Thesis
Evaluation of Static JavaScript Call Graph Algorithms

Jorryt-Jan Dijkstra
jjdijkstra1@gmail.com

August 5, 2014, 75 pages

Supervisor: Tijs van der Storm
Host organisation: Centrum Wiskunde & Informatica

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

This thesis consists of a replication study in which two algorithms to compute JavaScript call graphs have been implemented and evaluated. Existing IDE support for JavaScript is hampered due to the dynamic nature of the language. Previous studies partially solve call graph computation for JavaScript, but come with a disappointing performance. One of the two algorithms does not reason about interprocedural flow, except for immediately-invoked functions. The other algorithm reasons about interprocedural flow, where it considers all possible flow paths. The two call graph algorithms have been implemented in Rascal. A JavaScript parser for Rascal has been implemented as preliminary work. Furthermore flow graph algorithms were implemented in order to compute the call graphs. The call graph algorithms were ran with similar input scripts as the original study. Evaluation of the computed call graphs has been done by comparing computed call graphs to dynamic call graphs. A tool has been implemented to instrument the input scripts, so that dynamic call graph recording could take place. The evaluation has been done with a precision and recall formula on a per call site basis, where the precision formula indicates the ratio of correctly computed call targets to all computed call targets. The recall formula indicates the ratio of correctly computed call targets to the total call targets that were recorded in the dynamic call graph. Both these formulas were applied for each call site that was recorded in the dynamic call graph. The results were then averaged. The averaged results were compared to two different sources. One source was the data of the original study, whereas the other data came about by applying the same formulas on the call graphs that were used in the original study. The results of the latter source, turned out not to be corresponding to the data of the paper. The problem is confirmed by the authors, which is being further investigated by them. The precision and recall of the computed call graphs turned out to result in lower values than the previous study, with a maximum precision deviation of $\sim 25\%$ and a maximum recall deviation of $\sim 11\%$. The algorithm that reasons about interprocedural flow in the form of immediately-invoked functions, had an average precision of 74% and an average recall of of 89%. The other algorithm that reasons about all interprocedural flow, had an average precision of 75% and an average recall of 93%. Another form of evaluation has also been undertaken by calculating precision and recall of call relationships (edges from call site to call target) rather than averaging over call sites. This evaluation resulted in lower precision and recall values, where the algorithm for interprocedural immediately-invoked function resulted in an average of 64% in terms of precision and 87% in terms of recall. The fully interprocedural algorithm resulted in an average precision of 43% and an average recall of 91%. The call relationships evaluation indicates that the computed call graphs are quite complete, but more polluted than the other results and original study suggest. The largest contributor to lower precision values was found to be one of the design decisions of the algorithms. This design decision yields the merging of similarly named functions, which causes addition of spurious edges. Finally the thesis presents ideas on how to improve the call graph algorithms, on how to improve the JavaScript parser and on future research work for JavaScript call graph analysis.

Preface

The wide usage of JavaScript today in combination with the hampered tooling, have been a practical motivation for this thesis. Moreover the strong focus on mathematics, algorithms, a programming language and my growing interest in computing science made me feel like this would be a challenging project for me. I hope this thesis provides some insight that the previous study has not, given the fact that this thesis partially consists of a replication. Furthermore I am happy to hear that at least some of the work of this thesis will be adopted to the practical field; prerequisite work on the JavaScript grammar will be implemented in the Rascal standard.

I would like to thank Tijs van der Storm for his supervision, feedback and the initial Rascal JavaScript work. Furthermore I would like to thank Max Schäfer for his valuable responses to questions regarding the study that has been replicated. Additionally I would like to thank Jurgen Vinju and Davy Landman, for both helping me out with questions regarding Rascal grammars and sharing their general thoughts. Thanks to Sander Benschop for the collaboration on the JavaScript grammar as well as the valuable discussions about the replication. Thanks to Vadim Zaytsev for his work on a generic LaTeX layout. Thanks to Çiğdem Aytekin for her thorough review of the thesis. Finally I would like to thank my family for their everlasting care and support.

Chapter 1

Introduction

This thesis document describes a research, its process and supportive work during the period of february 2014 until august 2014. The research has been conducted at CWI (Centrum Wiskunde en Informatica) and consisted of a replication of an existing research. The replicated research is called *Efficient Construction of Approximate Call Graphs for JavaScript IDE Services* by Feldthaus et al. [1]. Their research suggests two similar algorithms to create call graphs for JavaScript. Their results contain computed call graphs with sometimes over 90% precision. The research by Feldthaus et al. will further on be referred to as **ACG** (coming from the term Approximate Call Graphs).
Preliminary work had to be done prior to the research of this thesis. This work entailed creating a grammar for parsing JavaScript in the Rascal programming language. This came with several difficulties (like automatic semicolon insertion). The Rascal language is mainly used for software engineering research at CWI. It was initially thought that this would save time due to its functionality for graphs and functional programming features. Writing a good enough grammar did however turn out to take longer than thought. This grammar could be considered a good contribution to the Rascal standard library and could thus serve for upcoming researches at CWI.
The research replication itself consisted of implementing both JavaScript static call graph algorithm specifications of ACG and comparing and validating their data with the results of the replication. Dynamic call graphs had to be obtained, in order to validate the computed call graphs. These dynamic call graphs were instrumented versions of the input scripts, which would log runtime call relationships. The set of input scripts for the algorithms has been similar to those of the ACG study. Average precision and recall values have been computed by comparing the call targets of the static call graph to those of the dynamic call graph on a per call site basis. The average precision and recall values of the static call graphs often deviated from the ACG research. Precision and recall has also been computed for the published call graphs from the ACG research, which are confirmed to be the call graphs used in their study. Output values deviated from their own paper, which problem has been confirmed by the authors and is under further investigation by them. Moreover the precision and recall of the extracted call graphs often turned out to be closer to the results of this thesis. Furthermore deviations were most often found to be caused by differing input data and by a design decision for the call graph algorithms. This design decision generalizes similarly named functions for scalability reasons.
Additionally other comparisons have been done, which indicated that the algorithms were less precise than the previous study and the original comparison suggests. The original formulas averages precision and recall over call sites, whilst the other comparison computed precision and recall for the sets of call graph edges (from call site to target). It has been found however, that most of the dynamic call graph data has been present in the static call graphs. This indicates that the static call graphs were simply polluted with spurious edges, but did contain the largest part of the dynamic call graph relations.

The background and related work for this thesis will be described in the upcoming chapter. This can be considered preliminary information in order to understand the thesis. Apart from that, some knowledge of graph theory, JavaScript and logic is advisable. Afterwards the addressed problem will be described. The subsequent chapter elaborates the original study and its results. A description

of the replication follows up, which includes the research questions. Afterwards two chapters will be dedicated to the specification of the algorithms. Then the implementation of the preliminary grammar will be elaborated, in combination with implementation details of the static and dynamic call graphs. The results and data are then presented. Results and the threats to validity are then discussed in a discussion chapter. Finally, in the conclusion the thesis answers the research question. Furthermore suggestions for future work are added as well as some appendices with instructions on preparing the subject scripts and an overview of the gathered data.

Chapter 2

Background and Related Work

This chapter will describe what a call graph entails as well as some of its common terminology. Furthermore it will shortly describe the JavaScript programming language and its characteristics. Call graphs for JavaScript will be further discussed in the problem analysis.

2.1 Call Graph Definition

Call graphs are indispensable graphs for programming languages. These graphs entail the runtime calling relationships among a program's procedures [2, p. 2]. They are used for various purposes, including optimization by compilers [3, p. 159], software understandability by visualisation [4, p. 1] and IDE functionality (like jump to declaration) [1, p. 1].
In a call graph a *call site*, is the vertex of the function call, whilst the *target* is the vertex of a function that is being called. A call site may invoke different targets, which is called a *polymorphic call site*, whereas a *monomorphic call site* only calls one target. A function may be called by different call sites. The "Compilers: Principles, techniques and Tools" book gives a clear specification that we can use to illustrate an example [5, p. 904]:

- There is one node for each procedure in the program
- There is one node for each call site, that is, a place in the program where a procedure is invoked.
- If call site c may call procedure p, then there is an edge from the node for c to the node for p

To illustrate how source code translates to a call graph, an example snippet (in JavaScript) has been made. In this snippet a sort function decides which function to call, depending on the size of the input. Afterwards it calls the function it decided to call:

```
1 function sort(smallSorter, largeSorter, inputToSort) {
2   var sortExecute = (inputToSort.length > 1000) ? largeSorter : smallSorter;
3   return sortExecute(inputToSort);
4 }
5 function smallSortFunction(input) { ... }
6 function largeSortFunction(input) { ... }
7 sort(smallSortFunction, largeSortFunction, ...);
```

The following call graph represents this snippet:

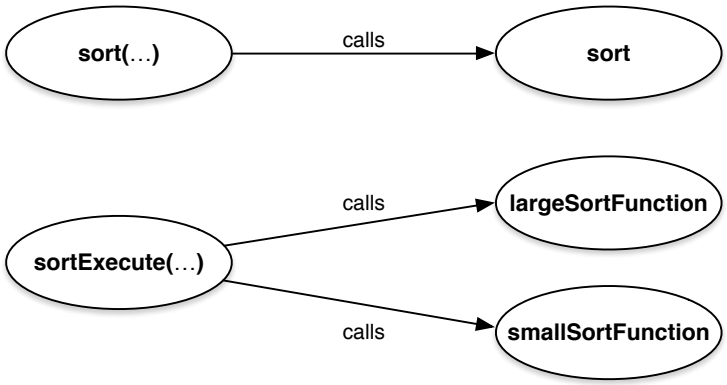

Figure 2.1: A call graph for the given sorting code

The demonstration above shows how the `sortExecute` (line #3) call site is polymorphic. The correct call target is selected depending on the input. It is common that position information (e.g. line number, filename, file offset etc) of call sites is stored with the vertices, so that each call site can be uniquely identified based on its position. In this example the call site on line #7 calls the sort function on line #1. The call site on line #3 calls the functions on line #5 and #6.
Call graphs come in two general forms: a static call graph and a dynamic call graph. A static call graph comes about, purely by statically analyzing a program and thus without running it. Getting to such static call graph is difficult and often comes with compromises that affect its validity or completeness. Unfortunately determining the exact static call graph of a program is an undecidable problem [3, p. 171].
Dynamic call graphs are the opposite of static call graphs. These type of call graphs are a recording of a possible execution of a program. They entail all the calling relationships that were existent during runtime, and are therefore snapshots. The *ideal call graph* is considered to be the union of all dynamic call graphs of all possible executions of a program [6, p. 2]. Given the fact that dynamic call graphs are snapshots of a possible execution, this means that a dynamic call graph of the given code snippet might only contain a relationship between one of the sort functions and its call site. That could be the case when inputs with less than 1000 or only inputs with more than 1000 elements are passed to the `sort` function (see line #2 of the snippet).

2.2 Static Call Graph Computation

Call graphs can easily be determined when all invoked procedures are statically bound to procedure constants [7, p. 2]. For languages where dynamically bound calls can occur (such as dynamic dispatch) computing static call graphs is far from trivial. Contextual knowledge is required in order to determine which call site calls which function. Aho et al. state the following about this problem: "When there is overriding of methods in subclasses, a use of method m may refer to any of a number of different methods, depending on the subclass of the receiver object to which it was applied. The use of such virtual method invocations means that we need to know the type of the receiver before we can determine which method is invoked" [5, p. 904]. Furthermore they mention the following: "the presence of references or pointers to functions or methods requires us to get a static approximation of the potential values of all procedure parameters, function pointers, and receiver object types. To make an accurate approximation, interprocedural analysis is necessary" [5, p. 904]. In summary modern programming languages, which are often object-oriented or dynamic, require prior analysis in order to obtain a call graph. The problem of requiring prior analysis to compute a call graph, can be referred to as *call graph analysis* [7, p. 2]. The interprocedural analysis Aho et al. refer to, reasons about the procedures of an entire program and their calling relationships [8, p. 3] [5, p. 903]. Interprocedural flow can be stored in a flow graph. A flow graph is a graphical representation of a program in which the nodes of the graph are basic blocks, where the edges of the graph show how control can flow among the blocks [5, p. 578]. The same concept is used for intraprocedural flow, which

is solely reasons about flow within procedures, rather than among them. Interprocedural flow analysis can be done *flow-sensitive* and *flow-insensitive*. Flow-sensitive analysis means that intraprocedural control flow information is considered during the analysis. Flow-insensitive analysis does not consider intraprocedural flow and is therefore less precise in general [9, p. 849]. Control flow refers to the order of execution of instructions in a program.

2.2.1 Terminology

Conservativeness and soundness are two typical terms for call graphs, which both say something about the characteristics of a computed call graph. A conservative call graph maintains the invariant that if there is a possible execution of program P such that function f is called from call site c, then $f \in F(c)$, where $F(c)$ is the set of functions that can be called from call site c [10, p. 2]. It can therefore be an overestimation. A conservative static call graph is a superset of the *ideal call graph* [6, p. 2]. The *ideal call graph* is the union of all dynamic call graphs of all possible executions of the program [6, p. 2]. For a call graph to be sound, it must safely approximate any program execution [2, p. 698]. A call graph is sound if it is equal to or more conservative than the ideal call graph [11, p. 4].

2.3 The JavaScript Language

JavaScript is a widely used programming language for the web. The usage of JavaScript goes beyond web browsers, as it gets popular for server-side programming (Node.js), writing cross-platform mobile apps with frameworks like Cordova and even for implementing desktop applications [1, p. 1]. JavaScript is a high-level, dynamic and untyped interpreted programming language [12, p. 19]. It derives its syntax from Java, its first-class functions from Scheme and its prototype-based inheritance from Self [12, p. 19]. First-class functions imply that functions may be used within data-structures, as arguments for functions, stored by variables and as return values [13, p. 84]. Prototype-based inheritance is a dynamic form of inheritance, where each object inherits properties from its prototype (which are prototype objects). Changes in the prototype cascade to their inheritors, meaning that existing objects can be augmented with new properties of their prototype, even after object creation [12, p. 207]. With this prototype system, each object inherits from the `Object` prototype. The same goes for other existing prototypes, for example: the `Date` prototype, inherits from the `Object` prototype. This means that instances of `Date` inherit properties from both the `Object` and `Date` prototype. The language specification for the JavaScript language is called the ECMAScript standard [14, p. 1]. An object in JavaScript is a collection of properties, where each property has a name and value [12, p. 29]. These names and values can be primitive values (including functions) or objects themselves.

Chapter 3

Problem Analysis

In this chapter the first section will clarify the obstacles that hamper call graph computation for JavaScript. It also explains why traditional methods do not work. Afterwards the two call graph algorithms for JavaScript from ACG will be discussed, which are the algorithms that were reimplemented and used in this research. Finally the difficulties and challenges for replicating the research will be stated. Note that ACG refers to the original study by Feldthaus et al.

3.1 JavaScript Call Graph Analysis

In Java call graphs can be efficiently constructed using class hierarchy, which uses type information to build a call graph [1, p. 1]. These algorithms are not directly applicable for JavaScript due to the dynamic typing and prototype-based inheritance [1, p. 1]. With this method first-class functions can not be handled easily either, as Java does not have native support for first-class functions. Existing analyses like CFA (Control Flow Analysis) and Anderson's points-to analysis that statically approximate flow of data are not fast enough for interactive use [1, p. 1]. In order to have an accurate idea of which functions are called, flow-sensitivity is necessary (previously described in section 2.2).
As JavaScripts is extremely popular [14, p. 15], it is important to have mature IDE and tool support. So far JavaScript support in IDE tooling seems to be fairly bare-bone [1, p. 1]. There has been some work on more principled tools and analyses, but unfortunately these approaches do not scale to real-world programs [1, p. 1]. Due to the dynamic nature of JavaScript, it is hard to do static analysis [15, p. 1] [16, p. 455]. Some of the JavaScript features that contribute to this difficulty are:

- First-class functions / Higher order functions [16, p. 455]
- Prototype-based inheritance [16, p. 455]
- Type coercion [17, p. 20] [16, p. 455]
- Dynamic code evaluation (e.g. eval) [16, p. 437]
- Arity mismatching [16, p. 439]

So far first-class functions and prototype-based inheritance have been explained. Type coercion yields the implicit conversion of a value to another type, which occurs for instance when comparing values. Values that are not booleans can therefore be considered to be true or false (known as truthy or falsy). Values are always considered to be true unless they are an element of the predefined falsy set, which includes values like `undefined` and `false` [12, p. 40]. This coercion requires knowledge of the types an operator is dealing with and the way it evaluates those.
Dynamic code evaluation consists of interpreting strings of JavaScript code and evaluate them to produce a value (with possible side-effects) [12, p. 79]. This comes with a certain unpredictability: for example it may be injected external code or be user input dependent. The presence of eval and other constructs for executing dynamically generated code means that a (useful) static analysis for JavaScript cannot be sound [16, p. 3]. According to Richards et al. `eval` is frequently used and often

comes with unpredictable behavior [15, p. 11].
The last element from the list, arity mismatching, means that functions may be invoked with any given amount of arguments regardless of the amount of formally defined parameters. Too few arguments result in the rest of the parameters being assigned to the value of `undefined`, whilst too many parameters can still be accessed via the so called `arguments` object [16, p. 5]. This `arguments` object is basically an object that is always present in function scope and represents the provided parameters. This arity mismatching makes it impossible to narrow down the set of functions that may be invoked from a given call site [16, p. 5]. To give an indication on what arity mismatching entails, an example code has been set up:

```
1 function f(x, y) { ... }
2 f(); // arity mismatch
3 f(1, 2, 3, 4, 5, 6); // arity mismatch
4 f(1, 2); // matching arity
```

The example illustrates that we can call f with an amount of arguments that deviates from the definition of f. Any arguments (including ones that exceed the amount of formal parameters) can be accessed within function f, using the `arguments` object in JavaScript.

3.2 JavaScript Call Graph Algorithms

ACG proposes two call graph algorithms that should diminish and overcome some of the mentioned analysis issues (which is discussed in section 3.1). Their results indicate that the emerged call graphs are quite sound for a variety of input programs. The difference between the two algorithms is that one is a so called pessimistic version, which consists of a non-iterative algorithm. Whilst the other algorithm is an optimistic version that iterates to a fix-point. The idea of a pessimistic and optimistic approach is based on a call graph construction framework that is presented by Grove and Chambers [2]. Their work will be discussed to get an insight on what pessimistic and optimistic call graph algorithms yield. According to Grove and Chambers, call graph construction algorithms typically rely on three different quantities that are circular dependent [2, p. 688]:

- The receiver class sets
- The program call graph
- An interprocedural analysis

The receiver class sets (as mentioned in section 2.2) depend on interprocedural analysis, but interprocedural analysis depends on a program call graph (we need to know which call site calls which exact target in order to determine the flow). The receiver class sets are the sets of receiver types. Polymorphism and dynamic dispatch require contextual knowledge to determine the exact target of a call. ACG describes this problem as the following: "Like any flow analysis, our analysis faces a chicken-and-egg problem: to propagate (abstract) argument and return values between caller and call site we need a call graph, yet a call graph is what we are trying to compute" [1, p. 2]. In order to break this circular dependency chain, an initial assumption has to be made for one of the quantities. Grove and Chambers state the following about pessimistic and optimistic call graph algorithms [2, p. 688]:

- "We can make a pessimistic (but sound) assumption. This approach breaks the circularity by making a conservative assumption for one of the three quantities and then computing the other two. For example, a compiler could perform no interprocedural analysis, assume that all statically type-correct receiver classes are possible at every call site, and in a single pass over the program construct a sound call graph. Similarly, intraprocedural class analysis could be performed within each procedure (making conservative assumptions about the interprocedural flow of classes) to slightly improve the receiver class sets before constructing the call graph. This overall process can be repeated iteratively to further improve the precision of the final call

graph by using the current pessimistic solution to make a new, pessimistic assumption for one of the quantities, and then using the new approximation to compute a better, but still pessimistic, solution. Although the simplicity of this approach is attractive, it may result in call graphs that are too imprecise to enable effective interprocedural analysis."

- "We can make an optimistic (but likely unsound) assumption, and then iterate to a sound fixed-point solution. Just as in the pessimistic approach, an initial guess is made for one of the three quantities giving rise to values for the other two quantities. The only fundamental difference is that because the initial guess may have been unsound, after the initial values are computed further rounds of iteration may be required to reach a fixed point. As an example, many call graph construction algorithms make the initial optimistic assumption that all receiver class sets are empty and that the main routine is the only procedure in the call graph. Based on this assumption, the main routine is analyzed, and in the process it may be discovered that in fact other procedures are reachable and/or some class sets contain additional elements, thus requiring further analysis. The optimistic approach can yield more precise call graphs (and receiver class sets) than the pessimistic one, but is more complicated and may be more computationally expensive."

The pessimistic call graph proposed by ACG does not reason about interprocedural analysis, except when the call site can be determined purely locally. These are in the form of so called *one-shot closures*, which are functions that are instantly invoked after they are defined [18, p. 115]. The assumption in this case, are the local call targets, because it is certain which call target is executed by a *one-shot closure* call. It does however consider functions that are known on the scope. Therefore not only one-shot closures can be resolved. What is meant is that interprocedural flow in the form of return values or arguments as one-shot closures, will be further analyzed with the pessimistic call graph analysis. Callbacks are therefore left unconsidered. Callbacks are functions that will be invoked when an event occurs [19, p. 1]. As JavaScript supports first-class functions, functions that are passed as arguments are used as a callback within another function.
The optimistic call graph that is proposed by ACG starts with an empty call graph and iterates to a fix-point. It starts with a partially interprocedural flow graph, where arguments and parameters still have to be connected, as well as return values and their result. It then has the repetitive process of connecting edges (arguments to parameters and return values to function calls) and then applying a transitive closures. This process iterates to a fix-point, where new discovered flows lead to additions in the call graph. The fix-point simply yields no other flows have been discovered and thus no other possible functions can be called according to the algorithm. The difference with the pessimistic algorithm comes forward with callback functions, where first-class functions are passed on between procedures. The pessimistic analysis does not reason about callbacks, whilst the optimistic does.
Both algorithms come with limitations as several concessions have been done in order to keep them scalable. The following limiting characteristics (referred to as **design decisions** further in the thesis) have been stated by ACG, which apply for both algorithms:

1. The analysis ignores the enclosing object of properties, when it analyzes properties. Meaning that it uses a single abstract location per property name. This is referred to as field-based analysis. This analysis implies that call targets with the same name in different objects, will be indistinguishable by the analysis (e.g. `objectA.toString()` and `objectB.toString()` are considered equal by the algorithms). Therefore a function call to a `toString` property, always yields two edges in this case. One to the `toString` vertex of *objectA* and one to the `toString` vertex of *objectB*. Properties can thus be seen as fields of an object.

2. It only tracks function objects and does not reason about non-functional values. Which implies that value types are not tracked nor influence the call graph. A simple example is the following:

```
1 function f() {
2   var x = 0;
3   if (x == 1) g();
4 }
```

The code above demonstrates that function g will never be executed. This however is not taken into consideration due to solely tracking function objects.

3. It ignores dynamic property accesses, i.e. property reads and writes using bracket syntax (e.g. `a[b] = x`). This is due to previous research, indicating that dynamic property access are often field copy operations [1, p. 2]. If we assume this is true, this can be considered to have no impact, as properties are merged to one representative set (as described in decision #1). Furthermore it ignores the side-effects of getters and setters in JavaScript, which have been found to be uncommon in the analyzed programs (section 8.3). These getter and setters are usually invoked when one simply assigns a property or refers to a property.

4. It ignores dynamic evaluated code like `eval` and `new Function`. The difficulty of dynamic evaluated code has been discussed before in section 3.1, and is excluded from the call graph analysis by ACG. The `new Function` expression creates a new function in the current scope, based on a string containing its code. With the proposed algorithms the effects of spawned functions remain untracked, nor will they be considered true functions.

An example snippet has been included in appendix B with its pessimistic and optimistic call graph. This provides insight in the call graphs that thesis works towards.

3.3 Replication Challenges

Replicating existing studies may be experienced as difficult, as implementation details often lack and environments may differ. The replication of the ACG study came with several difficulties. One of the difficulties was that the existing tooling did not support JavaScript parsing. This would be a prerequisite for reimplementing the algorithms. Implementing a correct JavaScript parser in Rascal has thus been one of the key challenges to conduct the research. Furthermore it was necessary to implement scoping, prior to creating flow graphs and call graphs. Hoisting and overriding (see section 6.3) of JavaScript variables were topics to read about and to be considered strictly. An accurate environment for the algorithms could only be provided by taking these JavaScript characteristics into account. Moreover dynamic call graph construction seemed to be one of the most difficult problems, as the authors were not able to give insight in such, due to licensing issues. Instrumenting code to obtain dynamic call graphs without changing its actual semantics, has been experienced as the biggest problem. Obtaining correct call relationships has also been a tough challenge for the dynamic call graphs.

Chapter 4

Original Study

The proposed call graph algorithms by the ACG paper (clarified in section 3.2), have been applied on a variety of ten input scripts. In this chapter the research questions and their results will be discussed. The input scripts rely on different frameworks (jQuery, Mootools and Prototype), whereas some of them do not utilize a framework at all. When the experiment was conducted, jQuery, Mootools and Prototype were the most widely-used frameworks, according to a survey of JavaScript library usage [1, p. 7]. The subject scripts were medium to large browser-based JavaScript applications covering a number of different domains, including games (Beslimed, Pacman, Pong), visualizations (3dmodel, Coolclock), editors (HTMLEdit, Markitup), a presentation library (Flotr), a calendar app (Fullcalendar), and a PDF viewer (PDFjs) [1, p. 7].

4.1 Research Questions

The ACG paper does not state any research questions in advance. It does however state evaluation questions afterwards, based on their implementation (in CoffeeScript) of the algorithms. The following questions have been mentioned:

- How scalable are our techniques?
- How accurate are the computed call graphs?
- Are our techniques suitable for building IDE services?

4.2 Results

This paragraph will answer the research questions of the ACG paper, based on their own evaluation. The results are summarized, as they are discussed later to compare it to the findings in this thesis.

The first question is answered by measuring the average time of both the pessimistic and optimistic call graph algorithms for each of the input scripts. For each of the programs, both call graph algorithms were ran ten times, in order to get a decent average. They find that their analysis scales well, as the largest input program (PDFjs) with over 30.000 lines of code, is analyzed in less than 9 seconds average, using the pessimistic computation. The optimistic approach took an average of 18 seconds for PDFjs. This is including about 3 seconds of parsing time. Smaller input programs took significantly less time. Therefore, they reason that the pessimistic analysis would be fast enough for IDE usage, especially due to the already available abstract syntax tree (AST). In other words, scripts are typically already parsed in an IDE, which means the algorithms do not need to parse scripts themselves. They consider both analyses very well scalable.

To answer the second research question, it is important to know that the exact static call graph of a program is an undecidable problem [3, p. 171]. The authors decided to compare their static call graph with a dynamic call graph. The input scripts were copied and instrumented to record the observed call targets for every call that it encountered at runtime. By measuring the amount of covered functions, they determined what the function coverage was. For all cases, except for one, the coverage was considered reasonable ($> 60\%$). The used formula for function coverage was $\frac{O}{T} \times 100\%$ where O represents the observed non-framework functions and T represents the total statically counted non-framework functions.
The recall and precision with respect to the dynamically observed call targets has been calculated afterwards. This is done by averaging a precision and recall formula over call sites. Recall for a call site has been determined with the following formula: $\frac{|D \cap S|}{|D|}$ (the percentage of correctly identified targets in the static analysis among all dynamic call graph targets). With D being the set of targets of a given call site in the dynamic call graph and S the set of targets for a given call site. The following formula has been used to calculate the precision for a call site: $\frac{|D \cap S|}{|S|}$ (the percentage of true call targets among all computed targets). These formulas have been applied for each call site that was present in the dynamic call graphs and are averaged to one single precision and recall value. They find that both pessimistic and optimistic analyses achieve almost the same precision, with the pessimistic performing slightly better. For 8 out of 10 programs they measure precision of at least 80%. The two remaining programs achieve lower precision, with a minimum of 65%. Both the analyses achieve a very high recall, of at least 80% for each of the input programs. These statistics imply that the call graphs are quite sound for the given input scripts, despite the mentioned limitations (design decisions) from the problem analysis (see section 3.2).

ACG states that the pessimistic call graph algorithm is well suited for use in an IDE, because it gives a small set of targets for most call sites. As pessimistic analysis comes with unresolved targets, they find it a better solution than listing many spurious call sites. Based on that statement, they measured the percentage of call sites with only one target for each pessimistic analysis of the input scripts. This is excluding native targets (where native targets are native JavaScript functions). It appeared that more than 70% of the call sites have at most one call target. Other than these findings, they implemented a couple of different tools (including smell detection and bug finding) that were based on the call graphs. They found code smells and bugs, indicating that the analyses are not only limited to jump to declaration functionality.

Chapter 5

Replication

This thesis is a partial replication of the ACG study. Implementations for their algorithms been made in the Rascal programming language, based on the information and formulas provided by their study. A JavaScript parser for Rascal has been developed in order to implement the call graph algorithms and their prerequisites. The implemented algorithms have been executed with the same input scripts (where possible) to measure the precision and recall of the call graph algorithms. The set consisted of 9 scripts, as one of the original scripts did not function properly. Furthermore some minor changes have been made to the flow graph algorithms, which are elaborated in chapter 6. One major change has been made as well, which is the disregarding of native functions (described in section 6.3.4).

5.1 Research Question

The research question is closely related to one of the evaluation questions of the ACG paper:

> How accurate are the computed call graphs, using the same precision and recall formulas, given the different environment?

What is meant with the different environment is the implementation in Rascal based on a preliminary JavaScript parser. Furthermore input scripts were found to be deviating from the ACG paper. The deviations came from different library versions, unversioned input scripts (thus possibly different) and having to replace some scripts to overcome parse errors. Other than that native functions have been disregarded in this study, whilst the ACG study did consider native functions.

5.2 Replication Process

First the Rascal JavaScript grammar has been implemented and tested. Afterwards the intraprocedural and interprocedural flow graph algorithm has been implemented and unit tested in Rascal. The output of the flow graph algorithms were also verified with those of another student, which replicated the same study. The pessimistic call graph algorithm was implemented subsequently. Validating this algorithm has been done by writing a unit test based on the example program that is stated in the ACG paper. Furthermore validation has also been done by comparing graphs to those of another student. The same was done subsequently for the optimistic algorithm. Several implementations have been made for the dynamic call graph algorithm. In the meanwhile a comparison module has been developed in Rascal, to compare the dynamic and static call graphs. This module include the given recall and precision formulas. Finally, call graphs were computed for the given input scripts and the input scripts were instrumented to create a dynamic call graph from. Statistics and data for the thesis were printed by using the comparison module.

Chapter 6

Flow Graphs

The call graph algorithms rely on flow graphs as an input. Flow graphs track intraprocedural and interprocedural flow and come about by applying a set of rules based on parse trees. Intraprocedural flow consists of the flow within procedures (thus functions), whereas interprocedural flow reasons about flow between procedures. The parse trees that are required for flow graph creation, are provided by the previously described JavaScript grammar. A parser for JavaScript has automatically been generated by Rascal, based on the prerequisite grammar (see section 8.1). The original algorithms rely on abstrax syntax trees (ASTs). Conversion of parse trees to abstract syntax trees within Rascal has not been implemented, because the parse trees contain the same relevant information as ASTs for the given algorithms. Further abstraction was thus not needed.
This chapter will explain what the required flow graph entails and how it is obtained using a parse tree as input. The flow graph serves as input for the call graph algorithms, which is after the processing that will be discussed in this chapter. An example of a textual flow graph has been included in appendix B, which could help in understanding the rules from this paragraph. Appendix A combines the formulas of intraprocedural and interprocedural flow for comprehension and overview reasons (it does however not yield different content).

6.1 Intraprocedural Flow

Intraprocedural analysis deals with the statements or basic blocks of a single procedure and with transfers of control between them [8, p. 3]. In other words an intraprocedural flow graph only contains flow from within single procedures. Intraprocedural flow analysis is the first step towards a flow graph after parsing an input script. Feldthaus et al. set up a set of rules to produce an intraprocedural flow graph [1, p. 5]. These rules indicate edges and vertices that have to be added to the flow graph, based on the nodes of a parse tree. Each rule is bound to a specific type of node in the parse tree (e.g. variable assignments or function declarations). All considered nodes have a unique location within the parse tree and represent a program element t (such as an expression, statement and function declaration). The set Π consists of all parse tree positions, where t^{π} represents such a *program element* t (such as an expression, statement and function declaration) at position π. Thus the following holds: $\forall \pi \rightarrow \pi \in \Pi$. In this thesis parse tree positions have been in the form of a 4-tuple, which consisted of the filename of the script, the line of the program element, the offset of the program element and the ending offset of an element. For a vertex ν, such tuple is represented as: ν(*filename.js@x:y-z*), where x is the line, y the starting offset and z the ending offset. Each vertex in the graph thus has a 4-tuple position that is an element of Π.
A set of vertex types that are represented in the flow graph has been defined. The basic set of intraprocedural vertices consists of four different types:

V	::=	**Exp**(π)	expression at position π
	|	**Var**(π)	variable declared at position π
	|	**Prop**(f)	property with name f
	|	**Fun**(π)	function declaration/expression at π

The vertices in set V presented up here, are all bound to a parse tree position (named π), so that they can be uniquely identified in a later phase.

Flow graph edges are added by matching program elements and applying a certain rule. Rules define which edges are added to the flow graph, based on the type of program element it matches to. The vertices in the flow graph come from the basic vertices set V. The following rules have been defined for intraprocedural flow:

rule #	**node at π**	**edges added when visiting π**
R1	$l = r$	$V(r) \rightarrow V(l)$, $V(r) \rightarrow \mathbf{Exp}(\pi)$
R2	$l \mid\mid r$	$V(l) \rightarrow \mathbf{Exp}(\pi)$, $V(r) \rightarrow \mathbf{Exp}(\pi)$
R3	$t \ ? \ l : r$	$V(l) \rightarrow \mathbf{Exp}(\pi)$, $V(r) \rightarrow \mathbf{Exp}(\pi)$
R4	$l \ \&\& \ r$	$V(r) \rightarrow \mathbf{Exp}(\pi)$
R5	$\{f\text{: } e\}$	$V(e_i) \rightarrow \mathbf{Prop}(f_i)$
R6	function expression	$\mathbf{Fun}(\pi) \rightarrow \mathbf{Exp}(\pi)$, if it has a name: $\mathbf{Fun}(\pi) \rightarrow \mathbf{Var}(\pi)$
R7	function declaration	if it is in function scope: $\mathbf{Fun}(\pi) \rightarrow \mathbf{Var}(\pi)$ if it is in global scope: $\mathbf{Fun}(\pi) \rightarrow \mathbf{Prop}(\pi)$

Figure 6.1: Rules for creating intraprocedural edges for the flow graph, based on parse tree nodes from set Π

The rules are defined by the data flow semantics of JavaScript. Each rule and the logic behind it will be explained separately. Function V that is mentioned in the table, will be described afterwards (in figure 6.1). The following list describes each rule in detail:

R1: Creates an edge for a property or variable to the expression it is set to. Another edge is introduced that connects the right hand side value to the assignment expression itself. This last edge is added because an assignment expression evaluates to the assigned value in JavaScript (`x = 3` evaluates to `3`).

R2: Creates an edge from both the considered truth values in a disjunction to the position of the disjunction expression itself. This is due to the characteristics of JavaScript, where values can be considered truthy or falsy [12, p. 75]. Non-boolean values are therefore coerced to be true or false. The result of the disjunction is therefore either the value of the left hand-side expression or the value of the right hand-side expression. Due to the truthy and falsy characteristics of JavaScript, these types may differ from booleans.

R3: Creates an edge from both result values in a ternary expression to the ternary expression itself. Depending on whether t evaluates to a truthy or falsy value, l or r will be the result. l for the truthy value or r for the falsy value. This rule is similar to that of a disjunction.

R4: Creates an edge for the right value of a conjunction expression to the conjunction expression itself. When both l and r evaluate to a truthy value, r will be the result of the conjunction expression. Functions are truthy values and a conjunction in JavaScript can only result in the l value when its falsy, therefore only the r value has to be tracked. This is due to design decision #3 (see section 3.2), which states that only function objects are tracked.

R5: This rule concerns object initializers in JavaScript. In the syntax `{ f: e }`, f is the name of a property and e represents its value. An object may contain multiple properties including functions. With an object initializer, each vertex of a property value will be connected to a vertex that represents its name. The value the object initialization evaluates to does not need to be tracked, because design decision #2 states that only function objects will be tracked.

R6: Introduces a function vertex that connects to the expression it is declared with. Because that is the position to which the function object flows to. If the function is named, it will also be connected to a variable vertex, so that later references to the function can be properly connected with edges.

R7: Introduces a function vertex that connects to a variable vertex in case it is not in global scope. In case the function is declared in global scope, it will be connected to a property vertex. This is because everything that is globally scoped, is not resolvable by a symbol (as the symbol table implementation only considers variables on local scope). The different approach for globally scoped functions is not documented in the ACG study. It is assumed that they had this intraprocedural edge in their specification as well. This edge is necessary in order to provide a more complete call graph (see subsection 10.3.2).

The V function (not to be confused with vertices V) as referred to in figure 6.1, represents a function that maps expressions to corresponding flow graph vertices. In this function, $\lambda(\pi, x)$ represents a lookup of variable x in the scope of position π (which is further explained in section 6.3):

$$V(t^{\pi}) = \begin{cases} \textbf{Var}(\pi') & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = \pi' \\ \textbf{Prop}(\pi) & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = undefined \\ \textbf{Prop}(f) & \text{if } t \equiv e.f \\ \textbf{Exp}(\pi) & \text{otherwise} \end{cases} \tag{6.1}$$

The mapping function V is dependent on a symbol table. This symbol table stores which variables are declared at which position in the local scope. The function V takes a program element as input, which has been described previously in section 6.1. In case program element t^{π} is a name label, the V function checks whether it can find the variable in the scope and returns a `Var` vertex with the position in the scope. In case it does not find the name in the scope, it will return a `Prop` vertex. The function will also return a `Prop` vertex when property names in the form of `someObject.propertyName` are served as input. The function V will return an `Exp` vertex with the position of the input program element if none of the previous rules apply. The way scoping of JavaScript works and is implemented (and thus impacts $\lambda(\pi, x)$) will be described later in this chapter (section 6.3).

6.2 Interprocedural Flow

Interprocedural analysis reasons about the procedures of an entire program and their calling relationships [8, p. 3] [5, p. 903]. Information flow is analyzed between callees[1] and targets, where callees are the call site of where a certain function (target) is executed. To reason about the interprocedural flow, the basic set of vertices V (previously defined in figure 6.1) has been extended:

V	::=	...	
	|	**Callee**(π)	callee of function call at position π
	|	**Arg**(π, i)	ith argument of function call at position π
	|	**Parm**(π, i)	ith parameter of function at position π
	|	**Ret**(π)	return value of function at position π
	|	**Res**(π)	result of function call at position π

The index i in vertices $Arg(\pi, i)$ and $Parm(\pi, i)$ is zero-based, where 0 will be reserved for the `this` keyword. Therefore any argument or parameter (other than the `this` keyword) will have an index $i \geq 1$.

[1] The ACG study refers to call site and callee interchangeably. This same approach has been adopted to this study. Callees in this thesis are therefore **not** the called target but the call sites.

In order to support interprocedural vertices, mapping function V also has to be extended (previously defined in figure 6.1). This extension is defined as follows (and should not be confused with vertices V):

$$V(t^{\pi}) = \begin{cases} \ldots \\ \textbf{Parm}(\phi(\pi'),\ 0) & \text{if } t \equiv this \\ \textbf{Parm}(\pi',\ i) & \text{if } \lambda(\pi, t) \textit{ is a parameter of function at position } \pi' \end{cases} \tag{6.2}$$

The above extension adds the mapping of the `this` keyword to a `Parm` vertex with index 0. Furthermore program elements that are parameters on the symbol table, will be resolved to `Parm` vertices, with their respective index value (starting at 1). This means that variable references to parameters will be represented as `Parm` vertices in the flow graph, as well as `this` references.

Finally rules for adding interprocedural edges are described. These are based on the just described extensions of function V and vertices V:

rule #	**node at π**	**edges added when visiting π**
R8	$f(\overline{e})$ or $newf(\overline{e})$ or $newf$	$V(f) \rightarrow \textbf{Callee}(\pi)$ $V(e_i) \rightarrow \textbf{Arg}(\pi,\ i)$, $\textbf{Res}(\pi) \rightarrow \textbf{Exp}(\pi)$
R9	$r.p(\overline{e})$	**R8** edges, $V(r) \rightarrow \textbf{Arg}(\pi,\ 0)$
R10	return e	$V(e) \rightarrow \textbf{Ret}(\phi(\pi))$

Figure 6.2: Rules for creating interprocedural edges for the flow graph, based on parse tree nodes from set Π

These interprocedural rules reason about calls and returns and extend the flow graph. Each of these rules will be explained shortly:

R8: Create an edge from the reference to function f to its calling expression. The reference of each argument will be connected to a vertex for the argument. The argument vertices are distinguishable by their index (based on the sequence of passed parameters). Therefore the first argument is considered to be index 1 and the second argument is considered to be index 2 etc. When no arguments are passed to a constructor function, parentheses can be omitted [12, p. 62].

R9: Create the same edges as **R8**. In addition there is the knowledge of which object contains property p. Therefore we can introduce a new edge from the reference to r (the containing object) to the zero argument (which represents the `this` reference). This is because the `this` keyword in the executed function will refer to its enclosing object.

R10: Create an edge from the reference of a return value to its enclosing function. The $\phi(\pi)$ notation denotes the innermost enclosing function of π (excluding π itself).

With the defined extensions, connecting `Parm` to `Arg` vertices and `Ret` to `Res` vertices, allows tracking interprocedural flow. Both the optimistic and the pessimistic call graph construction algorithm have a different approach of connecting these vertices. This will be discussed in chapter 7. To aid in understanding the rules above, an example of a textual representation of a flow graph has been included in appendix B. For the sake of understandability all flow graph rules have been combined and added to appendix A.

6.3 Scoping

Scoping is important for the mapping function V (formula 6.1 and 6.2) which maps program elements to vertices. It utilizes function $\lambda(\pi, x)$ to find local variable x for position π. An accurate variable lookup implies a more accurate flow graph and therefore also a more accurate call graph.
Scope in a programming language controls the visibility and lifetimes of variables and parameters [14, p. 36].

6.3.1 Scoping Types

JavaScript has two types of scoping. It has a *global* scope, which means that a global variable is defined everywhere in your JavaScript code. The second type of scoping in JavaScript is limited to the scope of a function. Variables declared within a function are only defined within its body. These are called *local* variables and include the function its own parameters as well [12, p.53]. Therefore this scope is known as *local* scope. JavaScript does not support block scoping, despite its block syntax support [14, p. 36].
Flow graph creation takes both types of scope into account. Global variables are not stored in a symbol table, as the lookup function for the flow graph algorithm solely looks up local variables. Global variables will therefore always be resolved to `Prop` vertices by mapping function V.
The symbol table maps variable names to the origin of its declaration; node position π in Π. The symbol table for a function f with position π consists of the union of the symbol table of $\phi(f)$, (f, π) and all its parameter names mapped to node position π. In other words the symbol table of function f consists of the symbol table of its enclosing function, the function name f itself and its parameters. The variable that refers to the function name and the parameter variables all refer to the position of function f. Variables that come from the enclosing scope will retain their original positions, unless they are overridden (see section 6.3.3). The following example gives insight in scoping of JavaScript:

```
1 function f(a, b) {
2   function g(b, c) {
3     ...
4   }
5 }
```

In the upper example, the symbol table of function g consists of variable a, b, c and d. The function names f and g are also included in the symbol table for g. In function g, b refers to the parameter of g, whereas in function f, b refers to the parameter of f. This is due to overriding. The phenomenon that b refers to the parameter of function g in the scope of g, is often called variable shadowing. This is due to the invisibility of parameter b of function f (it is shadowed by the inner variable).

6.3.2 Hoisting

An important feature of the JavaScript scope is called *Hoisting*. This feature implies that variable declarations are hoisted to the top of the corresponding scope [12, p. 54]. Therefore declaring a variable anywhere in a scope, is equal to declaring it at the top of the scope [20]. This also applies for function declarations as they are essentially declaring a variable name. Variables that are not yet initialized but hoisted will have the `undefined` value until they are initialized. Hoisting has been taken into account in order to refer to the correct position with mapping function V in the flow graphs. It is implemented by initializing the symbol table with the hoisted variables within the scope, before applying all flow graph rules.

A sample code is included to sketch how hoisting works. The reference to variable `x` on line #3 will refer to the declaration on line #4 rather than line #1:

```
1  var x = 3;           // not referenced
2  function f() {
3      console.log(x); // prints undefined, but x refers to the next line's x
4      var x = 20;
5      console.log(x); // prints 20 and refers to the previous line
6  }
```

6.3.3 Overriding

Global variables are defined throughout the program, whereas local variables are defined throughout its enclosing function and nested functions [12, p. 55]. Variables on the scope can be redeclared within nested functions in the scope, whereas redeclaration of a variable within the same scope is ignored in JavaScript. Such redeclaration attempt will simply result in a new assignment, rather than a different declaration origin of the variable. The following example provides some insight in such:

```
1  function f() {
2    var x = 1; // declaration
3    var x = 2; // reassignment
4  }
```

In the above example, line #2 is the declaration for x, whereas line #3 simply reassigns it.
Function names and parameters are also variables and can be overridden like any other variable. In addition to this rule, a function name on the scope can be overridden within its own parameters. The following example of function f clarifies this:

```
1  (function f(f) {
2      return f;
3  })(10);
```

Due to the overriding of function name f within its scope, the example code will evaluate to 10, rather than return a function object.
Overriding of variables and their restrictions in terms of redeclaration have been adopted to the flow graph creation algorithm. Mapping function V benefits from this as variable references will be more accurate.

6.3.4 Native Function Recognition

ACG defines a simple extension in their flow graph design. By extending vertices set V with a `Builtin` vertex, we can simply introduce a basic model of native functions. Creating an edge from `Prop(nativeFunctionName)` to `Builtin(nativeFunctionName)` allows us to resolve function calls to its native target. Schäfer released an open source implementation of the call graph algorithms, including this native function model.
Native functions have not been considered in the study. They have been added to the call graph algorithm for the sake of completeness and transferability for future work.

There are several reasons why the native function model has been disregarded in this study:

1. The authors of ACG are aiming to provide jump to declaration functionality in IDE's, where native functions would be out of reach as they are implemented natively.
2. The native function model seems incomplete. It has been observed that functions like `eval`, `HTMLDocument` prototype functions, `HTMLAudioElement` prototype functions, `Object.prototype.__defineGetter__` and `Object.prototype.__defineSetter__` are missing. The lack of these functions were observed by globally browsing through the native model. Therefore there is a possibility that more functions are missing.
3. It is unclear if the native model is based on the ECMAScript specification and if so on which version.
4. Even with a complete list of native functions, recording natives in a dynamic call graph would be error-prone due to finding the proper prototype (or object) and name for a given callee during runtime.

With point #4 remaining, some time has been invested to search for more complete native function models online. This however did not result in finding any model.

6.4 Multiple Input Scripts

ACG does not mention how multiple script files are supported. In this study flow graphs are created for each input script. The union of all flow graphs are used as an input for the call graph algorithms. The set of parse trees has to be kept as well, in order to have the call graph algorithms retrieve a list of arguments for calls during its analysis.

Chapter 7

Static Call Graph Analysis

This chapter will explain the design of both the pessimistic and optimistic call graph algorithms in depth. Both algorithms rely on a flow graph that is based on parsed input scripts. A flow graph is a graph that tracks flow within and beyond procedures of a program (see chapter 6). The core difference between the pessimistic and optimistic call graph computation has been discussed in section 3.1.
Whilst in principle unsound by its characteristics, the evaluation of ACG indicates that in practice few call targets are missed due to unsoundness [1, p. 2].
In this replication, the implementations of the flow graph (chapter 6) and static call graph algorithms have been done in Rascal. These implementations are both reliant on the prewritten JavaScript grammar, which is discussed in section 8.1. Details regarding the implementation in Rascal have been discussed in section 8.2.
Before reading this chapter it might be interesting to check appendix B, which shows an example of a pessimistic and optimistic call graph for the same JavaScript input program. This is an output of the implemented algorithms and thus indicates on what this paragraph works towards.

7.1 Pessimistic Static Call Graph Analysis

This paragraph explains how a pessimistic call graph can be derived with the flow graph from the previous paragraphs as an input. Call graph constructions typically rely on on three different quantities that are circular dependent (see section 3.2). In order to break the circular chain, the pessimistic version only considers *one-shot closures* in interprocedural analysis. *One-shot closures* are considered to be functions that are directly applied by zero or more arguments. In practice *one-shot closures* are also called immediately-invoked function expressions (IIFE) [18, p. 115]. The functions can be named as well as be anonymous. The following code shows how a *one-shot closure/IIFE* is constructed:

```
1 (function f(parameters) {
2   ...
3 })(arguments);
```

It is unclear whether the ACG research also considered named (non-anonymous) functions in a closure. In this research no distinction has been made, because it still allows us to reason about its actual callee. Interprocedural flow other than *one-shot closures* are modelled through `Unknown` vertices. For this purpose the `Unknown` (without parameters) vertex is added to the standard vertices set V. This vertex represents interprocedural flow that can not be tracked.

The pessimistic call graph algorithm is defined in pseudo-code:

Algorithm 1 Pessimistic Call Graph Construction

Input: Parse tree of code for which the call graph is
Output: Call graph C, escaping functions E, unresolved call sites U

1: $C := \{\ (\pi,\pi') \mid t^{\pi}$ is a one-shot call to a function $f^{\pi'}\ \}$
2: $E := \{\ \pi' \mid \neg\exists\pi\ .\ (\pi,\pi') \in C\ \}$
3: $U := \{\ \pi \mid \neg\exists\pi'\ .\ (\pi,\pi') \in C\ \}$
4: $G := \emptyset$
5: Add interprocedural edges(G, C, E, U) (algorithm #2)
6: Add flow graph edges to G (by applying rules R1 to R10)
7: $C := \{\ (\pi,\pi') \mid \mathbf{Fun}(\pi') \overset{opt}{\rightsquigarrow}_G \mathbf{Callee}(\pi)\ \}$
8: $E := \{\ \pi \mid \mathbf{Fun}(\pi') \rightsquigarrow_G \mathbf{Unknown}\ \}$
9: $U := \{\ \pi \mid \mathbf{Unknown} \rightsquigarrow_G \mathbf{Callee}(\pi)\ \}$

The algorithm starts with with a set of tuples (set C) that are one-shot calls from a call site to a local call target. E is the set that represents all functions that are not in in the tuples of set C. That means that each of the functions in E does not have a local call target and is thus not part of one of the tuples in set C. The set U represents the set of unresolved call sites, which are the call sites that are not resolved to a local call target (and thus not part of one of the tuples in set C). The set G represents the flow graph, which starts empty.
Interprocedural edges will have to be added (line #5), after the initialization of these four sets. This is done with an algorithm that connects **Parm** vertices (parameters) to **Arg** (arguments) vertices and **Ret** vertices (return values) to **Res** vertices (results). The algorithm is defined in the section 7.1.2 and allows for tracking interprocedural flow, as it will be clear which arguments are applied to which parameters of a local call target. Furthermore it will determine where the function results will flow to.
Afterwards the edges from the flow graph rules (chapter 6) are added to the flow graph G. To extract a final call graph, a transitive closure has to be made of flow graph G so that reachability of flows will be adapted. This transitive closure does not consider paths through **Unknown**, which is why its referred to as an optimistic transitive closure. This optimistic transitive closure only reasons about known interprocedural flow (thus only *one-shot closures*). A transitive closure answers a reachability question of a graph. Reachability queries test whether there is a path from a node ν to another node υ in a large directed graph [21, p. 21]. An example of how this transitive closure works on a basic level, is demonstrated in section 7.1.1.
Edges from **Fun** vertices (function) to **Callee** vertices (the callee of the called function) will be selected afterwards. This will extract the actual call graph from the flow graph. The sets E and U are filled with edges that have an **Unknown** vertex on one side. This means that E is the subset of a transitively closed flow graph G, where a function is connected to an unknown call site. The set U is the opposite of E, which is the subset of a transitively closed flow graph G, where an **Unknown** function is called from a known **Callee** vertex. Both sets E and U are not necessary for the call graph, but can be used to indicate missing information in the call graph. Note that the transitive closure used to fill set E and set U does consider paths through **Unknown** vertices. It is therefore a (normal) transitive closure.

7.1.1 Transitive Closure

The following piece of code gives an idea on how a transitive closure results in call relationships (note that irrelevant edges for the example are left out):

```
1 function divide(x, y) {
2   return x / y;
3 }
4 divide(4, 2);
```

With intraprocedural rule **R7** the graph will have the following edge added: **Fun**(...) → **Prop**(*divide*). The interprocedural rule **R8**, will introduce an edge as follows: **Prop**(*divide*) → **Callee**(...). Note that the dots are simply substitutes for positioning information, which are 4-tuples as described earlier in section 6.1.
The transitive closure in this case will connect the divide function with its call site. The following edge will emerge: **Fun**(...) → **Callee**(...). After the transitive closure, the **Prop** node will not be an intermediate node between the function and its callee anymore. Visually this will look as follows, with the dashed line representing the edge that emerged by the transitive closure:

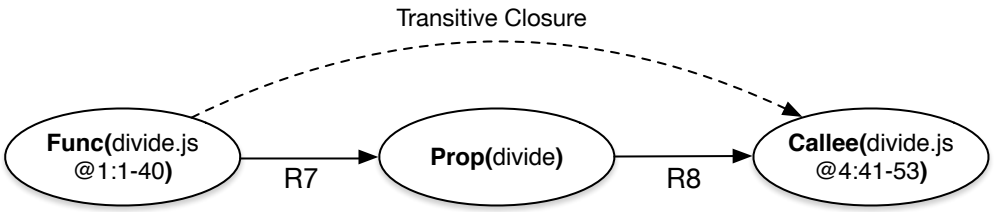

Figure 7.1: An example of a transitive closure on a flow graph

The difference between a transitive closure and an optimistic transitive closure, is that the optimistic transitive closure does not consider reach through **Unknown** vertices. In other words, a direct edge will not be created with an optimistic transitive closure, if the intermediate vertex of two vertices is of the **Unknown** type. With a (normal) transitive closure this distinction would not be made. A transitive closure on flow graph G is indicated in the form of $\rightsquigarrow_G$, whereas an optimistic transitive closure is signified as follows: $\overset{opt}{\rightsquigarrow}_G$. The transitive closure is necessary to model missing information, whilst the optimistic transitive closure should not reason through unknown paths.

7.1.2 Interprocedural Edges

Adding interprocedural edges as referred to on line #5 by the pessimistic algorithm, is defined as follows:

Algorithm 2 Add Interprocedural Edges

Input: Flow graph G, initial call graph C, escaping functions E, unresolved call sites U
Output: Flow graph G with added interprocedural edges

1: **for all** $(\pi, \pi') \in C$ **do**
2: add edges **Arg**(π, i) → **Parm**(π', i) to G
3: add edge **Ret**(π') → **Res**(π) to G
4: **for all** $\pi \in U$ **do**
5: add edges **Arg**(π, i) → **Unknown** to G
6: add edge **Unknown** → **Res**(π) to G
7: **for all** $\pi' \in E$ **do**
8: add edges **Unknown** → **Parm**(π', i) to G
9: add edge **Ret**(π') → **Unknown** to G

As defined earlier, the set of escaping functions E represents all functions that are not enclosed in a *one-shot call*. The set of unresolved call sites U represents all function calls that are not a *one-shot call*. Set C represents the call graph of *one-shot calls* and G represents the flow graph.
The algorithm starts with iterating over the tuples (from callee to function) in the call graph C. In this loop each of the arguments in the function call will be connected to the parameters of the local

function. Furthermore a return vertex (of the function) will be connected to the result (which is the position of invocation).
Afterwards the algorithm will iterate over the unresolved call sites U, which are function calls that are not connected to functions. For each of these call sites, their arguments are connected to the **Unknown** vertex, which signifies that it is unknown to where those arguments flow. Furthermore the result of the invocation is connected to the **Unknown** vertex as well, as it is unclear where the information for the result flows from.
For each of the escaped functions (functions that have no known call site), its parameters will be connected to the **Unknown** vertex. It is unclear which data flows to the parameters. Finally for each of the escaped functions, the return of the function will be connected to the **Unknown** vertex, as it is unclear where the return data flows to as well.

7.2 Optimistic Static Call Graph Analysis

The optimistic call graph algorithm is iterative up to a fix-point and relies on an assumption that is likely to be unsound (see section 2.2.1). Due to the iterative approach, it will reason until it can not find alternative paths. Therefore paths that are unlikely, might be adopted to the call graph. It is very similar to the pessimistic algorithm and also relies on the same flow graphs as input. In essence it varies in two ways: it is iterative rather than one pass and it starts off with an empty triple (rather than solely reasoning about *one-shot calls* when it comes to interprocedural flow). This triple yields the initial call graph, the set of escaping functions and the set of unresolved call sites. The optimistic call graph algorithm is also defined in pseudo-code:

Algorithm 3 Optimistic Call Graph Construction

Input: Parse tree of code for which the call graph is
Output: Call graph C

1: $\langle C, E, U\rangle := \langle\emptyset, \emptyset, \emptyset\rangle$
2: $G := \emptyset$
3: Add flow graph edges to G (by applying rules R1 to R10)
4: **do**
5: $\quad LastC := C$
6: $\quad LastG := G$
7: $\quad$ Add interprocedural edges(G, C, E, U) (*algorithm* #2)
8: $\quad C := \{\ (\pi,\pi') \mid \mathbf{Fun}(\pi') \overset{opt}{\rightsquigarrow}_G \mathbf{Callee}(\pi)\ \}$
9: $\quad fixpointReached := (C \equiv LastC \wedge G \equiv LastG)$
10: **while** $\neg fixpointReached$

This algorithm does not start with *one-shot closures*, but with empty sets for C, E, U and G. The flow graph is then extended with the intra- and interprocedural edges from rules **R1** to **R10**. After that the iterative process starts, where the interprocedural edges algorithm from section 7.1.2 is applied. This is followed up by the optimistic transitive closure and the extraction of the call relationships from **Fun** to **Callee**. Afterwards it is checked whether the call and flow graph have grown since the last iteration. If so, another iteration will follow up. This is done repetitively until flow graph G and call graph C are not extended anymore by the interprocedural edges algorithm and the optimistic transitive closure. The algorithm will thus be terminated when no other flows can be discovered. With this knowledge it is easy to observe that the optimistic algorithm demands heavier computation, due to the possibility of multiple executions of the interprocedural edges algorithm and its optimistic transitive closure. As described in section 3.2, callbacks are tracked with optimistic call graph algorithm, whilst the pessimistic call graph algorithm only tracks interprocedural flow with one-shot closures. The optimistic call graph algorithm may add edges from argument to parameter vertices and return to result vertices for function objects, whilst the pessimistic call graph algorithm would not do so, due to it solely considering one-shot closures. The pessimistic call graph algorithm

would only connect these same interprocedural vertices in case of one-shot closures. The jQuery subset example in appendix B demonstrates this, as the optimistic call graph analysis results in one more edge due to a callback.

Chapter 8

Implementation

This chapter starts with explaining the preliminary JavaScript grammar, which was written prior to implementing the algorithms. Afterwards the implementation to obtain flow and call graphs in Rascal will be discussed. Finally, implementation for the dynamic call graph is described in depth, since implementing dynamic call graph attainment has not been straightforward.

8.1 JavaScript Grammar

8.1.1 Introduction

The flow graph algorithms rely on parsed JavaScript as input, in the form of an abstract syntax tree. Rascal is a metaprogramming language that is being developed at the CWI and is used to carry out research in software engineering. Prior to this thesis, Rascal did not yet support JavaScript parsing. It has been chosen to implement a JavaScript grammar for Rascal, regardless of the presence of existing tooling. This is because it would improve the Rascal standard library for future usage, but also for the sake of learning by doing.
In this section the implementation and design of the grammar will be discussed. The parsing mechanism behind the grammar is beyond scope, because Rascal provides an automatic parser generator based on context-free grammars as an input [22].

8.1.2 Implementation

A basic preliminary version of the JavaScript grammar (by Tijs van der Storm) for Rascal has been taken as a starting point. The implementation of the grammar was led by the ECMAScript 5.1 (ECMA-262) specification of June 2011 [23]. In this section, some of the common problems in JavaScript will be discussed. Other than that, resolutions for experienced problems will be mentioned and the design will be elaborated.

8.1.2.1 Automatic Semicolon Insertion

JavaScript does not require semicolons after each written statement. Semicolons are one of the ways to mark separate statements. Line terminating characters are treated as a semicolon, in case the next non space character after the line break can not be interpreted as a continuation of the current statement [12, p. 25]. This is called *automatic semicolon insertion*. The following characters are considered line terminating characters: *carriage return*, *line feed*, *line separator*, *paragraph separator* [23, p. 15]. The following example code results in declaring variable x with 7, due to automatic semicolon insertion (the semicolon would be inserted at the end of line #3):

```
1 | var x = 1+
2 | 2
3 | *3
```

A closing curly brace (thus `}`) could also serve as terminating character for statements, provided that the code is correct. Thus the brace should be closing a block.
Automatic semicolon insertion applies differently for `return`, `continue`, `break` and `throw` statements [12, p. 26] [23, p. 26]. A line terminating character after any of these keywords will be assumed to be the end of the statement, regardless of their succeeding syntax. In the following example, the function invocation on line #3 in the following code can be considered dead code:

```
1 function f() {
2     return
3     g(10);
4 }
```

Furthermore postfix operators require to be on the same line as its operand [23, p. 27]. Therefore ++ and −− operators on a separate line, are not considered postfix operators for its previous connected operand, but prefix operators for its connected operand ahead. The following example will clarify how this works:

```
1 var i = 0, j = 10;
2 i
3 ++
4 j
```

In the example above, j will be increased, rather than i. In the grammar this is done by simply not allowing a new line between an expression and the postfix operators. Therefore the increment operator is considered a prefix operator of j rather than i.
The grammar that has been developed as a prerequisite of this research, takes automatic semicolon insertion into account for closing braces and linefeeds. It also recognizes the four aforementioned exceptional statements (`return`, `continue`, `break` and `throw`). Moreover, due to automatic semicolon insertion, some contextual knowledge has to be present. Ambiguities did occur when code similar to the following was parsed:

```
1 var x = 1
2 +1;
```

This was due to the parser considering line #2 as a separate statement as well as part of the variable declaration on line #1. The complete shorter production would be filtered (deleted from the parse tree) as soon as a larger production was available. This has been done by visiting the ambiguous node in Rascal in combination with a `filter;` statement.

8.1.2.2 Structure

This paragraph gives an impression of the structure of the grammar. This can be used as a reference when trying to interpret the algorithm implementations or when one wants to work with the JavaScript grammar. Important decisions of the grammar will be elaborated, as well as the overall layout of the grammar. Note that irrelevant layers and nodes are left out to maintain a clear overview.

The following grammar rules represent the top of the grammar hierarchy; each valid input script starts with and consists of at least one function declaration or statement.

⟨*Source*⟩ ::= ⟨*SourceElement*⟩ | ⟨*Source*⟩ ⟨*SourceElement*⟩
⟨*SourceElement*⟩ ::= ⟨*FunctionDeclaration*⟩ | ⟨*Statement*⟩

The following grammar shows how a function declaration is structured. It consists of at least a name, zero or more parameters (that are comma separated) and an implementation block (also known as function body). Note that the `Id` syntax is simply the specification of the `Identifier` syntax from the ECMAScript specification:

⟨*FunctionDeclaration*⟩ ::= 'function' ⟨*Id*⟩ '(' ⟨*IdSequence*⟩ ')' ⟨*Block*⟩
⟨*IdSequence*⟩ ::= ⟨*Id*⟩ | ⟨*IdSequence*⟩ ',' ⟨*Id*⟩

JavaScript statements are sentences/commands that are terminated by a semicolon [12, p. 87]. Statements can also be control structures, that alter the order of execution, like conditionals, loops and jumps [12, p. 87]. Statements can contain several different syntactic constructions, including expressions (e.g. the condition of an if-statement), blocks (e.g. try-catch blocks), identifiers (e.g. the label of a break statement). They can be recursive as well; e.g. if-statements may execute a statement after a condition has been met.

JavaScript expressions are phrases that can be evaluated to a value [12, p. 87]. Therefore arithmetics are considered to be expressions. Expressions may simply contain a left-hand side value and a right-hand side value; for example a division expression. Function expressions contain blocks and may contain a name (thus `Id`), in case they are not anonymous. If the function expression has parameters it will contain more `Id`s. Other than that, an expression can be recursive, until it matches a primary or non-recursive expression. Primary expressions are standalone expressions that do not contain any simpler expressions. They can be constants, literal values (number literals, hex literals, string literals and regular expression literals), some reserved keywords (e.g. `null` or `true`) and variable references [12, p. 87].

An important structure to describe is the structure of blocks. These are discussed separately for a specific reason; due to the specifics of statements termination, the parser needs to know if it is in a block and where. The closing brace can terminate a statement. This only counts for the last statement in a block. Other than that the alternative statement termination constructions are still usable, as well as nested function declarations. Due to the addition of this extra terminating statement, blocks have been implemented separately. In the implementation of the grammar, blocks are a sequence of statements with a potential last block statement that gets terminated with the closing brace. The closing brace also terminates the block itself. Note that the last statement in a block may also be terminated by other terminators than the closing brace.

8.1.2.3 Validation and Testing

When developing a grammar, correctness has to be preserved when adding or changing grammar rules. Tracking the issue of cascading erroneous changes has been done through unit testing. A pretty printer is developed for the types in the grammar, which helped with type and content assertion. For each type it would print a specific string. By creating snippets manually and asserting the expected string when parsing the snippets, one could determine whether something broke in the grammar. The following Rascal example will clarify this:

```
1 | outcomeIsCorrect("return 1; 1;", "|Return [1];|Expression [1];|")
```

The `outcomeIsCorrect` function parses the first parameter and pretty-prints it. Afterwards it asserts if its output is the same as the second parameter.
Other than testing parsing correctness, a lookup for ambiguous nodes in the parse tree was done by unit tests. Whenever a change broke the code or created ambiguous nodes, a unit test for it would be introduced. This form of testing has been carried through the whole grammar writing process.
Another form of validation was simply done by parsing the subject input scripts for the call graph construction algorithms. Edges that intersected with a dynamic call graph (which came about with a different parser), do mean that at least a part of the input script is interpreted correctly. This comes with the assumption that the used parser for the dynamic call graph, does not have the same exact faults. The output data of the research does give some guarantee of the validity of the grammar (see chapter 9).

8.1.3 Evaluation

The JavaScript grammar has been an investment of two months of work prior to this research project. It has been said that the CWI will integrate the JavaScript grammar into the Rascal standard library. It still has some remaining ambiguity and parsing issues. The included appendix E describes the problems that should be resolved, as well as work that can be done for the Rascal JavaScript grammar. Furthermore a list of erroneous snippets is stored with the project, so that they can be resolved in a later stage.

8.2 Algorithm Implementations in Rascal

In this section some of the implementation of the flow and call graph algorithms in Rascal will be discussed.

8.2.1 Flow Graphs

The flow graphs have been implemented based on the previously described algorithm from chapter 6. The Visitor pattern is useful in this case, as it can help in gathering information without polluting the parse tree. The Gang of Four advice to use the Visitor pattern for the following related case: "use Visitor when many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid polluting their classes with these operations. Visitor lets you keep related operations together by defining them in one class" [24, p. 321].
Parsed JavaScript has served as input for the so called `FlowGraph` module. This module visits each program element (as mentioned in chapter 6) and creates a graph based on the rules **R1** to **R10**. The Visitor traversal is a language construct in Rascal and therefore does not need to be implemented. A `Vertex` type has been introduced that entails the basic set of vertices V from the flow graph algorithm. The flow graph is a relation type, which are sets of tuples of one type (the `Vertex` type in this case). In Rascal this type can be represented as follows: `rel[Vertex, Vertex]`. The `Vertex` type that is introduced is defined as follows (which is similar to Haskell):

```
1 | data Vertex = Exp(Position p) | Var(str name, Position p) |
2 |     Prop(str name) | Fun(Position p) | Callee(Position p) |
3 |     Arg(Position p, int i) | Parm(Position p, int i) |
4 |     Ret(Position p) | Res(Position p)  | Builtin(str canonicalName) |
5 |     Unknown() | Empty();
```

The `Positions` are 4-tuples, which consist of the filename of the script, the line of the program element, the offset of the program element and the ending offset of an element. Such tuple would be represented as follows for a vertex ν: ν(*filename.js@x:y-z*), where x is the line, y the starting offset and z the ending offset. The `str` keyword abbreviates string and `int` abbreviates integer.
The output of the JavaScript parser when parsing a script is of the `Source` type. This is the top element of the grammar, which has been described in section 8.1. Visiting this `Source` type (which is a subtype of `Tree`) and adhering to the graph rules, allows us to create a flow graph. A simplified function to create a flow graph of functions within scope, is defined as follows:

```
1 | rel[Vertex, Vertex] createFlowGraph(Source source) {
2 |   rel[Vertex, Vertex] flowGraph = {};
3 |   top-down visit(source) {
4 |     case FunctionDeclaration f:{
5 |       Position p = getPosition(f);
6 |       flowGraph += <Fun(p), Var("<f.name>", p)>;
7 |     }
8 |   }
9 | }
```

The `visit` statement in Rascal can have a prefix to define the sequence of its tree traversal. In this example, the tree is traversed from top to bottom. The visitor used over here, is accumulative, meaning that it traverses the tree to collect information [25]. Flow graph rule **R8** is applied in the example by matching nodes in the tree with the `FunctionDeclaration` type. This `FunctionDeclaration` is a type defined in the grammar. This type contains a label called `name`, which is the name of the function and thus the name used in the `Var` vertex in the flow graph. The real implementation looks similar, but implements all the rules and includes scoping through a symbol table. In reality the function is therefore recursive, due to propagating scope to depth in the tree. This is done by appending the symbol table and passing it through as a parameter. Hoisting is also implemented through `visit`, where the symbol table gets filled prior to appending the flow graph. The symbol table is defined as a map and is stored as follows:

```
1 | data SymbolMapEntry =
2 |     parameterEntry(Position position, bool overridable, int parameter) |
3 |     entry(Position position, bool overridable);
4 | data Scope = global() | scoped(map[str, SymbolMapEntry] symbolMap);
```

The demonstration shows that there are two forms of scope, global and local. The global scope does not have to be stored, as everything there resolves to the `Prop` type (see chapter 6). Furthermore an entry can be a parameter entry or a regular entry, so that parameters on the scope can be discriminated from any other type of variable declaration. This is necessary for the mapping function V, which maps variables to vertices (as described on page 20). In the implementation, a symbol table (if present) is simply a map that contains a key (a variable name) and a value (variable declaration or parameter). The values in the map contain a boolean to define whether it is overridable or not. This is because in a new scope, a variable name might be redeclared, whilst in its original scope that is not allowed (see section 6.3). In addition, parameter entries contain an additional integer to store the index of the parameter.
An example of an output flow graph using the implementation has been included in appendix B.

8.2.1.1 Native Functions

The native functions are imported from the aforementioned native function model by Schäfer. This model is a plain text file, where each line maps a property name to its native function. In order to make this model take part of a flow graph, lines are read and mapped to tuples of the `Vertex` type. These tuples will have the following form: `<Builtin(propertyName), Prop(functionName)>`. The module that is created to do these imports, is called `FlowGraphNativeImporter`.

8.2.1.2 Utilities

A pretty printer (`FlowGraphPrettyPrinter`) has been developed in order to output a flow graph (and thus also a call graph) in a textual representation. This also facilitates storing flow graphs and unit testing, as the expected results for an input program can be stored in a textual format and thus used for comparison.

8.2.2 Call Graph Computation

This section will demonstrate some of the code to create a call graph. Unfortunately the call graph algorithm sources themselves do not fit here and are therefore described shortly.

8.2.2.1 Pessimistic Call Graph Computation

The pessimistic call graph algorithm starts of with an initial call graph in the form of one-shot calls. The Visitor pattern is used to retrieve those, which works similarly to the accumulating flow graph example (section 8.2.1). The same goes for the sets of escaping functions and unresolved call sites. A data type is introduced to store relevant information for each of these sets. For one-shot closures this is the position of the call, the arguments and the position of the expression that is being called.

For unresolved call sites and escaping functions, these are simply their arguments and parameters respectively. After having these initial sets, the interprocedural edges will be added (described in depth in section 8.2.2.3). Then the just described `createFlowGraph` call (section 8.2.1) is used to add the edges from rules **R1** to **R10**. Finally the call graph can be extracted by the optimistic transitive closure. This extraction is written with the following comprehension:

```
rel[Vertex, Vertex] callGraph =
     { <y, x> | <x,y> <- optimisticTransitiveClosure(flowGraph),
       (Fun(Position _) := x || Builtin(str _) := x), Callee(Position _) := y };
```

The comprehension collects tuples from the optimistic transitive closure that are edges from functions and builtins (natives) to call sites. The tuples are reversed so that the extracted call graph consists of edges from call site to either functions or natives. The implementation of the optimistic transitive closure is presented in subsection 8.2.2.4.

8.2.2.2 Optimistic Call Graph Computation

The optimistic call graph algorithm, does not deviate much from the pessimistic one. It starts with empty sets for the initial call graph, escaping functions and unresolved call sites. Afterwards the flow graph is filled with the `createFlowGraph` call for rules **R1** to **R10** (see section 8.2.1). Then the looping starts until a fix-point is reached. The loop entails the interprocedural edges algorithm and the call graph extraction. Those are executed through the same procedure as the pessimistic call graph. This extracted call graph will then later function as an input for the interprocedural edges algorithm in the next iteration. This looping process is repeated until a fixpoint is reached. The fixpoint is reached when the flow graph and call graph did not grow since the last iteration.

8.2.2.3 Adding Interprocedural Edges

The algorithm for adding procedural edges is applied after initially filling sets of one-shot calls, unresolved call sites and escaping functions (see page #26). This algorithm takes these three sets as an input and looks as follows:

```
rel[Vertex, Vertex] flowGraph = {};
for (OneShotCall call <- initialCallGraph) {
  int i = 1;
  for (Expression arg <- call.args) {
    flowGraph += <Arg(call.oneShotCall, i), Parm(call.expressionToCall, i)>;
    i += 1;
  }
  flowGraph += <Ret(call.expressionToCall), Res(call.oneShotCall)>;
}
for (UnresolvedCallSite unresolvedSite <- unresolvedCallSites) {
  int i = 1;
  while (i <= unresolvedSite.args) {
    flowGraph += <Arg(unresolvedSite.position, i), Unknown()>;
      i += 1;
  }
  flowGraph += <Unknown(), Res(unresolvedSite.position)>;
}
for (EscapingFunction escapingFunction <- escapingFunctions) {
  int i = 1;
  for (Id param <- escapingFunction.args) {
    flowGraph += <Unknown(), Parm(escapingFunction.position, i)>;
    i += 1;
  }
  flowGraph += <Ret(escapingFunction.position), Unknown()>;
}
```

Note that the `Expression` and `Id` types in the sample code are data types that come from the grammar.

8.2.2.4 Optimistic Transitive Closure

In Rascal a transitive closure can be represented similar to its mathematical notation. The transitive closure of R (R^+ in mathematics) is therefore written as `R+`. The optimistic transitive closure does not consider paths through `Unknown` vertices (see section 7.1). Therefore the set to optimistically transitively close, is the set without the tuples that have `Unknown` vertices as second element. Because the transitive closure in Rascal is optimized, it is better to subtract the set of tuples that can lead to paths through `Unknown`, rather than create an own transitive closure implementation. The code to compute an optimistic transitive closure in Rascal looks like the following:

```
1 rel[Vertex, Vertex] optimisticTransitiveClosure(rel[Vertex, Vertex] R) {
2   rel[Vertex, Vertex] unknownRelation = { <x, y> | <x, y> <- R, isUnknown(y) };
3   rel[Vertex, Vertex] relationWithoutUnknowns = R - unknownRelation;
4   return ((relationWithoutUnknowns+) + unknownRelation);
5 }
6 bool isUnknown(Vertex vertex) = vertex := Unknown();
```

Note that the binary `:=` operator is used for pattern matching in Rascal.

8.3 Dynamic Call Graphs

As mentioned before, computing an exact static call graph is an undecidable problem. In order to measure how complete the static call graphs were, they had to be compared to a dynamic call graph. In essence a dynamic call graph logs the position of function calls and their invoked functions during runtime. In other words; the dynamic call graph is a snapshot of the call relationships that were available at runtime. This means that it is likely that multiple dynamic call graphs for a given input program may differ, because different paths and states might have been traversed. Multiple dynamic call graphs of a given program are therefore likely to entail different call relationships.
The dynamic call graphs are intended to be a complete comparison model for the statically computed call graphs. They can be used to analyze the correctness and completeness of the static call graphs. In order to have a dynamic call graph as complete as possible, all reachable calls and paths have to be traversed. This is done by manually attempting to get the subject script/program to be in different states using different inputs and combinations of those. Recording all calls however is far from trivial, because there are various syntactic constructions in JavaScript that can invoke functions. The table below lists various invocation constructions:

#	type	code example
1	Regular invocation	`f()`
2	Dynamic invocation	`eval("f()")`
		`new Function("return f()")()`
		...
3	Indirect native invocation	`f.apply()`
		`f.call()`
4	Setter invocation[1]	`object.weight = 2`
5	Getter invocation[2]	`object.weight`

Table 8.1: Ways to invoke functions in JavaScript

Apart from the most rudimentary case (case #1), it is possible in JavaScript to invoke code dynamically in the form of `eval` and `function` initialization [14, Appendix B3]. According to Richards

[1] Requires a defined setter function for invocation
[2] Requires a defined getter function for invocation

et al. `eval` is frequently used and often comes with unpredictable behavior [15, p. 11]. Furthermore functions can be indirectly invoked within a specified scope, using `call` and `apply`. Finally there is getter and setter invocation, which are invocations that are triggered by assignments and property names.
The original research obtained dynamic call graphs by instrumenting the input scripts. It is however difficult to prove that instrumented scripts have the same semantics as the original scripts (except for having added logging functionality). They are also hard to read and analyze. Another problem was that dynamic code invocation could not be logged, because instrumentation would not be added on-the-fly. Getter and setter invocation are hard to detect as well, as getters and setters are optional features for expressions. To prevent such issues, several attempts other than instrumenting code have been made. Such as a debugging extension for Chrome, a Firebug extension and modifying existing JavaScript engines. Despite these attempts, rewriting original code with instrumentation seemed to be the most feasible overall. This means that only regular invocation has been recorded. Instrumentation also comes with a couple of validity issues, which have been described in the threats to validity (section 10.3).
The upcoming section discusses how instrumentation aids in acquiring a dynamic call graph and how the instrumentation method overcame problems from alternative methods. The subsequent section mentions post-processing of the acquired dynamic call graphs, as post-processing allowed to have a more accurate model.

8.3.1 Instrumentation

In the ACG study, a dynamic call graph is realized by creating an instrumented version of the analyzed scripts. The instrumentation would log two things:

- The position of the function call on each function call
- The position of the invoked function on each function invocation

The logged positions should be of the same format as the positions in the static computed call graph, so that they are comparable in a later phase. This means they have to be of the same 4-tuple format (see section 8.2.1). Furthermore it is important to emphasize that this instrumentation has to preserve the semantics of the existing code and should solely consist of added logging functionality. A dynamic call graph comes about by running an application/library and connecting call sites to invoked functions. The call graph is exported after attempting to use all functionality of the scripts. Such rewrite required code to be parsed in order to rewrite functions and function calls. To perform an accurate rewrite to add instrumentation to the scripts, Rhino (an open source implementation of JavaScript) has been used. This is to take possible grammar limitations into account when analyzing the computed call graphs. Having the dynamic call graph emerge from external tooling, allows us to externally validate the output of the call graph algorithm implementations in Rascal. This comes with the assumption that Rhino parses and converts source code reliably. The latest version of Rhino (1.7R5) has been used with simple bugfixes (bug #798642, #800616 and Github issue #129) to support proper conversion of an abstract syntax tree (AST) to source code. The `toSource` functions of AST nodes have been modified in Rhino to output instrumented source code.
The first subsection discusses how function calls are tracked. The subsequent subsection talks about the tracking of functions and connecting them to the correct call sites.

8.3.1.1 Call Site Tracking

This subsection discusses how call sites are tracked. The **final solution** for instrumentation of call sites will be presented at first. To illustrate how this works, the code of a function that is going to be instrumented is followed by its instrumented version:

```
1 function f() {
2   var x = doSomething();
3   return x;
4 }
```

This is rewritten to:

```
function f() {
  var x = (logCallExpression("filename.js", 1, 2, 3) ||
    popCallStack(doSomething()));
  return x;
}

function logCallExpression(file, line, startOffset, endOffset)  {
  // creates a correct function call vertex on the vertex stack
  ...
  return false;
}
```

This method relies on the earlier discussed *truthy* and *falsy* system behind JavaScript. This is previously discussed in section 6.1. By appending a disjunction with the rewrite operation we can preserve original semantics. The left expression adds a vertex for the call site to a call stack and returns a constant falsy value. This falsy return guarantees that the disjunction evaluates to the right expression. This can be described as follows: "The `||` operator produces the value of its first operand if the first operand is truthy. Otherwise, it produces the value of the second operand" [14, p. 40]. In other words, in JavaScript $\perp \vee \pi$ will always resolve to expression π, which is in turn truthy or falsy.
The parentheses around the logging call and the function execution are introduced to preserve the associativity of the original call. A call stack is necessary because the left hand side of a disjunction would be evaluated first prior to executing nested calls. The following example clarifies this:

```
f(g(), h());
```

is rewritten to:

```
(logCallExpression(...) ||
  popCallStack(f(
    (logCallExpression(...) || popCallStack(g())),
    (logCallExpression(...) || popCallStack(h()))
  ))
);
```

In the example the call stack gets appended for the invocation of `f`. Afterwards the call stack gets appended for the invocation of `g`, followed by the invocation of `g`. This is followed up by adding a vertex to the call stack for invocation `h`. Then `h` gets invoked. Finally `f` gets invoked. The `popCallStack` function removes the last call site vertex from the call stack and returns the argument it is invoked with. This allows the call stack to be maintained, even after native function calls, where we do not have access to its implementation. The call stack is thus necessary because the logging for `f` would be executed prior to the logging of `g` and `h`, whilst those function calls will be evaluated first.
The solution using a disjunction, overcomes problems with the more intuitive rewrite, where one spawns an immediately-invoked function at each place of a function call. Such solution is similar to the aspect-oriented programming (AOP) proposal for JavaScript by Washizaki et al. [26]. With such solution, the original function `f` is rewritten to:

```
function f() {
  // Call instrumentation function and pass 'this' reference and arguments object
  var x = (function() {
    logCallExpression("filename.js", 1, 2, 3);
    var result = doSomething();
    return result;
  }).apply(this, arguments);
  return x;
}
```

This however does not work, because the `arguments` object is altered for calls within the introduced nested function. This is due to the arguments object changing per function scope. Such solution thus imposes different semantics. The `arguments` object will remain the same in the final solution as this solution does not introduce nested scoping. The problem specifically occurs when the `doSomething` function refers to `arguments.callee.caller`. The `arguments.callee.caller` variable refers to the function where the call originates from [12, p. 173], which in this case is the anonymous immediately-invoked function, rather than `f`. Invoking the `doSomething` function with a similar `apply` call, did not make a difference.

8.3.1.2 Target Tracking

After invoking a function, a target vertex had to be introduced and connected to its call site. Simply calling `logInvokedFunction` (with location parameters) on top of each function implementation block, connects a target to its call site. The `logInvokedFunction` takes the call site from the stack of call sites (with a peek method, as the call site will be popped from the stack later). A function that would be logged, is rewritten as follows:

```
1 | function doSomething() {
2 |   ... // original implementation
3 | }
```

This is rewritten to:

```
1 | function doSomething() {
2 |   logInvokedFunction("filename.js", 4, 5, 6);
3 |   ... // original implementation
4 | }
```

After gathering all data, it appeared that the mechanism for connecting functions and call sites was insufficient. It was observed that the dynamic call graph was distorted by three phenomena:

1. Dynamically evaluated code falsely connected `eval` call sites to functions. The code `eval("f()")` connected the `eval` call site to the function f, rather than `eval` to a native and `f()` to f.
2. Indirect native invocation (`call` and `apply`) falsely connected a function to `call` or `apply` call sites. The problem is that `call` and `apply` are in fact native functions that serve as an intermediate.
3. Native callbacks could get connected to function calls that were not (yet) resolved. These unresolved calls could be calls to native functions, calls to dynamically injected code or simply non-instrumented calls.

To check for native function invocation, two other parameters have been added to the logging of function calls (thus creating call site vertices). One is the name of the call target (in case it is a name) so that we can check if it is a native function (e.g. `String` or `RegExp`). The other parameter is a simple boolean to filter `eval` from the dynamic call graph, which was checked during the rewrite.
In order to mitigate the third problem, the `arguments.callee.caller` variable has been added as an argument to the `logInvokedFunction` function call. This allows the logging functionality to validate whether its caller would be known as it yields the function object of its caller. The value of `arguments.callee.caller` is `null` in case of an invocation from top-level (thus global scope) or callbacks. The information of whether a call site was on global scope also had to be stored, as top-level callers would also result in `null` for the `arguments.callee.caller` argument. It is widely known that `argument.callee.caller` is unreliable; in some callbacks its value did not appear to be `null`, whilst it was certainly invoked from a native call site. This happened for instance with `body.onload` callbacks. The method however did at least add an extra barrier to diminish the amount of spurious call edges.

The final function header for logging call expressions looks as follows (note that the disjunctive solution to append logging remained the same)[3]:

```
function logCallExpression(file, line, startPosition, endPosition,
  functionProp, globalScope, proxyFunction)
```

The final function header for adding target vertices looks as follows:

```
function logInvokedFunction(file, line, startPosition, endPosition, caller)
```

8.3.2 Post-Processing

As described in the previous paragraph, spurious edges could be added by indirect native invocation and dynamically evaluated code. In the ACG study `eval`, `call`, `apply` are considered natives. To keep the study comparable, these spurious edges were removed after obtaining the dynamic call graph. Invocations by `eval` were detected at runtime due to the added parameter to detect native functions like `eval`. Afterwards it was decided that other indirect native invocations could still be removed after obtaining the dynamic call graphs. A post-processing tool has been written in Java that uses Rhino to parse a given JavaScript program. Furthermore it imports a dynamic call graph for that given JavaScript program. For each call site in the dynamic call graph, it is checked whether in the original source code this a `call`, `apply` or `eval` invocation. This diminishes dynamic call graph problem #1 and #2 from the previous paragraph. After removing the spurious edges, the residue is stored to compare it with the static call graphs. Removing these edges would also make it more comparable to the ACG study, as they considered those three functions to be native functions.

8.4 Comparison Implementation

This section shortly describes how comparison between the static call graphs and the dynamic call graphs is done in Rascal. The post-processed dynamic call graphs are stored in the form of a plain-text file, where call sites are connected to targets. An example of such connection between call site and target looks as follows: `game.js@50:3444-3463 -> mootools-beta-1.js@1518:50493-51874` [4].
It has been chosen to store the static call graphs in the same format as the post-processed dynamic call graphs, so that they would be stored in a light-weight way, without losing the necessary information. A Rascal module has been introduced (`CallGraphComparison`) that arranges that call graphs for programs are stored within the same format, so that they could be compared. It further allows to import dynamic and static call graphs of this format and it allows to calculate the recall and precision of these imports (see section 9.1). It only considers the call sites that were found during runtime analysis (thus the call sites that are present in the dynamic call graph). To determine the call target precision and recall, the sets of the target vertices are compared on a per call site basis. The comparison of the targets may be implemented as follows:

```
num calculatePrecision(set[Vertex] dynamicTargets, set[Vertex] staticTargets) =
  ((size(dynamicTargets & staticTargets)) * 1.0) / size(staticTargets);
num calculateRecall(set[Vertex] dynamicTargets, set[Vertex] staticTargets) =
  ((size(dynamicTargets & staticTargets)) * 1.0) / size(dynamicTargets);
```

In the implementation for this thesis, the code above is used to calculate a cumulative precision and a cumulative recall value. These were averaged by dividing the cumulative numbers with the amount of call sites that were found in the dynamic call graph.
Determining precision and recall for edges (thus full relations) is implemented with the same demonstrated code, where `rel[Vertex, Vertex]` is the type of both the static and dynamic call graphs. The intersection works because a `rel` (relation) is a set of tuples.

[3] Note that the two final function headers have a different naming in the implementation
[4] This is a real example that originates from the dynamic call graph from the Beslimed subject program

8.5 Statistics

The following table displays the written artifacts, their programming languages and their lines of code, which have been developed for this thesis project:

Artifact	**Language(s)**	**Lines of Code**
JavaScript Grammar with Unit Tests	Rascal	992
JavaScript Grammar Test Input Scripts	JavaScript	2242
Algorithm Implementation with Unit Tests	Rascal	1511
JavaScript Algorithm Test Input Scripts	JavaScript	622
Dynamic Call Graph Rewriter and Post-Processor	Java, JavaScript	1586
Edited Rhino JavaScript Engine[5]	Java	150

Table 8.2: Statistics

The final project therefore consists of about 7100 lines of code. Bare in mind that this is just an approximation, as the Rascal numbers are SLOC (source lines of code), whereas the JavaScript and Java has been measured in terms of LOC (lines of code). The difference is that SLOC measurements contain lines of comments and white spacing, whilst LOC measurements do not. The Cloc v1.60[6] utility has been used to measure LOC of JavaScript and Java. The amount of Rascal code has been measured with the `countSloc` function.

[5] Only contains the amount of lines that were actually appended
[6] http://cloc.sourceforge.net

Chapter 9

Results

The ACG study used a specific set of input scripts for the call graph algorithms. The versions that were used in the study were unknown for most of the scripts. The same input has been used where possible, whereas in other cases different versions of scripts and libraries have been used. The input set for the algorithms of scripts consisted of 3dmodel, Beslimed, Coolclock, Flotr, Fullcalendar, HTMLEdit, Markitup, Pacman and PDFjs. Most of these scripts rely on frameworks like jQuery, MooTools and Prototype. Others were standalone scripts. One of the original input scripts (Pong) had to be omitted, because it did not function properly. The following table gives an overview of the input scripts, the exercised versions and their relative size:

Program	Version	Framework	Lines of Code	Functions	Calls
3dmodel	Unknown	None	432	41	150
Beslimed	Unknown	MooTools	3325	703	2017
Coolclock	Unknown	jQuery	10799	1214	3792
Flotr	0.2.0-alpha	Prototype	5177	698	2503
Fullcalendar	1.5.3	jQuery	15467	1731	7420
HTMLEdit	4.0.3	jQuery	3335	387	1285
Markitup	1.1.12	jQuery	5210	557	1848
Pacman	Unknown	None	2177	152	484
PDFjs	Unknown	None	123259	964	3570

Table 9.1: Input programs and their sizes

These numeric values are incomparable to the ACG study, because changes to scripts had to be made quite often, in order to have the scripts parseable in Rascal. The scripts were also formatted with Rhino, which might convert code differently than its input. This was done for readability. Appendix C describes what has been changed per input program in order to gather data. The lines of code have been calculated with the same utility as the one in the ACG paper: Cloc[1]. The functions and calls have been calculated with Rhino 1.7R5.

On page 35, I have discussed threats that could affect the completeness as well as the validity of the dynamic call graph. Measurements of possible dynamic code threats are listed in the following table:

[1] http://cloc.sourceforge.net - cloc v1.60

Program	Eval Expressions	New Function Expressions	Getters	Setters
3dmodel	0	0	0	0
Beslimed	3	0	0	0
Coolclock	0	0	0	0
Flotr	4	0	0	0
Fullcalendar	1	1	0	0
HTMLEdit	0	0	0	0
Markitup	1	1	0	0
Pacman	0	0	0	0
PDFjs	0	0	40	1

Table 9.2: Input programs and their listed threats for the dynamic call graph

9.1 Static Call Graph Validation

This section describes two different ways of validating the static call graphs by comparing them to the dynamic call graphs.

9.1.1 Validation Per Call Site

The call graph data has been compared for each JavaScript program, using a recall and precision formula. The recall for a call site is computed with the following formula: $\frac{|D \cap S|}{|D|}$, where D represents the set of targets in the dynamic call graph for a given call site, and S represents the set of targets in the static call graph for a given call site. Precision for a call site is calculated with the same variables, but a different formula: $\frac{|D \cap S|}{|S|}$. The recall formula yields the percentage of correctly computed targets among all existing (thus dynamically measured) targets of a call site. The precision formula represents the percentage of correct call targets among all computed targets per call site. The precision and recall of a program are the average precision and recall values of all its call sites that were found during dynamic call graph recording. These average recall and precision values are computed by looping over all call sites that are present in the dynamic call graph and averaging over a cumulative recall and cumulative precision value. The following algorithm represents the just described process of acquiring average precision and recall of a program:

Algorithm 4 Averaging Precision and Recall over Call Sites

Input: Static Call Graph S, Dynamic Call Graph D
Output: Average Recall, Average Precision

1: $callSites := \{\ \pi \mid (\pi, \pi') \in D\ \}$
2: $cumulativePrecision := 0$
3: $cumulativeRecall := 0$
4: **for all** $\pi \in callSites$ **do**
5: $\quad staticTargets := \{\ \pi' \mid \exists \pi' .\ (\pi,\pi') \in S\ \}$
6: $\quad dynamicTargets := \{\ \pi' \mid \exists \pi' .\ (\pi,\pi') \in D\ \}$
7: $\quad cumulativePrecision := cumulativePrecision + \frac{|dynamicTargets\ \cap\ staticTargets|}{|staticTargets|}$
8: $\quad cumulativeRecall := cumulativeRecall + \frac{|dynamicTargets\ \cap\ staticTargets|}{|dynamicTargets|}$
9: $averagePrecision := \frac{cumulativePrecision}{|callSites|}$
10: $averageRecall := \frac{cumulativeRecall}{|callSites|}$

Note that the divisor of the cumulative precision and recall is determined by the amount of call sites in the dynamic call graph. In other words, a call site that is not present in the static call graph, has a precision of 0 and a recall value of 0. The average recall and precision value, do not indicate the accuracy of call sites that have been analyzed statically, as non-analyzed call sites may suppress these values. This form of validation is adopted to the thesis to compare the results to the ACG study, as

the ACG study did the same computation.

9.1.2 Validating Edges

Validation on a per call site basis reasons about the average call site and its accuracy. This however does not view the absolute ratio between spurious and correct edges in the static call graph. A spurious edge is an edge that is computed during statistical analysis, which does not appear in the dynamic call graph. Furthermore, the previous method does also not reason about the overall completeness of the static call graph in comparison to the dynamic call graph. Therefore precision and recall have been computed for the complete edges in the call graph. The formula for recall is similar to the previous formula and is as follows: $\frac{|D \cap S|}{|D|}$. The set S and D however, respectively represent the static call graph and dynamic call graph. In other words, S entails the set of tuples from call site to target (edges) in the static call graph, whereas D does the same for the dynamic call graph. This method requires prior filtering of the static call graph. This is necessary because it is typical for dynamic call graphs not to cover 100% of a program. The filtering process entails disregarding call edges that start from a call site that would not be present in the dynamic call graph. Coverage can be lower for programs with libraries, as it is common that only a small percentage of the functions will be covered. The following image shows how this filtering applies prior to computing recall and precision for edges. Note that similarly named vertices are considered to be of the same identity in both graphs:

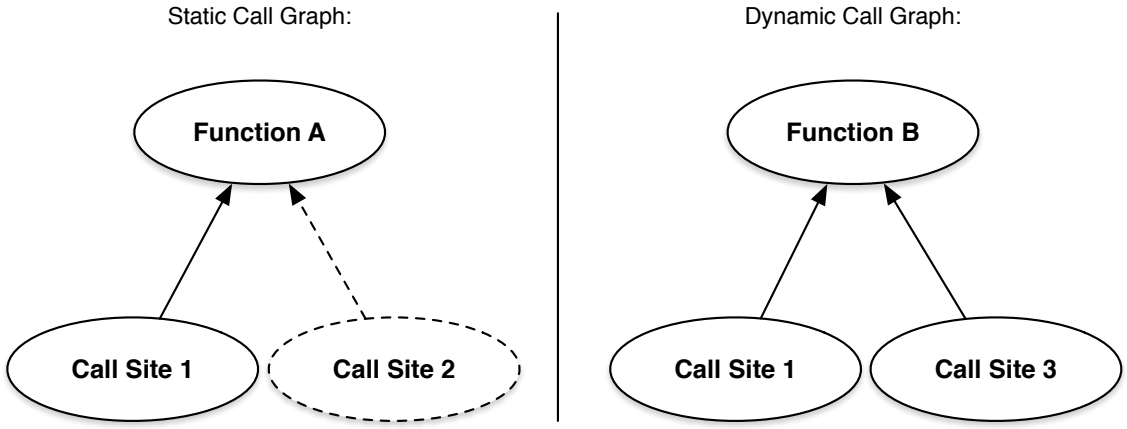

Figure 9.1: Filtering by call sites before measuring precision and recall for edges

Both the dashed call site and its edge to the invoked function A are removed from the static call graph. This is due to call site #2 not being present in the dynamic call graph. The fact that the static call graph computed a call to a different function (A instead of B) does not matter, as the call site is present in both the static and the dynamic call graph. The graphs from the example would therefore not intersect. Note that the validation by averaging applied the same filtering implicitly, as precision and recall would only be computed for edges that were also present in the dynamic call graph.
The following algorithm in pseudocode represents the just described recall and precision computation for edges:

Algorithm 5 Recall and Precision of Call Graph Edges

Input: Static Call Graph S, Dynamic Call Graph D
Output: Recall, Precision

1: $dynamicCallSites := \{ \pi \mid (\pi, \pi') \in D \}$
2: $staticEdges := \{ (\pi, \pi') \mid \exists (\pi, \pi') \,.\, ((\pi, \pi') \in S \wedge \pi \in D) \}$
3: $precision := \frac{|D \cap staticEdges|}{|staticEdges|}$
4: $recall := \frac{|D \cap staticEdges|}{|D|}$

Note that the static edges that are used to compute precision and recall are determined by the call sites that are present in both the dynamic and static call graph.

9.2 Data

Each rewritten program was ran locally without hosting, except for PDFjs, Coolclock and Flotr. Hosting for those has been necessary due to AJAX[2] calls and security issues with iframes. For PDFjs an old rewritten version has been used, due to performance issues and issues with strictness of ECMAScript (disallowance of the `arguments.callee.caller` reference) with the later instrumentation code. Each of the rewrites has been manually stressed in Google Chrome (v36) so that the dynamic call graph increased. This is done by exercising the user interface with manual input events. The dynamic call graphs function coverage has been measured for each of the programs, indicating whether a significant part of the code has been hit or not. The dynamic function coverage is a percentage that indicates the magnitude of functions (excluding library/framework functions) that are part of the dynamic call graph. The total amount of functions used in this calculation are the functions that are statically determined after parsing the JavaScript code. Consider a dynamic call graph with 5 different functions, whilst statically 10 functions were measured. Such dynamic call graph would yield a 50% dynamic call graph coverage. The lower percentage could be due to dead code, unreached code, callbacks, indirect invocations etc.
The ACG study considers the total amount of functions hit, rather than the functions that are part of the dynamic call graph. This study only counts functions that are actually part of the dynamic call graph, which is more strict as this does not consider native callbacks at all. The following table lists the measured dynamic call graph coverage per input program of this study:

Program	Dynamic Call Graph Coverage of Functions
3dmodel	63,33%
Beslimed	62,79%
Coolclock	83,33%
Flotr	59,11%
Fullcalendar	62,23%
HTMLEdit	50,00%
Markitup	46,93%
Pacman	85,52%
PDFjs	77,28%

Table 9.3: Dynamic call graph coverage for each input program

This shows that for three programs 3dmodel, HTMLEdit and Markitup the function coverage might be considered scarce. The ACG authors examined 3dmodel and found out that it contains a lot of dead code. HTMLEdit did contain some Internet Explorer only functions, as it spawned popups that a certain feature was unavailable for other browsers. This could be the reason why its coverage is low. For Markitup some manual checking has been done and it turned out that it had a lot of native callbacks[3]. It might be that programs with little coverage have relatively more native callbacks than

[2] AJAX: Asynchronous JavaScript And XML
[3] Native callbacks are considered to be callback functions that are invoked by a native event

the ones with higher coverage. Other than that it might also be that significant parts of programs were not hit. This however is not considered likely, because previous observations that did consider native callbacks turned out in higher coverage values.

The dynamic call graphs have been imported after post-processing (see section 8.3.2) and compared within Rascal. They were compared with both the pessimistic and optimistic static call graphs using the aforementioned precision and recall formula. The two upcoming sections present the data that has been gathered by comparing the computed static call graphs with the recorded dynamic call graphs.

9.2.1 Call Graph Targets Per Call Site

This section shows charts where the average precision and recall results are compared with the results of the ACG study. Their statistical data is taken from Github[4]. The precision and recall values in this section entail the average precision and recall on a per call site basis (as described in section 9.1.1). The data gathered in this thesis is compared to two different results; results of the ACG study and results that are computed from downloaded call graph models (exported call graphs). These models are the call graphs that were used in the ACG study and were added to the same Github repository after the ACG research. Computing average precision and recall on a per call site basis for the downloaded models, resulted in different values than mentioned in the ACG study. This is despite the fact that one of the authors validated the formula to compute average precision and recall on a per call site basis. The problem is confirmed by the authors and is under further investigation. The downloaded models were imported without native call targets, in order to have them more comparable to the call graphs of this study. The precision and recall values of the ACG study however, do consider native call targets. Having the downloaded models take part of the analysis, means that we know how the precision and recall value came about for two out of three data sources. The plotted data in this section is also presented in tables in appendix D.
The following data has been gathered (using algorithm #4):

Pessimistic Call Graph

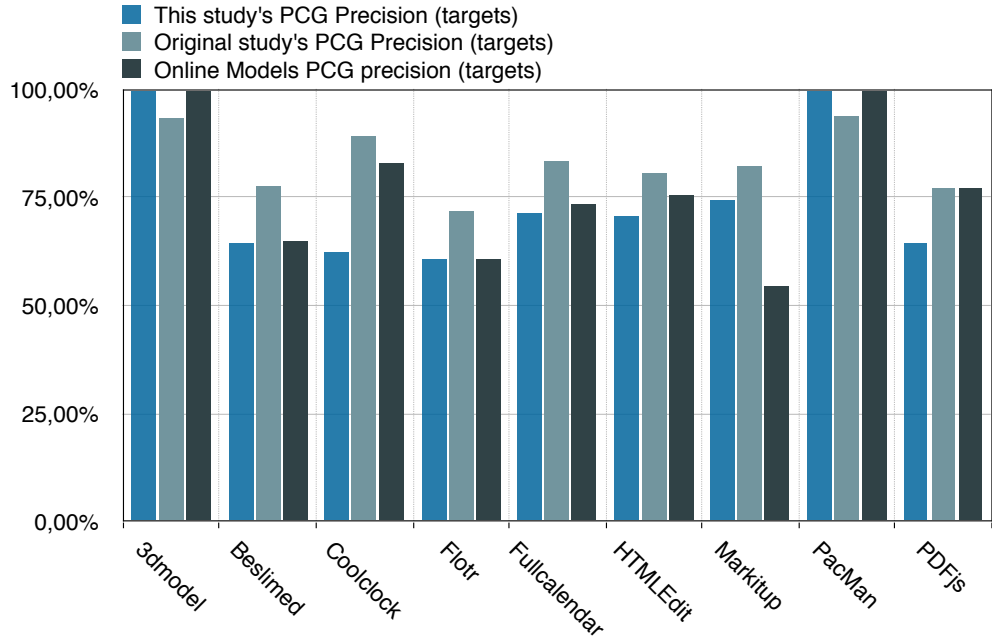

Figure 9.2: Pessimistic call graph average precision per call site, compared to the original study results and the online models

[4]https://github.com/asgerf/callgraphjs/tree/master/callgraphs/

In figure 9.2 the precision of targets of the pessimistic call graphs has been compared between both this study and the ACG study. We can see that only 2 programs out of 9 programs of this study ended up with a higher precision value than the ACG results. Coolclock has the largest deviation of roughly 17% less precision in this study. Furthermore the precision values of this study are often closer to the computed values of the online models. Coolclock and PDFjs have more than 10% less precision than the precision values for the online models. The precision for Markitup however, is about 20% higher in this study in comparison to the online models. The computed Markitup precision is still 8% less than in the ACG study. The average precision of the displayed data in figure 9.2 is 74%, whilst its median is 71%.
The following figure presents the recall of targets of the pessimistic call graphs, compared to that of the ACG study and their exported call graph models:

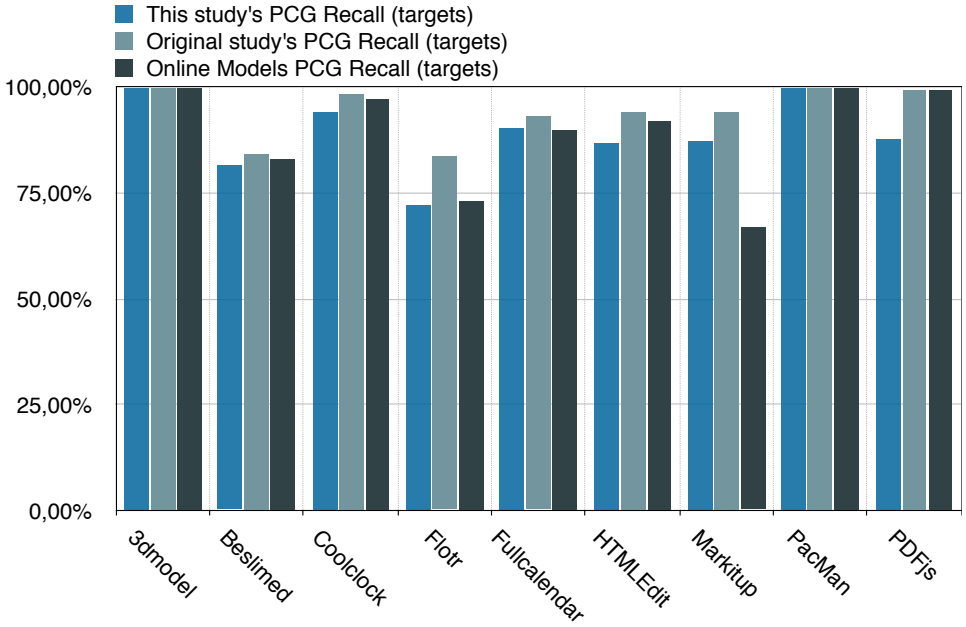

Figure 9.3: Pessimistic call graph average recall per call site, compared to the original study results and the online models

The figure shows us that recall often does not deviate much between both the studies. Flotr and PDFjs are exceptions, which both have about 10% less recall than the analysis from ACG. What is remarkable though is that the recall value of the downloaded models of Flotr results in almost the same recall value as the one computed for this study, whilst the ACG recall is 12% more. The recall deviation between the computations of this study and the online models is only about 0,8%. The average recall of the displayed data in figure 9.3 is 89%, whilst its median is 88%.

Optimistic Call Graph

The following figure shows the difference between the precision of the optimistic call graphs of this study, the ACG study and the exported call graph models:

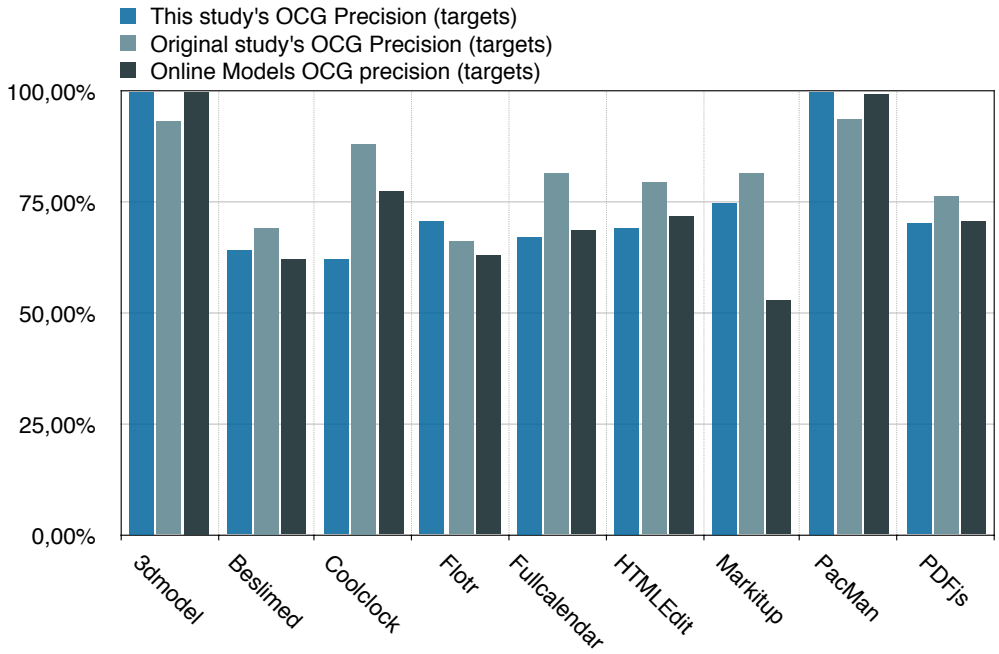

Figure 9.4: Optimistic call graph average precision per call site, compared to the original study results and the online models

Deviations are similar to those in the pessimistic analysis. PDFjs and Flotr have precision increments with the optimistic call graphs in this study. Flotr has a higher precision than the ACG study with optimistic analysis, whilst it has a lower precision with pessimistic analysis. PDFjs has gotten closer to the precision of the ACG study. The computations for the online models are often similar to the thesis' results. The average precision of the displayed data in figure 9.4 is 75%, whilst its median is 70%.

The last figure represents the recall values of both studies and the exported call graphs for the optimistic call graphs:

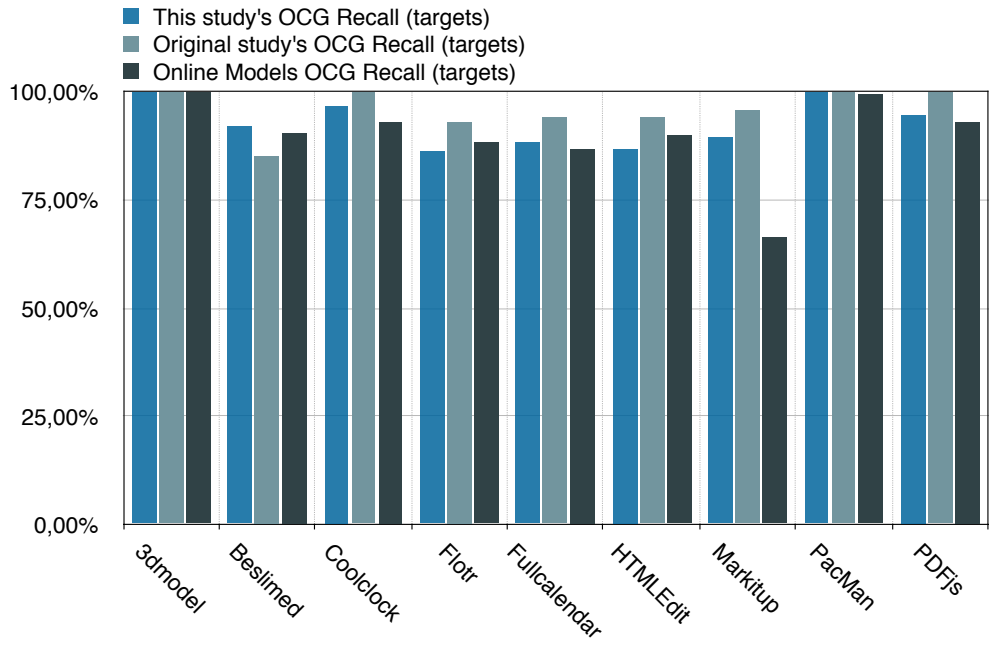

Figure 9.5: Optimistic call graph average recall per call site, compared to the original study results and the online models

The recall values between both studies do not seem to deviate much. With optimistic analysis, the recall value of PDFjs got closer to the recall value of the ACG study. It is remarkable that the recall value of the exported model of Markitup is an outlier with more than 25% less recall than this study. The value also does not correspond to that of the ACG study. The average recall of the displayed data in figure 9.5 is 93%, whilst its median is 92%.

9.2.2 Call Graph Edges

The ACG study calculates recall and precision on a per call site basis using the algorithm specified in section 9.1.1. Precision and recall values are averaged per program as described in algorithm #4. This study however also considers the overall precision and recall of the static call graph (using algorithm #5 from section 9.1.2). The static call graph is excluding the edges that come from call sites that were not recorded by the dynamic call graph, as there is no way to validate those edges. The precision indicates the ratio of spurious edges to computed edges. The recall indicates the ratio between the correctly computed edges and the dynamic call graph. The following data is gathered by considering edges:

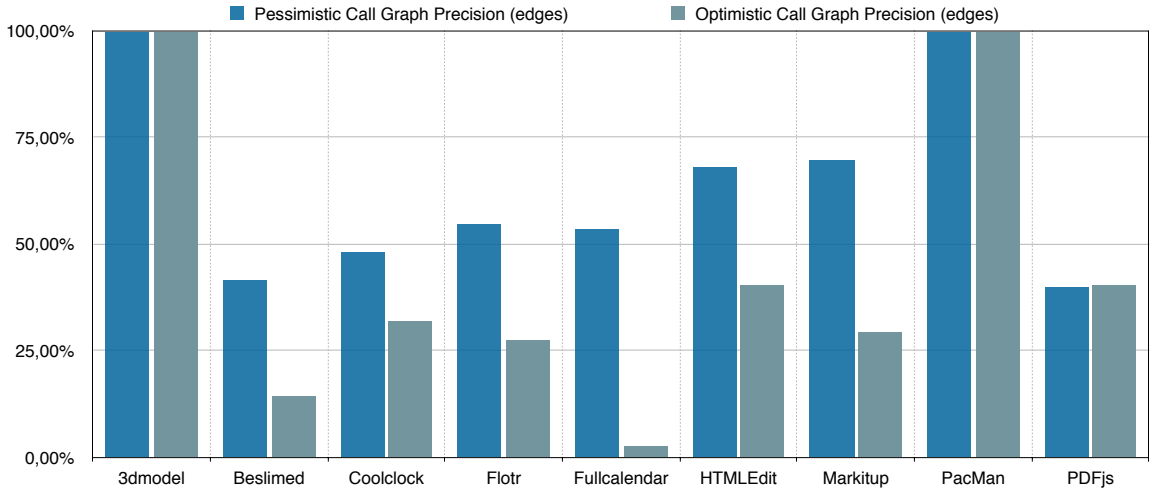

Figure 9.6: Precision for each input program based on call graph edges

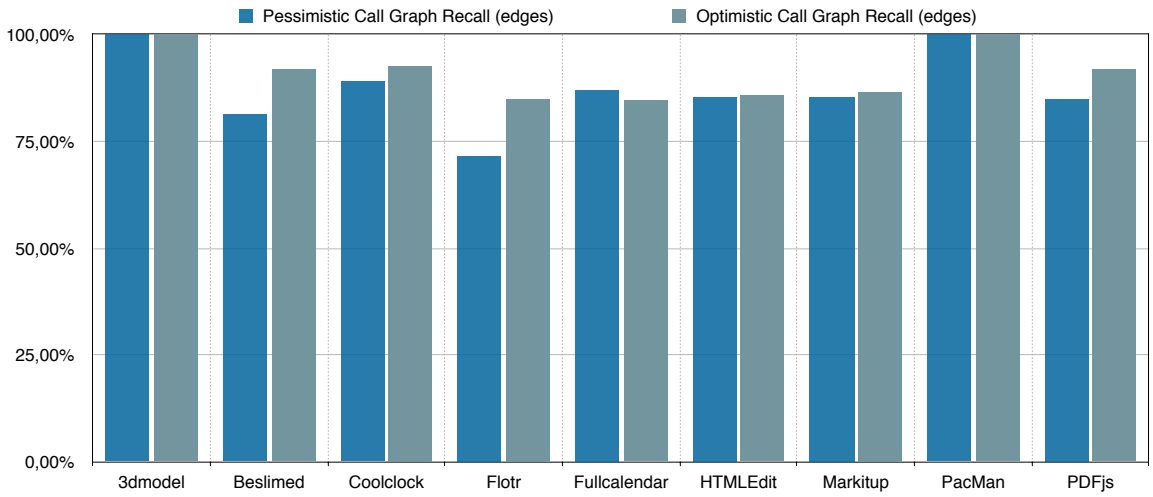

Figure 9.7: Recall for each input program based on call graph edges

Figure 9.6 demonstrates the percentage of correctly computed call edges within the pessimistic analysis. The higher this precision percentage, the more accurate the computation is in general; less spurious edges would have been computed. Figure 9.7 shows the contrary, where the percentages signify the correctly computed call edges from the total call edges that were recorded in the dynamic call graph. This means that the higher the percentage of the recall analysis, the less missing edges in the computed call graph. The trend seems to be that precision often decreases (or stays the same) with optimistic analysis, whilst exceptionally an increase can be observed for PDFjs. Optimistic analysis increases the recall value for 7 out of 8 possible subject programs.
The average precision in terms of pessimistic analysis with edges comparison is 64% and the precision median is 55%. With optimistic analysis the average precision dropped to 43%, whilst the median dropped to 32%. The average recall of pessimistic analysis with edges comparison is 87%, whilst the median is 85%. The optimistic analysis turned out to have an average recall of 91% and a recall median of 92%.

Chapter 10

Discussion

This chapter will discuss and analyze the results that are presented in the previous chapter (page 48). The retrieved data regarding call target accuracy per call site will be compared to the ACG study and their exported call graph models (explained in section 9.2.1) at first. Afterwards, the precision and recall values of call graph edges will be discussed. Finally this chapter explains the threats to validity. Please note that the analysis for the Markitup program contains a known error. This will be further mentioned in this chapter.

10.1 Comparison to the Original Study

The results of the ACG paper have been plotted with those of this study in section 9.2.1. This is done with the knowledge that they are not fully comparable due to differing versions of scripts and having no insight in the call graphs that were used in the ACG study. The exported call graph models on the ACG Github repository have been adopted to this study, because they do provide insight in why precision and recall values might differ. These models however are not the same models as the ones that were used during the study, due to licensing issues. In this paragraph the data of this study, the ACG study and their exported models are compared. There will be reflected on the causes of deviation and why the data is (sometimes) considered to be difficult to compare. First the pessimistic call graph analysis will be discussed, which will be followed up with a discussion about the optimistic call graph analysis.

10.1.1 Pessimistic Call Graphs

3dmodel does not rely on any library and results in precision and recall values of 100%. The ACG study has 6,6% less precision, whereas the online models result in a precision of 100% as well. The observed program does not instantiate many objects and declared functions loose instead of attached to objects. A result of that is uniquely named functions, which are therefore easy to resolve. The unique function names allow for this precision value, as field-based analysis is often found to suppress the precision values.
Beslimed has a lower precision value than the ACG study ($\sim$ 13%). The online models however seem to result in approximately the same values. Different versions of the library can be the reason for the precision deviation between this study and the ACG results. Field-based analysis has suppressed the precision the most, which has been observed by examining spurious edges. This was an initial design decision of the static call graph algorithms (described in section 3.2). Furthermore Mootools overwrites native functions (e.g. `Window`). It is unclear how the ACG study took those into account, but this study disregards natives and takes overridden native functions into account. This could be a reason for the deviation. Moreover it is unclear which Beslimed version (and thus Mootools) is used by the ACG authors, as Beslimed is unversioned and the authors did not publish their input programs. Finally some missed calls were due to not tracking dynamic fields (e.g. `functions[x] = Date.now`), which has been a design decision for the algorithms as well (design decision #3).
Coolclock is an unversioned program as well, which cause precision and recall values to deviate, as

versions may not correspond. By observing the spurious edges in the static call graph, it was found that field-based analysis had a significant negative impact on the precision value. A possible reason for deviation with the ACG study and the extracted call graphs is a different version of Coolclock itself, but more importantly jQuery. The jQuery version that was used in this study was the latest version (1.11.1 at the time of writing), whilst the jQuery version used for the online models was version 1.7.2. The field-based analysis caused a lot of spurious edges in the static call graph, as functions with common names could not be distinguished. Coolclock did not call many jQuery functions, but internally jQuery often called itself. In the version used in this study, 4 `each` fields were present. Whereas in the version from the online model, only 2 `each` fields were present. Several sites therefore pointed to 4 rather than 2 targets. The same goes for example for the `ready` function, of which 3 fields have been found, whereas the old jQuery version only has 2 of those. Another example is the `promise` function, which is used internally in jQuery and appears to be declared 5 times. The old version does not have this field. Therefore it is likely that spurious edges caused by field-analysis be the largest contributor to a lower precision value.
The version of Flotr has been the same version as the one in the original study. The Prototype library on which Flotr depends had to be replaced, due to parsing issues in Rascal with the embedded version. This replacement on its own may be a reason for deviations. Field-based analysis was observed to cause spurious edges, which affected the precision value. Further observation indicated that some native functions were overridden by Prototype (e.g. `Object.keys`). It is unclear if the original study took those into account, which could have caused deviations in results. Moreover the computation for the extracted call graph models resulted in a similar precision and recall for Flotr. These extracted all graph models used a closely related version of Prototype (one version earlier) to the version used in this study, which might be the reason for similar results.
Fullcalendar 1.5.3 has been used for both studies. The extracted models by the ACG authors indicate that the same jQuery version was used. They did however use a jQuery-UI version that had been separated in modules. The version of that jQuery-UI remains unknown. This thesis used a complete jQuery-UI version, which could be the reason for deviations. Like observed in Coolclock, a different version may yield more spurious edges due to the field-based analysis. The precision value of the extracted models is similar to that of this study. The same goes for the recall value, which in turn does not deviate much from the ACG study. This indicates that the average set of targets for a call site in the pessimistic call graph is quite complete (as the recall value is over 90%), but polluted with spurious edges.
HTMLEdit 4.0.3 has been exercised in both studies. The ACG study computed a precision of about 10% higher than this study, whilst the precision of the extracted models is in between of both values. The same jQuery version has been used for the extracted models. By examining the spurious edges, it turned out that spurious edges were often caused by field-based analysis due to similar function names. Examples of those were found to be function fields with the following names: `each`, `data` and `css`. A reason for the deviations can be that this study used a different HTML page (demo-full.html) to obtain the dynamic call graph. Exercising this different page resulted in a higher dynamic call graph coverage. This could also be the cause for the slightly lower recall value as more paths would have been recorded ($\sim 8\%$).
The results for Markitup are close to that of the ACG study, where the precision turned out to be $\sim 7\%$ less. Recall turned out to be $\sim 8\%$ less than the ACG study. The reason the results of this study might deviate is because an old grammar version had to be used in order to parse this program. This grammar version sometimes reported faulty offsets for the ending offset of functions. This error was found for a couple of edges in the static call graphs, which caused these edges to be false negatives in the precision and recall formula. The dynamic call graphs did not contain this error as they came about by a instrumented version that was rewritten by different parser (Rhino). Correct offsets for the static call graphs would have intersected with the dynamic call graph, which in turn would have positively affected precision and recall values. The extracted call graphs ended up with a significant lower precision and recall value, but the further reason for this has not been investigated.
The pessimistic analysis for Pacman resulted in a precision of 100% for both the extracted models and this study. The ACG study turned out to have a slightly lower precision value. Pacman does not rely on any framework and turned out to have unique field names for functions. These unique names positively affected the precision values.

As for PDFjs it is known that they examined a different version. PDFjs did not seem to be versioned, but according to their paper their exercised PDFjs consisted of 31694 lines of code. In this study the lines of code of PDFjs were almost 4 times as much. Other than that PDFjs has very specific features, which made it difficult to have a representative dynamic call graph. Examples of this specific functionality are arabic characters or embedded fonts in PDF files. It is unknown if such paths were triggered in the ACG study or in the extracted dynamic call graph from their Github repository. A large set of PDF files with such specifics has been used as an input in this study, in order to obtain dynamic call graphs with as many functions covered as possible. PDFjs has also been measured to have a lot of getters and some setters (see table 9.2). These functions are not accounted for in the dynamic call graph and thus causes the verification model to be incomplete. We can therefore state that the recall values are higher than they should be for PDFjs, as the static call graph algorithms do not take getters and setters into account. Furthermore it was found that precision values for PDFjs were often suppressed by field-analysis. Good examples of those are functions attached to fields with a common name like `get` or `getBytes`. The `get` function for example caused `get` call sites to connect to 9 different targets. This would affect precision negatively. It is possible that less `get` fields were present in the version used in the ACG study.

10.1.2 Optimistic Call Graphs

It is clear that the optimistic call graph analysis is susceptible to the same issues as the pessimistic call graph algorithm, as it is based on the same fundamentals. In general the precision values for the optimistic call graph (figure 9.4) seem to be lower than the pessimistic call graph precision values. This is consistent with the definition of optimistic call graphs, as optimistic analysis traverses all paths it can reason about. In other words, more spurious paths might be considered by an optimistic call graph analysis. This paragraph will reason about the optimistic call graphs that resulted in different results than one would expect; thus higher precision values or lower recall values. The optimistic call graphs are all supersets of the pessimistic call graphs, which means that recall values should never be lower than for pessimistic call graphs. This is the case in this study, except for the Fullcalendar analysis. This is because the optimistic call graph algorithm for Fullcalendar did not terminate within reasonable time (12h) for the latest Fullcalendar input (due to being a heavy computational process). The dynamic call graph rewriter has been changed a couple of times in order to increase its accuracy. The optimistic results of Fullcalendar that are presented in this study, came about by comparing an old optimistic analysis with an old (less accurate) dynamic call graph rewriter. With a newer optimistic analysis, it is assumed that recall values would be at least of the same value as the recall values of the pessimistic analysis.
Flotr, PDFjs and Markitup all experienced an increase of precision with optimistic analysis. For Flotr only 19 call sites were missing from the optimistic call graph, whereas the pessimistic analysis missed 95 call sites. Precision values are averaged over all call sites that were covered in the dynamic call graph (see section 9.1.1). This entails that call sites that were not present in the static call graph, may lower the total average (as they have an individual precision of 0 per definition). When optimistic analysis can reason about more call sites, it may have a low accuracy for its targets, but the precision will at least be 0 or more. This phenomenon in combination with not finding many additional spurious paths for the call sites that are present in both pessimistic and optimistic analysis, might cause the average precision value to be higher in the optimistic analysis. PDFjs happened to have a lot of callbacks and covered over 100 more call sites with optimistic analysis, which caused the optimistic analysis to result in a higher precision value. Optimistic analysis for Markitup resulted in 8 more covered call sites, which could be the reason for the very slight precision increase ($\sim 0,50\%$) with optimistic analysis.
It is unclear why the extracted models resulted in lower recall values with optimistic analysis for a few programs. This is because the implementations of the static call graph algorithms in the ACG study are not available due to licensing issues.

10.2 Call Graph Edges

Comparing edges rather than averaging precision and recall values over call sites, turns out to negatively affect precision values (as plotted in section 9.2.2). The edges that were compared solely originate from call sites that are present in the static call graphs as well as in the dynamic call graphs. Therefore call sites that were not present in the static analysis, do not negatively impact the precision value, but do negatively impact the recall value. The precision value reasons about the ratio between correctly computed call graph edges and the total computed call graph edges (which have a call site that is verifiable with the dynamic call graph). The recall value reasons about the ratio between correctly computed call graph edges and the total recorded call relationships.
3dmodel and Pacman both have precision and recall values which correspond to the observed averages in the previous paragraph; every edge that is computed and corresponds to a call site that is present in the dynamic call graph, is computed correctly. Besides the set of edges is complete for both pessimistic and optimistic analysis as the recall value is 100%.
Recall values, seem to be close to the observations by averaging over call sites, where pessimistic analysis has a maximum deviation of $\sim$ 6% and optimistic analysis has a maximum deviation of $\sim$ 9%. Precision values however seem to deviate more from the averaged by call site precisions, as some programs deviate more than 20%. This can partially be caused by the divisor in the averaging formula, as call sites that were not computed, negatively affected the average precision. By comparing edges, these call sites have not affected the precision values at all, as they were not taken into account.
The precision and recall values by edges comparison of HTMLEdit turns out to be fairly close to the averaged statistics per call site with the pessimistic analysis. The precision deviates $\sim$ 3%, whilst the recall value deviates $\sim$ 1%. The same goes for pessimistic analysis for Markitup with $\sim$ 5% lower precision and $\sim$ 2% lower recall. For other programs these deviations were higher.
The precision values of comparing edges turn out to be less for all of the subject programs, other than 3dmodel and Pacman. Comparing edges gives insight in the pollution of the computed call graphs; 8 out of 10 subject programs have pessimistic static call graphs (with a call site that is present in the dynamic call graph as well) which consist for more than 30% out of spurious edges.
Computing precision over edges shows that the optimistic analysis is likely to introduce spurious edges. The averaging over call sites does not provide the same insight to this extent; some programs have precision drops of over 50% with optimistic analysis. Recall however always seems to increase, except for Fullcalendar (which has been elaborated in the previous section). PDFjs is the only program with a slight precision increase with optimistic analysis ($\sim 0,40\%$). This could only happen when optimistic analysis entails less spurious edges than the pessimistic analysis in proportion to the total computed edges.

10.3 Threats to Validity

In this section the threats to validity will be discussed. It discusses several issues that might affect the comparison to the previous study in a negative way. Furthermore some issues also have impact on the validity of the call graph edge analysis.

10.3.1 JavaScript Grammar

Both JavaScript call graph algorithms require a parse tree as input. Code analysis at the CWI is often done with the Rascal Meta-Programming Language, which provides a toolset that could ease implementation of such algorithms. Rascal did not provide a JavaScript parser. Therefore creating a Rascal grammar for JavaScript was a prerequisite for the research project. The Rascal grammar seems to parse the given input scripts correctly and has been tested with unit tests with various snippets. Even though the grammar is thoroughly tested, it is possible that deviations of the ECMAScript specification remain. It is known that ambiguities remain and have to be solved (see appendix E). Other than that it has not been unit tested for correctness with large scripts like libraries. This was because developing such tests would take a significant amount of the project its time.
Due to the limited time ($\pm$8 weeks) and almost all subject scripts parsing successfully, the grammar has been considered good enough. I do not consider the impact of these possible deviations to be

significant, but this could explain differences between the output data of this paper and the ACG study. Faults in the grammar, would impact the algorithms, as the parse trees could be incorrect or incomplete. Potentially, this could have caused invalid flows, extra edges in the graphs (due to ambiguities) and undiscovered flows. It is known that faults in the grammar have negatively affected the results from at least one input program (Markitup). This however was known in advance, because an old version of the grammar had to be used.
The static call graphs have been compared to dynamic call graphs that are facilitated with a different parser (Rhino). This is to prevent seeding the same possible faults by the Rascal JavaScript grammar and its ambiguities. It is considered unlikely that the Rhino parser would seed the same errors. The further impact of the grammar remains unknown due to the limitation of time.

10.3.2 Change in the Algorithms

The assumption has been made that the flow graph rules contained an error in rule **R7** (see section 6.1). The algorithm described in the ACG paper does not seem to distinguish global function declarations from in-scope ones (functions that are declared in the scope of another function). The original rule specifies that function declarations are added to the flow graph in the following form:

$$Fun(\pi) \rightarrow Var(\pi)$$

The algorithm specification also considers a lookup function for local variables, which is mentioned as follows:

> We assume a lookup function λ for local variables such that $\lambda(\pi, x)$ for a position π and a name x returns the position of the local variable or parameter declaration (if any) that x binds to at position π [1, p. 5].

In case a global function declaration would also be considered as a `Var` vertex, it would need to be added to the symbol table, and this does not comply to the definition of the lookup function that only considers local variables. Ignoring this would result in not resolving global function declaration f in case f gets called. In case both the vertices in the function declaration and the function call would result in the same vertex, the algorithm could determine where and which global function would be called through the transitive closure.
A successive script, that would essentially implement the same algorithm (according to Schäfer), considers globally declared functions as `Prop` vertices. It was chosen to implement this similarly. This decision could be a reason for deviation between the results of this thesis and those of the ACG study. It is considered unlikely that the ACG study did not implement flow graph rule **R7** similarly, because this would have resulted in call graphs that were less complete. Besides their successive script also indicates that the ACG paper forgot to mention the extension of this rule.

10.3.3 Scoping Consideration

The algorithm specification frequently uses mapping function V, which heavily relies on a symbol table. Mapping function V is used to map expressions to vertices for flow graph creation (e.g. variable references to a variable vertex). In subsection 6.3, the symbol table implementation has been discussed for this study. It is unknown whether the authors of the ACG study implemented scoping similarly. Scoping has often been evaluated by comparing the flow graph output with another student that replicated the same study. In some extraordinary cases they did appear to slightly differ due to scoping interpretations. These differences were never found to affect call graphs, although that could be potentially possible. Furthermore Chrome v38 has been used to examine how overriding works, in case the literature did not tell explicitly. It would not matter if Chrome would deviate from the standards, as the instrumented scripts were ran in Chrome as well.

10.3.4 Optimistic Call Graph Interpretation

In the ACG study, there is little description on the optimistic call graph. An implementation has been made based on the description in the original paper. It has been verified with the jQuery subset script

which was provided as an example in the ACG study. Other than that Schäfer has provided me with a successive call graph analysis program that essentially does the same analysis. This implementation did yield a slightly different code and has been copied to Rascal as well. It turned out to have the same output through the whole period of using the call graph algorithms. It is not clear however whether the ACG study had this exact implementation of this optimistic call graph algorithm. Furthermore the optimistic call graphs were compared to that of another student and they were not found to be deviating.

10.3.5 Disregarding Native Function Calls

Native function calls are disregarded in this study, as discussed in subsection 6.3.4. The original study did take native functions into account. Furthermore it is not known whether the ACG study considered overridden native functions in their dynamic call graph recorder. In this study overridden functions have been taken into account in the dynamic call graph instrumentation. Furthermore the models that were used in the ACG study have also been added to the comparison between this study and the ACG study. They were imported without native call targets. These models make the results more comparable, but do not provide insight in how overridden native functions have been taken into account.

10.3.6 Different Subject Scripts

The ACG study has mentioned which scripts were subjected to the call graph algorithms. Many of the input script were unversioned and the authors did not provide an archive of the exact exercised set of programs. This makes it possible that the subjected scripts in this study deviate. Flotr, Fullcalendar, Markitup and HTMLEdit were the versioned subject scripts. Even for those scripts, replacements or adjustments had to be made (elaborated in appendix C), in order to make them parseable with Rascal. The differences between the subject scripts can be a reason for deviating results between both the ACG study and this thesis.

10.3.7 ACG Statistical Problems

The ACG study elaborates how precision and recall values have been computed in their study. This has been done by a cumulative formula for precision and recall, where precision and recall would be computed on a per call site basis. Afterwards these cumulative values were divided by the amount of call sites that were present in the dynamic call graph. The formula has been implemented for this thesis and has been validated by one of the authors of the ACG study. This formula has been tested on the exported call graphs from the ACG study. The results did not correspond with those of the ACG paper. As the formula and the call graphs are confirmed to be consistent with the original study, the exact impact remains unknown. It is possible that the call graph models that the ACG authors published, are somehow not the same as the ones used in the ACG study. Furthermore it might be possible that the results in the ACG paper are faulty. Finally the formula could be wrongly implemented, which is unlikely as it has been verified with the authors, their paper and another student that implemented the same formula. The computations with the formula have been made for the extracted call graphs and are included in the results of this thesis. This is to guarantee that the results have been computed in the same way as the results of this thesis. This increases comparability between the two studies.

10.3.8 Unsound Dynamic Call Graph

The dynamic call graph sometimes added spurious edges. This could occur when a native function was invoked, followed by entering the scope of an instrumented function. This would be interpreted as a direct invocation. The last call site could be a proxy function, but sometimes entering a function was unrelated to the last call site. This was due to the function scope being entered by a callback from a native event (e.g. `body.onload`). To mitigate these results some measures have been taken, including a post-processing tool for the dynamic call graphs, which removes call relationships invoked

from proxy functions (`eval`, `apply` and `call`). For recording call sites, several checks have been added to mitigate inaccuracies and thus spurious edges. These complicate the call graph recording and could therefore in turn also introduce inaccuracies. For instance, a call stack has been used, which is more dangerous than setting the last invoked function call. This is because connecting a function with a wrong call site on the call stack (thus being one off), could introduce a systematic error that propagates to other call sites in the case of nested function calls. So far only one inaccuracy has been observed, where a function call was considered native, when in fact it was not.
With the applied technique for dynamic call graph recording, the dynamic call graph is not totally sound, but its threats are mitigated to a large extent. It is unlikely that the static call graph would contain the same spurious edges as the dynamic call graph. With that in mind, spurious edges in the dynamic call graph could affect both recall and precision negatively. With the ACG study indicating that their computed call graphs are sound, they can in turn also partially validate dynamic call graphs.
Other invocations than regular invocation (thus other than `f()`) can simply not be accounted for with the current technique. An example of those is getter and setter invocation, because they rely on standard expressions, but come with different semantics. This means that recall could have turned out higher than it should be, as such calls are not recorded. Getters and setters were not statically measured except for PDFjs.
Given the fact that connecting functions to call sites at runtime is a difficult problem, it could be that the ACG study suffered from spurious edges as well, which could be a threat to the validity of their results.

Chapter 11

Conclusion

An existing study of JavaScript call graph construction algorithms has been replicated. A prerequisite JavaScript grammar for the Rascal programming language has been implemented in order to conduct the replication. Flow graph algorithms were implemented in order to track flow of JavaScript programs. These flow graphs served as an input of the replicated call graph algorithms. Two different call graph algorithms were implemented, where both handle interprocedural flow differently. One of them considered all interprocedural flow that it could discover and was called the optimistic call graph algorithm. The other algorithm only reasoned about interprocedural flow when call sites would be locally determinable from the invoked function. This algorithm was marked as the pessimistic call graph algorithm.
The input set of scripts for the call graph algorithms was similar to that of the original study. The resulting call graphs were compared to a dynamic call graph, which consisted of a snapshot of call relationships that were available at runtime. These snapshots were obtained by instrumented versions of the input programs, which would observe the call relationships. A comparison of call graphs between this study and the original study has been done by averaging over computed precision and recall values of call sites. Furthermore recomputations of the extracted call graphs from the original study were also involved in the comparison, as they resulted in different precision and recall values than the study itself. The reason for that is under investigation by the authors, as the comparison algorithm as well as the extracted call graphs have been validated with the authors. For computing these precision and recall values, the dynamic call graphs served as the call graph to compare the static call graphs to. Other than that, another form of comparison has been done where precision and recall values for edges in the call graphs have been computed. This second comparison has not been done in the original study.
A change has been made to the flow graph algorithm in order to increase reach for globally declared functions. Furthermore native JavaScript targets were not taken into consideration in this study, due to the lack of complete models of native JavaScript functions. The computations for precision and recall have been made with these two changes. The data that was gathered by these computations can answer the previously stated research question: "*How accurate are the computed call graphs, using the same precision and recall formulas, given the different environment?*". The pessimistic call graphs had an average precision of 74% and a precision median of 71% when averaged over call sites. The average recall for averaging over call sites was 89%, whilst the median of the recall was 88%. The optimistic call graphs had an average precision of 75% and a precision median of 70% when averaging over call sites. The recall for optimistic analysis when averaged over call sites is 93% in average, whilst the median is 92%. These values indicate that the computations missed little call targets per average call site. It also indicates that spurious edges were not uncommon.
Comparing the complete call relationships (edges) resulted in significant less precision. This indicates that the computed call graphs are more polluted with spurious edges than the values obtained by averaging over call sites suggest. The average precision of edges for pessimistic call graphs was 64%, whilst the median was 55%. The average precision of edges for the optimistic call graphs was 43% and its median was 32%. Recall values of edges did not deviate much from the recall values that were obtained by averaging over call sites. The recall of pessimistic analysis in terms of edges resulted in

an average of 87% and a median of 85%, whilst the optimistic analysis would result in an average of 91% and a median of 92%.
When averaging over call sites, the average precision turned out to be $\sim$ 9% less with pessimistic analysis than the original study. The average recall turned out to be $\sim$ 5% less than the original study with pessimistic analysis. Optimistic analysis turned out to result in $\sim$ 6% less average precision and and $\sim$ 3% less average recall than the original study. When comparing edges (thus the true call relationships) the precision seemed to be disappointing compared to the averaging comparison. The recall values for edges however, tell us that very few call relationships are missed in these computations. This means that the static call graphs contained a significant percentage of correct call relationships, but that these graphs were simply polluted by spurious edges. Precision values were found to be suppressed the most by one of the design decisions for the call graphs. This design decision generalized similarly named functions over multiple objects.
Furthermore static call graphs turned out to be difficult to verify in terms of accuracy and completeness. Call relationships were difficult to obtain accurately by instrumenting existing code without affecting its semantics. Connecting call sites to their actual target turned out to be complicated in such process. The issue of dynamic call graph instrumentation has not been predicted prior to the research and should be a lesson for future call graph analysis. Moreover this research provides more insight in the pollution of the static call graphs.
Contributions of this thesis are the in-depth descriptions on how the static call graph algorithms work over multiple files, what is hard about instrumenting code to obtain dynamic call graphs, how dynamic call graphs come about and how dynamic call graphs that were obtained by instrumented code can be incomplete. Furthermore the averaging over call sites algorithm is described in-depth and the source code that was used is open source. Finally the JavaScript grammar has been a great learning experience in terms of grammars and the JavaScript language. The grammar will be adopted to the Rascal standard library as a result of this preliminary work. Remaining work on the grammar has been documented and added to the appendices of this thesis.

Chapter 12

Future Work

The algorithms are based on the data flow of the ECMAScript 5.1 specification. A new ECMAScript specification will be formally approved in December 2014. Significant changes include asynchronous code, which could make data flow less predictable. Furthermore new syntactic features will be introduced which will also affect the data flow. It would be interesting to see how adaptable and applicable these algorithms are as soon as ECMAScript 6 has matured.
Unfortunately dynamic call graphs are the only way of validating a statically computed call graph, as the ideal call graph is typically incomputable. This means that the dynamic call graphs should be as correct as possible. Instrumentation of JavaScript comes with the problem that it is hard to connect functions to their actual call sites. There is no exact knowledge of the point of invocation of a function. Besides getters and setters are difficult to instrument, especially because there is little static knowledge on whether something is actually a getter or setter. A debugging extension or modified JavaScript engine would have more in depth knowledge of the call stack. Besides such solution could consider dynamically evaluated code and getter and setter invocation. It would be interesting to see how these considerations affect the precision and recall of the static call graphs.
Moreover there is an indication that the call graph computation does contain the most significant part of call relationships, but is polluted with spurious edges. Some form of filtering or alternative constructions should be considered in order to increase the correctness of the computed call graphs. This problem might be considered when implementing this algorithm. It is also a point to improve.
As for the call graph algorithms there are some constructions that could improve the overall accuracy, without changing its fundamentals:

1. Function objects that are dynamically declared through `new Function` are currently not considered as functions. This could be extended so that it would be adopted to the call graph. With jump to declaration functionality one could then jump to the instantiation of the function. This could be achieved by creating an intraprocedural rule for the flow graph for a new function object. Even though we would not know what happens inside those newly declared functions, we could still reason about its call relationships. The following code:

   ```
   1 | var f = new Function("return 1");
   ```

   Would need the same rules as **R7** (see section 6.1):
   Fun(π) $\rightarrow$ **Var**(π) in function scope and **Fun**(π) $\rightarrow$ **Prop**(π) in global scope. Where f is the name on the scope.

2. Consideration of `call` and `apply` invocation could further improve the call graph. These functions both invoke a function object with the given parameters and scope. For example:

   ```
   1 | function f() { console.log("my scope is " + this.toString() +
   2 |   " with arguments " + arguments.toString()); }
   3 | f.apply(document, 1, 2, 3);
   ```

   In the example up here f would be executed where the `this` reference would point to `document`. The arguments of the invocation would be 1, 2 and 3. Even though `call` and `apply` are native

functions, we know what they will do and what they will invoke. Both `call` and `apply` are observed to be used often in libraries (hence the post-processing of dynamic call graphs was necessary). The first parameter of `apply` and `call` represents the scope a function will execute in, which could override the `this` parameter in the function. The other parameters are simply passed through to the invoked function. At this moment these function calls are recognized as calls to native functions, whilst in practice these call sites execute the function on which they are called. The knowledge that these call sites call a native (proxy) function, is probably less interesting than the knowledge of which functions will be called by these call sites.

Furthermore the initial idea was to compare both call graph algorithms to a library (TernJS[1]) from the practical field. This library supports jump to definition and seems to be fairly popular. It is interesting to see if the algorithms are more accurate than the widely used library. They might both incorporate ideas that could be merged to an even more accurate call graph construction. The ACG paper does not consider this library. Therefore further analysis might end up with promising results. Finally with all this theoretical work being done, it would be nice to put the algorithm to use in an IDE. Eclipse provides the so called JavaScript Development Tools (JSDT) to add JavaScript support to Eclipse in the form of plugins. One could start with implementing the call graph algorithms for Eclipse.
Future work that can be done for the Rascal JavaScript grammar is included in appendix E.

[1] https://github.com/marijnh/tern

Bibliography

[1] A. Feldthaus, M. Schaefer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 752–761. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486887

[2] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 6, pp. 685–746, Nov. 2001. [Online]. Available: http://doi.acm.org/10.1145/506315.506316

[3] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Trans. Softw. Eng. Methodol.*, vol. 7, no. 2, pp. 158–191, Apr. 1998. [Online]. Available: http://doi.acm.org/10.1145/279310.279314

[4] P. Young and M. Munro, *A New View of Call Graphs for Visualising Code Structures*, ser. Technical report: School of Engineering and Computer Science. Department of Computer Science, University of Durham, 1997. [Online]. Available: http://books.google.nl/books?id=LfI7PQAACAAJ

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[6] O. Lhoták, "Comparing call graphs," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251542

[7] M. W. Hall and K. Kennedy, "Efficient call graph analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 3, pp. 227–242, Sep. 1992. [Online]. Available: http://doi.acm.org/10.1145/151640.151643

[8] T. Marlowe and B. Ryder, "Properties of data flow frameworks," *Acta Informatica*, vol. 28, no. 2, pp. 121–163, 1990. [Online]. Available: http://dx.doi.org/10.1007/BF01237234

[9] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 848–894, Jul. 1999. [Online]. Available: http://doi.acm.org/10.1145/325478.325519

[10] J. Dean, "Call graph analysis in the presence of higher-order functions," *UW General Exam report*, 1994.

[11] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," *SIGPLAN Not.*, vol. 32, no. 10, pp. 108–124, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/263700.264352

[12] D. Flanagan, *JavaScript: The Definitive Guide*, 6th ed., ser. Definitive Guide Series. O'Reilly Media, Incorporated, 2011. [Online]. Available: http://books.google.nl/books?id=4RChxt67lvwC

[13] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, MA, USA: MIT Press, 1996.

[14] D. Crockford, *JavaScript: The Good Parts. Unearthing the Excellence in JavaScript.* O'Reilly, 2008.

[15] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 1–12.

[16] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *ECOOP 2012 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, J. Noble, Ed. Springer Berlin Heidelberg, 2012, vol. 7313, pp. 435–458. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31057-7_20

[17] S. Ducasse, N. Petton, G. Polito, and D. Cassou, "Semantics and security issues in javascript," *CoRR*, vol. abs/1212.2341, 2012.

[18] A. Osmani, *Learning JavaScript Design Patterns.* O'Reilly Media, Inc., 2012.

[19] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 186–189. [Online]. Available: http://doi.acm.org/10.1145/1133373.1133410

[20] Mozilla. (2014, May) Mozilla variable statements. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

[21] C. Aggarwal and H. Wang, "Graph data management and mining: A survey of algorithms and applications," in *Managing and Mining Graph Data*, ser. Advances in Database Systems, C. C. Aggarwal and H. Wang, Eds. Springer US, 2010, vol. 40, pp. 13–68. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-6045-0_2

[22] CWI. (2014, June) Rascal syntax definition. [Online]. Available: http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Declarations/SyntaxDefinition/SyntaxDefinition.html

[23] ECMA International, *Standard ECMA-262 - ECMAScript Language Specification*, 5th ed., June 2011. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[25] CWI. (2014, July) Rascal visit. [Online]. Available: http://tutor.rascal-mpl.org/Rascal/Rascal.html#/Rascal/Expressions/Visit/Visit.html

[26] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto, "Aojs: Aspect-oriented javascript programming framework for web development," in *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ser. ACP4IS '09. New York, NY, USA: ACM, 2009, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/1509276.1509285

Appendices

Appendix A

Flow Graph Rules

This appendix presents the combination of the intraprocedural and interprocedural flow graph rules, which can be used to increase the understanding of the flow graphs. Furthermore they provide a complete overview.

V		::=	
	|	**Exp**(π)	expression at position π
	|	**Var**(π)	variable declared at position π
	|	**Prop**(f)	property with name f
	|	**Fun**(π)	function declaration/expression at π
	|	**Callee**(π)	callee of function call at position π
	|	**Arg**(π, i)	ith argument of function call at position π
	|	**Parm**(π, i)	ith parameter of function at position π
	|	**Ret**(π)	return value of function at position π
	|	**Res**(π)	result of function call at position π
	|	**Builtin**(f)	native function with the name f
	|	**Unknown**	placeholder for unknown flow

rule #	**node at π**	**edges added when visiting π**
R1	$l = r$	$V(r) \rightarrow V(l)$, $V(r) \rightarrow$ **Exp**(π)
R2	$l \mid\mid r$	$V(l) \rightarrow$ **Exp**(π), $V(r) \rightarrow$ **Exp**(π)
R3	$t \ ? \ l : r$	$V(l) \rightarrow$ **Exp**(π), $V(r) \rightarrow$ **Exp**(π)
R4	$l \ \&\& \ r$	$V(r) \rightarrow$ **Exp**(π)
R5	$\{f\colon e\}$	$V(e_i) \rightarrow$ **Prop**(f_i)
R6	function expression	**Fun**(π) $\rightarrow$ **Exp**(π), if it has a name: **Fun**(π) $\rightarrow$ **Var**(π)
R7	function declaration	if it is in function scope: **Fun**(π) $\rightarrow$ **Var**(π) if it is in global scope: **Fun**(π) $\rightarrow$ **Prop**(π)
R8	$f(\overline{e})$ or $newf(\overline{e})$ or $newf$	$V(f) \rightarrow$ **Callee**(π) $V(e_i) \rightarrow$ **Arg**(π, i), **Res**(π) $\rightarrow$ **Exp**(π)
R9	$r.p(\overline{e})$	**R8** edges, $V(r) \rightarrow$ **Arg**(π, 0)
R10	return e	$V(e) \rightarrow$ **Ret**($\phi(\pi)$)

$$V(t^{\pi}) = \begin{cases} \textbf{Var}(\pi') & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = \pi' \\ \textbf{Prop}(\pi) & \text{if } t \equiv x \text{ and } \lambda(\pi, x) = undefined \\ \textbf{Prop}(f) & \text{if } t \equiv e.f \\ \textbf{Parm}(\phi(\pi'),\ 0) & \text{if } t \equiv this \\ \textbf{Parm}(\pi',\ i) & \text{if } \lambda(\pi, t) \textit{ is a parameter of function at position } \pi' \\ \textbf{Exp}(\pi) & \text{otherwise} \end{cases} \tag{A.1}$$

Appendix B

Flow and Call Graph Examples

This appendix displays the flow graph of an input program (jQuery subset). The two textual forms of both pessimistic and optimistic call graphs are also added. This jQuery subset script originates from the ACG study and can be used for understanding the flow graph rules and the differences between the pessimistic and optimistic call graph. Note that native functions have been disregarded for this example.
Input program:

```
1  (function() {
2    function jQuery(n) {
3      var res = Object.create(jQuery.fn);
4      var elts = document.getElementsByTagName(n);
5      for(var i=0;i<elts.length;++i)
6        res[i] = elts[i];
7      res.length = elts.length;
8      return res;
9    }
10
11   jQuery.fn = {
12     extend: function ext(obj) {
13       for(var p in obj)
14           jQuery.fn[p] = obj[p];
15       }
16   };
17
18   jQuery.fn.extend({
19       each: function(cb) {
20         for(var i=0;i<this.length;++i)
21           cb(this[i], i);
22       }
23   });
24
25   window.jQuery = jQuery;
26 })();
27 (function($) {
28   $.fn.highlightAlt = function(c) {
29   this.each(function(elt) {
30     for(var i=1;i<elt.children.length;i+=2)
31         elt.children[i].style.backgroundColor = c;
32       });
33   }
34
```

```
35      window.highlightAltRows = function() {
36        $('tbody').highlightAlt('#A9D0F5');
37      };
38    })(jQuery);
```

Flow Graph

Expr(jquerysubset.js@11:236-327) → Expr(jquerysubset.js@11:224-327)
Expr(jquerysubset.js@11:236-327) → Prop(fn)
Expr(jquerysubset.js@12:248-324) → Prop(extend)
Expr(jquerysubset.js@14:311-317) → Expr(jquerysubset.js@14:296-308)
Expr(jquerysubset.js@14:311-317) → Expr(jquerysubset.js@14:296-317)
Expr(jquerysubset.js@18:348-437) → Arg(jquerysubset.js@18:331-438, 1)
Expr(jquerysubset.js@19:359-434) → Prop(each)
Expr(jquerysubset.js@1:1-467) → Callee(jquerysubset.js@1:0-470)
Expr(jquerysubset.js@20:388-389) → Expr(jquerysubset.js@20:386-389)
Expr(jquerysubset.js@20:388-389) → Var(i, jquerysubset.js@20:386-387)
Expr(jquerysubset.js@21:417-424) → Arg(jquerysubset.js@21:414-428, 1)
Expr(jquerysubset.js@27:473-735) → Callee(jquerysubset.js@27:472-744)
Expr(jquerysubset.js@28:508-650) → Expr(jquerysubset.js@28:488-650)
Expr(jquerysubset.js@28:508-650) → Prop(highlightAlt)
Expr(jquerysubset.js@29:533-645) → Arg(jquerysubset.js@29:523-646, 1)
Expr(jquerysubset.js@30:561-562) → Expr(jquerysubset.js@30:559-562)
Expr(jquerysubset.js@30:561-562) → Var(i, jquerysubset.js@30:559-560)
Expr(jquerysubset.js@35:679-732) → Expr(jquerysubset.js@35:653-732)
Expr(jquerysubset.js@35:679-732) → Prop(highlightAltRows)
Expr(jquerysubset.js@36:694-704) → Arg(jquerysubset.js@36:694-728, 0)
Expr(jquerysubset.js@36:696-703) → Arg(jquerysubset.js@36:694-704, 1)
Expr(jquerysubset.js@36:718-727) → Arg(jquerysubset.js@36:694-728, 1)
Expr(jquerysubset.js@3:48-72) → Expr(jquerysubset.js@3:42-72)
Expr(jquerysubset.js@3:48-72) → Var(res, jquerysubset.js@3:42-45)
Expr(jquerysubset.js@4:87-119) → Expr(jquerysubset.js@4:80-119)
Expr(jquerysubset.js@4:87-119) → Var(elts, jquerysubset.js@4:80-84)
Expr(jquerysubset.js@5:133-134) → Expr(jquerysubset.js@5:131-134)
Expr(jquerysubset.js@5:133-134) → Var(i, jquerysubset.js@5:131-132)
Expr(jquerysubset.js@6:166-173) → Expr(jquerysubset.js@6:157-163)
Expr(jquerysubset.js@6:166-173) → Expr(jquerysubset.js@6:157-173)
Func(jquerysubset.js@12:248-324) → Expr(jquerysubset.js@12:248-324)
Func(jquerysubset.js@12:248-324) → Var(ext, jquerysubset.js@12:248-324)
Func(jquerysubset.js@19:359-434) → Expr(jquerysubset.js@19:359-434)
Func(jquerysubset.js@1:1-467) → Expr(jquerysubset.js@1:1-467)
Func(jquerysubset.js@27:473-735) → Expr(jquerysubset.js@27:473-735)
Func(jquerysubset.js@28:508-650) → Expr(jquerysubset.js@28:508-650)
Func(jquerysubset.js@29:533-645) → Expr(jquerysubset.js@29:533-645)
Func(jquerysubset.js@2:15-220) → Var(jQuery, jquerysubset.js@2:15-220)
Func(jquerysubset.js@35:679-732) → Expr(jquerysubset.js@35:679-732)
Parm(jquerysubset.js@19:359-434, 1) → Callee(jquerysubset.js@21:414-428)
Parm(jquerysubset.js@27:473-735, 1) → Callee(jquerysubset.js@36:694-704)
Parm(jquerysubset.js@28:508-650, 0) → Arg(jquerysubset.js@29:523-646, 0)
Parm(jquerysubset.js@28:508-650, 1) → Expr(jquerysubset.js@31:597-638)
Parm(jquerysubset.js@28:508-650, 1) → Prop(backgroundColor)
Parm(jquerysubset.js@2:15-220, 1) → Arg(jquerysubset.js@4:87-119, 1)
Prop(Object) → Arg(jquerysubset.js@3:48-72, 0)
Prop(create) → Callee(jquerysubset.js@3:48-72)

Prop(document) → Arg(jquerysubset.js@4:87-119, 0)
Prop(each) → Callee(jquerysubset.js@29:523-646)
Prop(extend) → Callee(jquerysubset.js@18:331-438)
Prop(fn) → Arg(jquerysubset.js@18:331-438, 0)
Prop(fn) → Arg(jquerysubset.js@3:48-72, 1)
Prop(getElementsByTagName) → Callee(jquerysubset.js@4:87-119)
Prop(highlightAlt) → Callee(jquerysubset.js@36:694-728)
Prop(jQuery) → Arg(jquerysubset.js@27:472-744, 1)
Prop(length) → Expr(jquerysubset.js@7:177-201)
Prop(length) → Prop(length)
Res(jquerysubset.js@18:331-438) → Expr(jquerysubset.js@18:331-438)
Res(jquerysubset.js@1:0-470) → Expr(jquerysubset.js@1:0-470)
Res(jquerysubset.js@21:414-428) → Expr(jquerysubset.js@21:414-428)
Res(jquerysubset.js@27:472-744) → Expr(jquerysubset.js@27:472-744)
Res(jquerysubset.js@29:523-646) → Expr(jquerysubset.js@29:523-646)
Res(jquerysubset.js@36:694-704) → Expr(jquerysubset.js@36:694-704)
Res(jquerysubset.js@36:694-728) → Expr(jquerysubset.js@36:694-728)
Res(jquerysubset.js@3:48-72) → Expr(jquerysubset.js@3:48-72)
Res(jquerysubset.js@4:87-119) → Expr(jquerysubset.js@4:87-119)
Var(i, jquerysubset.js@20:386-387) → Arg(jquerysubset.js@21:414-428, 2)
Var(jQuery, jquerysubset.js@2:15-220) → Expr(jquerysubset.js@25:442-464)
Var(jQuery, jquerysubset.js@2:15-220) → Prop(jQuery)
Var(res, jquerysubset.js@3:42-45) → Ret(jquerysubset.js@2:15-220)

Pessimistic Call Graph

Callee(jquerysubset.js@18:331-438) → Func(jquerysubset.js@12:248-324)
Callee(jquerysubset.js@1:0-470) → Func(jquerysubset.js@1:1-467)
Callee(jquerysubset.js@27:472-745) → Func(jquerysubset.js@27:473-736)
Callee(jquerysubset.js@29:523-646) → Func(jquerysubset.js@19:359-434)
Callee(jquerysubset.js@36:695-705) → Func(jquerysubset.js@2:15-220)
Callee(jquerysubset.js@36:695-729) → Func(jquerysubset.js@28:508-650)

Optimistic Call Graph

Callee(jquerysubset.js@18:331-438) → Func(jquerysubset.js@12:248-324)
Callee(jquerysubset.js@1:0-470) → Func(jquerysubset.js@1:1-467)
Callee(jquerysubset.js@21:414-428) → Func(jquerysubset.js@29:533-645)
Callee(jquerysubset.js@27:472-745) → Func(jquerysubset.js@27:473-736)
Callee(jquerysubset.js@29:523-646) → Func(jquerysubset.js@19:359-434)
Callee(jquerysubset.js@36:695-705) → Func(jquerysubset.js@2:15-220)
Callee(jquerysubset.js@36:695-729) → Func(jquerysubset.js@28:508-650)

The bold line in the optimistic call graph represents the discovered call relationship that is not found in the pessimistic analysis. As we can see this call yields a callback to a function that is passed through as an argument in line #29.

Appendix C

Data Gathering Details

All input data for the algorithms and their output are gathered on github, on the following URL: https://github.com/abort/javascript-thesis. The comment stripping tool is also placed on github. The rewriter tool necessary is placed on github and is called *Rewriter.java*. It runs with one argument, which is the path of the directory to rewrite.

Preparation Steps

The following steps describe how the call graphs have been emerged for each program:

3dmodel:

1. remove google analytics code from Javascript 3D Model Viewer.html
2. extract embedded javascript code (body onload and select onchange) to index.js
3. embed index.js in the appropriate place
4. run both static call graph algorithms on the root directory and store the results
5. rewrite the root directory and include the instrumentation file in Javascript 3D Model Viewer.html
6. run Javascript 3D Model Viewer.html for dynamic call graph recording
7. do some manual invocation to increase call graph coverage:

```
1  rotate_x(1,1,1)
2  rotate_y(1,1,1)
3  rotate_z(1,1,1)
4  rotate_solid(1, 1, 1, world.solid[0])
5  rotate_solid_x(world.solid[0], 1, world.solid[0])
6  rotate_solid_y(world.solid[0], 1, world.solid[0])
7  rotate_solid_z(world.solid[0], 1, world.solid[0])
8  translate_solid_direction(1, 1, world.solid[0])
```

beslimed:

1. remove google analytics code from BeSlimed - Mootools Game.html
2. run both static call graph algorithms on the root directory and store the results
3. rewrite the root directory and include the instrumentation file in BeSlimed - Mootools Game.html
4. run BeSlimed - Mootools Game.html

coolclock:

1. replace jquery with a non-minimal version
2. strip the comments of jquery with the comment stripping tool
3. create an index.html that embeds all three demo html files by iframes

4. run both static call graph algorithms on the root directory and store the results
5. rewrite the root directory and include the instrumentation file in the index.html
6. prefix each expression starting with `_wrap_` (instrumentation code) with `window.parent` so the parents instrumentation code will be called
7. run the index.html for dynamic call graph recording

htmledit:

1. replace jquery minified with full version
2. strip the comments of jquery with the comment stripping tool
3. strip comments from htmlbox.full.js with the comment stripping tool
4. replace usage (script references) of htmlbox.min.js to htmlbox.full.js in demo-full.js
5. replace usage (script references) of jquery.min.js to the jquery full version
6. extract demo-full.html embedded javascript to demo-full.js and refer to it
7. strip the comments of demo-full.js with the comment stripping tool
8. run both static call graph algorithms on the root directory and store the results
9. rewrite the root directory and include the instrumentation file in demo-full.html
10. run demo-full.html for dynamic call graph recording

pacman:

1. move the first javascript code block of index.html to main.js
2. move the second javascript block to main2.js
3. include those files in the appropriate places (embed script)
4. delete both js files that are not for chrome (pacman10-hp.2.js and pacman10-hp.js)
5. run both static call graph algorithms on the root directory and store the results
6. rewrite the root directory and include the instrumentation file in index.html
7. run index.html for dynamic call graph recording

markitup:

1. download the jquery version (non-compressed/non-minified) that is refered to in index.js
2. strip the comments from the downloaded jquery version
3. emded the downloaded jquery version instead of the external one with the script tag
4. extract javascript from the index.html to index.js
5. embed index.js in index.html with the script tag
6. change the `functionDeclaration` to
   `"function" Id name "(" {Id ","}* parameters ")"`
   `Block implementation NoNL ZeroOrMoreNewLines NoNL () !>> [\n];`
   (this will provide inaccuracies, but is necessary to have markitup parsing). Make sure to undo this afterwards, as this change will also negatively affect other analysis
7. run both static call graph algorithms on the root directory and store the results
8. rewrite the root directory and include the instrumentation file in index.html
9. run index.html for dynamic call graph recording

flotr:

1. replace prototype with 1.6.0.3 to prevent parsing issues
2. strip the comments from both the replaced prototype with the comments stripping tool
3. create full-flotr.html with iframes with all example html files
4. extract embedded javascript for each example html file to <html_filename>.js
5. embed the extracted javascripts in each html file with the script tag
6. replace canvas2image.js with the one from this url https://github.com/hongru/canvas2image (for parsing issues)

7. strip comments from canvas2image.js with the comment stripping tool
8. run both static call graph algorithms on the root directory and store the results
9. rewrite the root directory and include the instrumentation file in demo-full.html
10. prefix each expression starting with `_wrap_` (instrumentation code) with `window.parent` so the parents instrumentation code will be called
11. run demo-full.html for dynamic call graph recording

fullcalendar:

1. replace jquery and jquery ui with the same versions, but unminified
2. strip the comments from both the replaced versions with the comments stripping tool
3. include those files in the appropriate places (embed script) in theme.html
4. extract the theme.html embedded javascript to theme-main.js
5. embed theme-main.js in theme.html with the script tag
6. run both static call graph algorithms on the root directory and store the results
7. rewrite and include the instrumentation file
8. run theme.html for dynamic call graph recording

pdfjs:

1. build / make a final pdfjs file (see the readme of pdfjs)
2. run both static call graph algorithms on the root directory and store the results
3. rewrite pdf.js
4. include the instrumentation file in viewer.html
5. run viewer.html for dynamic call graph recording

Dynamic Call Graph Recording

After rewriting each JavaScript program and including the instrumentation, dynamic call graphs could be recorded. This would be done by manually stressing each JavaScript program trying to invoke as many functions as possible. Afterwards the `_wrap_postProcess` function will post-process data and add calls to the *native* vertex, as well as output the dynamic call graph in JSON format. This format can be imported in Rascal for the precision and recall formula.
The `_wrap_calculateFunctionCoverage` calculates the total function coverage of the loaded rewritten scripts. There is a different version of this function that supports a parameter to disregard one or more scripts (e.g. jQuery).
After recording each dynamic call graph, it has been checked and beautified with JSONLint (due to Rascal not parsing the raw outputted JSON) and stored to a JSON file.

Post-Processing Data

After recording the dynamic call graph, it should be post-processed. This is done with the same project as the Rewriter, but with the *PostProcess.java* file instead. This main method takes two arguments, the first being the original input folder and the second being the recorded JSON file. It will output a new text file with a post-processed call graph.

Static Call Graph Construction

Each JavaScript program is ready to be analyzed, just before the rewriting process. By importing the `CallGraphComparison` module in Rascal, we can create a call graph of a folder and store it for later analysis. The `storePessimisticCallGraphForStatistics` and `storeOptimisticCallGraphForStatistics` functions create call graphs for a given path and store them in txt format.

Statistical Analysis

After storing both the post-processed dynamic call graph and the static call graphs, they are ready to be compared. The `importCallGraphFromPlainText` function imports a call graph in txt format. After importing the `calculatePerCallSite` function is used to calculate average precision and recall over call sites. Afterwards the `calculateStatistics` function can be used to calculate precision and recall for edges.
In order to import call graphs in JSON format, the function `importCallGraphFromJSON` can be used to import call graphs that are stored in JSON format, which map functions to call sites. Call sites that are mapped to functions in JSON can be imported with `importReversedCallGraphFromJSON`, which has been done for the exported call graph models that were published by the ACG authors.

Appendix D

Gathered Statistics

This appendix presents all gathered call graph values. Note that PCG represents pessimistic call graphs, whereas OCG signifies the optimistic call graphs.
The following table contains the gathered call graph data[1]:

	PCG Precision (edges)	OCG Precision (edges)	PCG Recall (edges)	OCG Recall (edges)	PCG Average Precision (targets)	OCG Average Precision (targets)	PCG Average Recall (targets)	OCG Average Recall (targets)
3dmodel	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
Beslimed	41,42%	14,32%	81,38%	91,81%	64,34%	64,15%	81,68%	92,22%
Coolclock	48,24%	32,06%	89,02%	92,51%	62,24%	62,21%	94,24%	96,89%
Flotr	54,63%	27,50%	71,71%	85,08%	61,06%	70,68%	72,21%	86,19%
Fullcalendar	53,28%	2,72%	86,77%	84,70%	71,43%	67,28%	90,28%	88,40%
HTMLEdit	68,14%	40,52%	85,44%	85,71%	70,68%	69,24%	86,79%	86,92%
Markitup	69,63%	29,45%	85,14%	86,63%	74,41%	75,06%	87,46%	89,39%
PacMan	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
PDFjs	39,85%	40,22%	84,93%	91,89%	64,62%	70,45%	87,64%	94,82%

The gathered call graph data of the thesis

The following table contains the comparison data between the thesis and the study by Feldthaus et al.:

	This study's PCG Average Precision (targets)	Original study's PCG Average Precision (targets)	This study's OCG Average Precision (targets)	Original study's OCG Average Precision (targets)	This study's PCG Average Average Recall (targets)	Original study's PCG Average Recall (targets)	This study's OCG Average Recall (targets)	Original study's OCG Average Recall (targets)	Online Models PCG Average Precision (targets, no natives)	Online Models PCG Average Recall (targets, no natives)	Online Models OCG Average Precision (targets, no natives)	Online Models OCG Average Recall (targets, no natives)
3dmodel	100,00%	93,33%	100,00%	93,33%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
Beslimed	64,34%	77,79%	64,15%	69,19%	81,68%	84,37%	92,22%	85,18%	64,99%	83,05%	62,11%	90,69%
Coolclock	62,24%	89,08%	62,21%	88,28%	94,24%	98,47%	96,89%	99,69%	82,62%	97,22%	77,35%	93,05%
Flotr	61,06%	71,70%	70,68%	66,33%	72,21%	83,43%	86,19%	92,83%	61,05%	73,01%	63,34%	88,51%
Fullcalendar	71,43%	83,58%	67,28%	81,78%	90,28%	93,16%	88,40%	94,04%	73,48%	89,78%	68,61%	86,53%
HTMLEdit	70,68%	80,91%	69,24%	79,76%	86,79%	94,26%	86,92%	94,35%	75,34%	92,03%	71,62%	89,84%
Markitup	74,41%	82,41%	75,06%	81,59%	87,46%	93,96%	89,39%	95,53%	54,59%	66,84%	53,00%	66,17%
PacMan	100,00%	93,94%	100,00%	93,94%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	99,47%	99,47%
PDFjs	64,62%	76,82%	70,45%	76,59%	87,64%	99,57%	94,82%	99,91%	77,04%	99,39%	70,73%	93,26%

Comparison table of the call graph data by this thesis and the study by Feldthaus et al.

[1] All thesis data is available at https://github.com/abort/javascript-thesis

Appendix E

Future Work for JS Grammar

This appendix gives insight in the work that has to be done for the JavaScript grammar in Rascal.

The following should still be done on the JavaScript grammar:

- Support for EcmaScript versions higher than 5.1. Even though 5.1 seems to cover a lot of scripts, CWI might consider to support newer and upcoming ECMAScript versions as well.
- Disambiguation of regular expressions: At this moment regular expressions might parse with ambiguities and should have a unique production.
- Disambiguation of multiline strings: At this moment multiline strings appear to have the same problem as regular expressions and should also have a single production rather than multiple.
- Disambiguation of hex literals: At this moment hex literals have the same problem as multiline strings and regular expressions.
- Function closures can be falsely identified as an argument to the preceding declared function. The following snippet is therefore ambiguous:

```
1 function f() { }
2 (function g() { })();
```

  Function f should be a declaration, whereas g is an immediately-invoked function. The ambiguity does contain the correct interpretation, but also contains a production where it sees the entire syntax as an invocation without arguments of an invocation with arguments (where the arguments is the closure of g). This description is cryptic, but the function `parseAndView` provided in the grammar should provide insight in this.
- Multiline and single line comments seem to cause parsing errors. This is probably due to the introduction of alternative layout (called `NoNL`) between syntax, which was necessary to be added for alternative automatic semicolon insertion.
- Multi-variable assignments (assignments in the form of `x = 1, y = 2, z = 3;`) may cause parse errors, when operators precede the assignment symbol. An example of these would be `x -= y, y *= 2`. These simply have not been implemented yet. Multi-variable declarations already have been implemented.
- Create large code tests to validate whether parse trees are correct. So far only small snippets have been tested to validate the parse trees. The assumption has been made that parse trees for longer scripts would be fine, as the small snippets did turn out correct. This assumption has to be validated/tested to a larger extent.
- A feature of JavaScript parsers is that when input can not be parsed, the parser will insert a semicolon at the end of the input stream [23, p. 27]. This function still has to be implemented for the grammar.

- The set of test scripts should be complemented for these ambiguities, but also for uncommon statements like the `with` statement.
- The coercion prefix operator should be supported (`!!x`). This operator would coerce a certain truthy or falsy variable to a boolean type. Currently this is interpreted as a double negation instead.
- Line terminators other than line feed (lf) are still unsupported by the grammar. A complete list of line terminators is included in the ECMAScript specifications [23, p. 15].
- Blocks and their nested statements require refactoring to make it more maintainable. They are hard to read and extend at this very moment.
- Introduction of abstract syntax trees. It would be nice to have ASTs for JavaScript so that layers in the tree can be left out. It would also eliminate no distinction between the differently terminated statements.