis for dynamic, intra-procedural backward slicing of JavaScript programs using JALANGI2. The implementation is evaluated on a set of artificial test cases. The results show >90% reliability and limitations of the presented approach are discussed.**

***Index Terms*—dynamic slicing, JavaScript, Jalangi2**

I. Introduction

Program slicing is a decomposition technique for programs based on data-flow and control-flow analysis [1]. It aims at finding a subset of a program while maintaining the program's behavior at a particular location, hereinafter referred to as *slicing criterion*. More precisely, a slice S of a program P is obtained by deleting zero or more statements from P such that no observer can distinguish between a run of P and a run of its slice S by only looking at the slicing criterion [1]. Obviously, a trivial slice, namely the program itself, always exists. However, most often there exists more than one slice and thus the challenge of slicing is to find *statement-minimal* slices [1]. As outlined by [2], program slicing has been subject of many research papers. The survey investigates typical applications of slicing, with debugging being the most prominent one: It is motivated by the fact, that in case a variable holds a erroneous value at a slicing criterion C, the fault is most likely to be found in the slice of P on C. Beside that, slicing is used for optimizing parallelization or compilers, for testing, software maintenance, reverse engineering [2].

II. Related Work

In the literature, two classes of approaches for program slicing are distinguished: On the one hand, *static slicing* such as Weiser's proposed algorithm does not make any assumptions regarding the input of a program. Instead, the slice is computed for arbitrary input without executing the program [2]. However, a generic algorithm for computing the statement-minimal slice does not exist, which follows from the proof by [1]. On the other hand, *dynamic slicing* fixes the input and computes the slice only for a specific program execution [2].

Sen et al. proposed *JALANGI*, a framework for writing dynamic analyses of JavaScript programs [3]. *JALANGI* is well suited for implementing a dynamic slicing tool, because it features selective record-replay as well as shadow values, which are useful to attach information to any value during execution [3].

III. Approach

The goal of this project is to develop a program analysis for dynamic, intra-procedural backward slicing of JavaScript programs. Given an input program, that contains a single function, the analysis should compute and pretty-print a statement-minimal slice on an in advance defined slicing criterion.

A. Overview

Figure 1 depicts the overall architecture of the proposed solution: Firstly, the input program is parsed and its abstract syntax tree (AST) is preprocessed to extract static information from the input. Secondly, the dynamic analysis executes the input program and by additionally incorporating static artifacts, both data- and control-flow dependencies are traced. We will show, that computing the slice is equivalent to solving the data-flow equations. Finally, the AST is traversed again and only nodes, that are contained in the slice, are kept.

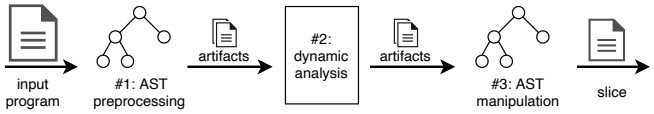

Fig. 1. Overview of the proposed tool for dynamic backward slicing

1) Challenges: Several challenges in designing the program analysis have been encountered: Firstly, multiple variables may point to the same object in memory. Therefore, being able to identify objects unambiguously is a crucial requirement. Secondly, it is hard to reason about the effect of control-flow jumps such as the fall-through in switch-case structures or `break` / `continue` within loops.

Moreover, the JALANGI2 framework poses further challenges: On the one hand, some callback functions lack information: For instance, the 'declare'-callback does not provide the precise line number of a VariableDeclaration node. Neither does the 'conditional'-callback provide information about the kind of branching construct, which makes it difficult to link SwitchCase nodes to their corresponding switch-statement. On the other hand, JALANGI2 does not throw any callback when `break` / `continue` statements are executed, which complicates the control-flow analysis.

2) Methodology: A two-step methodology for the development is proposed as follows: In the first step, a control-flow agnostic analysis is developed, which traces only the data-flow in the input program with respect to the slicing criterion. This includes assignments, object allocation, reads and writes of object properties as well as function invocation. In a second step, this data-flow framework is extended and generalized to capture control-flow dependencies (i.e., branching, loops, exception handling or jumps) as well.

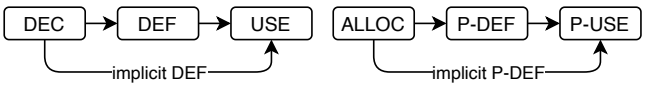

Fig. 2. Topology of variable interactions

Fig. 3. Topology of object interactions

B. Analyzing data-flow dependencies

The analysis maintains a gen-set and a kill-set for every statement, denoted by $gen(s)$ and $kill(s)$ respectively. On the one hand, $kill(s)$ lists all interactions that happen within a statement s. On the other hand, $gen(s)$ lists all interactions, that are required before s to finally execute s. The information of the gen- and kill-sets is accumulated across multiple statements by the sets $RQ_{\text{entry}}(s)$ and $RQ_{\text{exit}}(s)$. The notion of *interactions* summarizes concepts such as reads and writes of variables or object properties:

Definition 1. There are three primitive types of interactions with variables: Whenever a variable appears in the program, it is either declared (DEC), defined (DEF) or used (USE). Interactions with concrete variables are denoted as tuples, e.g. (DEF, x) for the statement `x=2;`.

Definition 2. There are three primitive types of interactions with objects: Objects may be allocated (ALLOC), their properties may be either defined (P-DEF) or used (P-USE). The set of interactions with concrete objects is denoted as a set of tuples, e.g. $\{(\text{ALLOC}, sid), (\text{P-DEF}, sid.x)\}$ for the statement `obj={x: 13};`.

1) Variable interactions: Valid programs must respect the topology illustrated in figure 2. Any variable must be both declared and defined before it can be used. Moreover, it must be declared before it can be defined. This topology is reflected by the gen- and kill-sets. Since a statement s might have an arbitrary number of interactions with variables, both sets are updated incrementally using the following equations:

If statement s declares variable x:

$$gen(s) \coloneqq gen(s) \cup \emptyset$$
$$kill(s) \coloneqq kill(s) \cup \{(DEC, x), (DEF, x)\}$$

or if statement s defines variable x:

$$gen(s) \coloneqq gen(s) \cup \{(DEC, x)\}$$
$$kill(s) \coloneqq kill(s) \cup \{(DEF, x)\}$$

or if statement s uses variable x

$$gen(s) \coloneqq gen(s) \cup \{(DEC, id), (DEF, id)\}$$
$$kill(s) \coloneqq kill(s) \cup \emptyset$$

2) Object interactions: The shadow memory of JALANGI2 generates a unique shadow id (sid) for each object. Any property p of an object sid is uniquely identified by $sid.p$. Similarly to variables, the topology illustrated in figure 3 must be respected and the update equations are defined as follows:

If statement s allocates and object with shadow id sid and arbitrary properties p^*:

$$gen(s) \coloneqq gen(s) \cup \emptyset$$
$$kill(s) \coloneqq kill(s) \cup \{(\text{ALLOC}, sid), (\text{P-DEF}, sid.p^*)\}$$

or if statement s defines a property p of an existing object sid:

$$gen(s) \coloneqq gen(s) \cup \{(\text{ALLOC}, sid)\}$$
$$kill(s) \coloneqq kill(s) \cup \{(\text{P-DEF}, sid.p)\}$$

or if statement s uses property p of an existing object sid:

$$gen(s) \coloneqq gen(s) \cup \{(\text{ALLOC}, sid), (\text{P-DEF}, sid.p)\}$$
$$kill(s) \coloneqq kill(s) \cup \emptyset$$

There remains a special case of data-flow, namely the *implicit property use*: If an object-value is returned and the return statement is marked as the slicing criterion, then all properties are implicitly used (cf. example in Listing 1). In particular, this holds true recursively for nested structures or arrays.

```
1 function sliceMe() {
2   var x = {a:1};
3   x.b = 2;
4   return x; // slice
5 }
```

Listing 1. Input program being equivalent to the statement-minimal slice

```
1 GEN(4)={(x,DEF),(x,DEC),
2         (7,ALLOC),
3         (7.a,P-DEF),
4         (7.b,P-DEF)}
5 KILL(4)={}
```

Listing 2. Implicit property use is reflected by the gen- and kill set (w.l.o.g. assuming sid is 7)

3) Formal definition: The data-flow analysis for dynamic slicing is defined by six generic properties:

- *Domain*: The set of all possible variable and object interaction tuples forms the domain.
- *Direction*: The analysis is classified as a backward analysis, because it reasons about the executions in reverse.
- *Transfer function*: Given by equation (1). Information of s is propagated iff s kills any required interaction for the next statement in the program execution.
- *Meet operator*: Not required and thus left undefined. This is justified by the fact, that a dynamic analysis flattens the execution into a linear sequence of statements, where control flow never merges.
- *Boundary condition*: The slicing criterion refers to a special statement s^*, which serves as a boundary condition.

$$RQ_{\text{Entry}}(s^*) \coloneqq gen(s^*) \quad \text{and} \quad RQ_{\text{Exit}}(s^*) \coloneqq \emptyset$$

- *Initial values*: For all other statements of the program P, the initial value is the empty set.

$$RQ_{\text{Entry}}(s) = RQ_{\text{Exit}}(s) = \emptyset \qquad \forall s \in P \setminus \{s^*\}$$

$$RQ_{\text{Entry}}(s) = \begin{cases} \big(RQ_{\text{Exit}}(s) \setminus kill(s)\big) \cup gen(s) & \text{if } RQ_{\text{Exit}}(s) \cap kill(s) \neq \emptyset \\ RQ_{\text{Exit}}(s) & \text{otherwise} \end{cases} \tag{1}$$

4) Slicing algorithm: We exploit the fact, that computing the slice is closely related to solving the data-flow equation (1) for all statements. Intuitively, if a statement s does not kill any required interactions with respect to the slicing criterion i.e., $RQ_{\text{Exit}}(s) \cap kill(s) \neq \emptyset$, then s is superfluous and therefore not included in the slice. However, all other statements contribute to the data-flow of interest and therefore form the slice. Hence, algorithm 1 is proposed as the slicing algorithm. It requires that the input program has been executed, whereby gen- and kill-sets as well as the *stack* are dynamically computed using JALANGI2.

Algorithm 1 Slicing algorithm

Require: Let $\mathcal{P}^*$ be a list of all executed statements in the program $\mathcal{P}$, represented by a *stack*
Require: Let s^* be the slicing criterion
Require: $gen(s)$ and $kill(s)$ exist $\forall s \in \mathcal{P}^*$
Ensure: $\mathcal{S}$ = set of all statements contained in the slice

```
RQ_Entry(s) ← ∅,  RQ_Exit(s) ← ∅ ∀s ∈ P*      ▷ initial values
RQ_Entry(s*) ← gen(s*)                        ▷ boundary condition
S ← ∅ ∪ {s*}                                  ▷ include s* in the slice
while s ≠ s* do                               ▷ find s* on the stack
    s ← stack.pop()
end while
s_previous ← s                                ▷ initialize loop
while stack is not empty do
    s ← stack.pop()
    RQ_Exit(s) ← RQ_Entry(s_previous)
    if RQ_Exit(s) ∩ kill(s) ≠ ∅ then
        RQ_Entry(s) ← (RQ_Exit(s) \ kill(s)) ∪ gen(s)
        S ← S ∪ {s}                           ▷ include s in the slice
    else
        RQ_Entry(s) ← RQ_Exit(s)
    end if
    s_previous ← s
end while
```

C. Control-flow dependencies

Naively applying the data-flow analysis on programs with control flow leads to severe underapproximations of the slice. Two types of errors occur (cf. example in Listing 3-4):

- Control-flow statements are neglected
- Secondary data-flow dependencies are not propagated

```
1  function sliceMe() {
2     var x = 2;
3     var y = 1;
4     while (x < 5) {
5        y *= x;
6        if (y > 2) {
7           x++;
8        }
9     }
10    return x; // slice
11 }
```

Listing 3. Input program being equivalent to the statement-minimal slice

```
1  function sliceMe() {
2     var x = 2;
3     x++;
4     return x; // slice
5
6
7
8
9
10
11 }
```

Listing 4. Dataflow-only slice: Branching and secondary data-flow (e.g. variable y) is missing

1) Remedy: This issue is resolved by extending the condition for propagating gen- and kill-sets within equation (1). Intuitively, a branching point is included in the slice $\mathcal{S}$ if any of its control-dependent nodes are already included in the slice. Let $\text{CD}(s', s)$ be a predicate, that is true iff s' is control-dependent on s. CD contains static information about the input program and is obtained by preprocessing its AST. Hence, the extended condition is formally stated:

$$RQ_{\text{Exit}}(s) \cap kill(s) \neq \emptyset \ \vee \ \exists s' \in \mathcal{S} : \text{CD}(s', s)$$

As a result of the generalization, programs with branching or loops such as Listing 3 are sliced correctly. In the example, the if-statement of line 6 is included in the slice, since its control-dependent statement `x++;` of line 7 is already contained. Since line 6 uses y, secondary control-flow dependencies are also propagated to include line 5 and 3.

2) Refinements: JavaScript features sophisticated control-flow constructs, which require additional refinements. Therefore, further static information of the AST is incorporated:

- AST preprocessing reasons about the fall-trough semantics of switch-case to determine control-dependencies correctly.
- To cope with the complexity of jump instructions, it is assumed that `break` / `continue` statements are included in the slice, whenever they are executed. Due to the lack of a corresponding callback, superior branching statements (e.g. `if`, `for`, `while`, etc.) serve as a trigger for including the jump statements. Using the control-dependencies, AST preprocessing computes a mapping from trigger statements to jump statements.
- Analogously, it is assumed that a throw-statement remains in the slice if it is executed. AST preprocessing computes a mapping from throw-statements to the respective catch-clauses to ensure that every thrown exceptions is caught.

IV. Implementation

The program analysis for dynamic backward slicing is implemented with Node.js (v17.2.0). It is delivered as a command line tool and matches the provided interface of this project's code repository. The following libraries are required:

- JALANGI2 (v0.2.6): framework for writing dynamic analyses for JavaScript
- Acorn (v8.7.0): parses JavaScript programs into an AST
- Estraverse (v5.3.0): traverses and manipulates the AST
- Escodegen (v.2.0.0): ECMAScript code generator

Moreover, the development of the program analysis is aligned to an incremental software development process model: Using GitHub Actions, individual increments have been integrated and tested continuously. Tests are therefore written in a behavior driven manner and are executed by the Mocha test framework (v9.2.0) for Node.js.

V. Evaluation

The implementation is evaluated on a basis of 54 test cases, 37 of them being custom created to cover both common and exotic language features of JavaScript. The test cases

are manually designed to trigger distinct behavior and not to overlap whenever possible. The obtained results are presented by table I: For each category, the number of produced statement-minimal slices (exact), overapproximations (+) and underapproximations (-) are given.

Category	exact	+	-	total
acceptance: milestones and progress meeting	17	0	0	17
data-flow: assignment, (nested) objects, arrays, arithmetics, strings, regex	10	0	2	12
control-flow: branching, loops, switch-case (fall-through), jumps	11	2	0	13
exception handling: try-catch-finally, throw	11	1	0	12

TABLE I. Overview of the evaluation

VI. Discussion

To summarize, the results confirm the applicability of the presented approach. The analysis was able to compute statement-minimal slices for 49 out of 54 test cases, which corresponds to $> 90\%$ reliability on the sample. In particular, every provided acceptance test is correctly sliced. However, the evaluation has also revealed limitations of the proposed program analysis, which are investigated in the following.

A. Limitations

To gain a systematic understanding of the analysis' blind spots, the observed limitations are categorized as follows:

1) Scope limitations: Due to the intra-procedural setting, slices of programs with function calls may be underapproximated. On the one hand, the analysis reasons correctly about parameters being passed to or return values being retrieved from function calls. Therefore, pure query methods, that do not alter any data, are correctly handled by the slicer. On the other hand, command methods are not supported, since the analysis does not reason about any side-effects that occur within the called method. For instance, the analysis is unable to capture that calling the `push()` method on an array changes its data.

2) Syntax limitations: Although the analysis is capable of handling a large subset of ECMAScript 5th edition (ES5) language features, the following control-flow constructs are not supported: Calls to `eval` and `with` statements, labeled statements or `debugger` statements. Regarding data-flow, the `delete` operator is not supported, as it is hard to reason about the consequences of deleting object properties. Omitting or including the operator might result in over- and underapproximated slices depending on the intention of the deletion.

3) Line-based approach: Contrary to the data-flow equations presented in section III-B, the analysis implements algorithm 1 using line numbers rather than unique identifiers of the statements s. Reducing statements to the line numbers of their occurrence simplifies the implementation significantly, because there is no need to precisely identify nodes of the program's AST. However, this comes at the expense of not being able to differentiate between several statements within one line. For instance, the following two assignments `x=1, y=2;` are treated as a atomic piece of code, i.e., either both or no variable is assigned. In particular, both the gen- and kill-sets of the line are also propagated in a atomic manner, which might result in overapproximated slices.

4) Overapproximation by assumption: Section III-C introduced conservative assumptions about jump statements in the control-flow (e.g. `break`, `continue` or `throw`). This reduces the analysis' complexity significantly, because deciding whether a jump changes the overall behavior of a program is difficult. Moreover, it avoids underapproximations and makes the slicing algorithm sound. Counterexamples are provided in Listing 5 and 7: The analysis overapproximated the slice due to too conservative assumption.

```
1 function sliceMe() {
2    var x = 0;
3    while (x < 5) {
4       x++;
5       continue;
6    }
7    return x; // slice
8 }
```

Listing 5. Input program being equivalent to the computed slice

```
1 function sliceMe() {
2    var x = 0;
3    while (x < 5) {
4       x++;
5
6    }
7    return x; // slice
8 }
```

Listing 6. Ideal slice does not contain trailing continue statement

```
1 function sliceMe() {
2    var x = 0;
3    try {
4       x++;
5       throw Error();
6    } catch(error) {
7    }
8    return x; // slice
9 }
```

Listing 7. Input program being equivalent to the computed slice

```
1 function sliceMe() {
2    var x = 0;
3    try {
4       x++;
5    }
6    return x; // slice
7
8
9 }
```

Listing 8. Statement-minimal slice does neither contain the throw statement nor the catch block

B. Mitigation and outlook

Although not being statement-minimal, all observed overapproximations are valid slices of the respective input program. Regarding the use case of debugging, limitations 3 and 4 are not considered critical and could be even helpful, because the slice is presented in a less alienated way and control-flow is made explicit. On the contrary, scope and syntax limitations might cause underapproximations (i.e. incorrect slices) and therefore restrict the set of admissible input programs. As a next step, the algorithm should be extended to support the latest ECMAScript standard as well as the `delete` operator. It is anticipated, that this requires only minor changes in the proposed framework. However, extending the scope to inter-procedural slicing requires a more exhaustive analysis and could be subject of another research project.

References

[1] M. Weiser, "Program Slicing," IEEE Transactions on Software Engineering, vol. SE-10, pp. 352–358, July 1984.

[2] F. Tip, "A survey of program slicing techniques", Journal of Programming Languages, 3rd ed., pp.121–189, 1995.

[3] K. Sen, S. Kalasapur, T. Brutch and S. Gibbs, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript", ESEC/FSE, Saint Petersburg, Russia, 2013