Explain Bug Provenance with Trace Witness

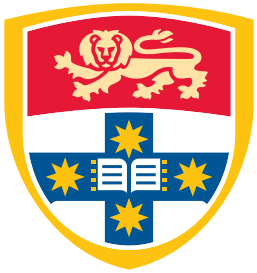
JIXIANG SHEN
SID: 481775250

Supervisor: Dr. Xi Wu & A/Prof. Bernhard Scholz

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Computer Science and Technology(Advanced)(Honours)

School of Computer Science
The University of Sydney
Australia

11 November 2019



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

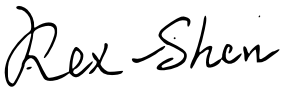
I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Jixiang Shen

Signature:  **Date**: 11/11/2019

Abstract

Bug finding is an established technique to improve the productivity of software engineers. Without running the code, software anomalies can be identified by scrutinising the source code algorithmically. However, current bug finders lack to explain why bugs are found due to technical reasons such as flow-insensitive static program analyses. Hence, there is a need to explain a reported bug by bug finders to the end-users.

In this work, we explain the root cause of a bug by establishing a program trace. We introduce a new approach that overcomes the undecidability of computing a trace. The approach uses a flow-sensitive/approximated semantics of the program. By weakening the semantics of the program, the path computation for bug provenance becomes computable. For this purpose, we introduce a new computational model for the construction of paths, i.e., the Constant Copy Machine. With the means of the Constant Copy Machine, the bug provenance can be solved for complex program semantics in polynomial time.

The real-world application of Trace Witness problem is also discussed in the paper. With an additional gadget, an Object-Oriented language can be easily mapped to the simplified machinery and thus solved by our algorithm. Null Pointer Exception as a case study is presented to evaluate the effectiveness of our approach. We also extend the experiment to large-scale Java open-source programs to visualise and validate Null Pointer Exceptions reported by bug checker. The result invalidates around 80% bugs so that our algorithm can be found efficient in the real-world application.

Acknowledgements

Upon completing this honor thesis, I would like to give my thanks to my two amazing supervisors Dr Xi Wu and A/Prof. Bernhard Scholz, for your both relentless guidance and support thoughout every stage of the research in the last one year. I would also like to thank everyone in the PLANG research group Brody, David, Abdul, Martin, Gavin and all, for the shared camaraderie as well as all the constructive advices for me at the early state of this research.

CONTENTS

List of Figures

CHAPTER 1

Introduction

Software failure could lead to great consequences, and devising an efficient bug-finding tool is imperative for a software to hold secure. Bug finding tools have matured so that they are widely used in industry (Wagner et al., 2005). However, state of the art bug-finding tools that employ static program analysis, mainly focus on the discovery of bugs without providing a comprehensive explanation for their existence.

To improve the utility of bug-finding tools, comprehensive explanations for reported bugs are of paramount importance. Bug finding tools have developed some notions of *provenance* that explain the root cause of a reported bug. However, the root cause of a bug is in most cases spatially and temporally removed from the reported line number of the bug itself. For a software engineer, it may not be immediately intuitive to understand the true existence of a reported bug without understanding the connection between the root cause of a bug and reported line number. Hence, finding the provenance of a bug exposes this causal connection and is paramount for the end-user of a bug-finding tool.

State of the art tools provides weak notions of provenance, such as recording partial paths and heuristics to expose the root cause of the bugs. However, we can summarise that most of the techniques are ad-hoc since the problem is inherently hard and undecidable if definite explanations are sought after (Shepherdson and Sturgis, 1963).

The aim of this work is to find the causal connection between the root cause of a bug and the reported line number. We use an abstract interpretation framework to build the scaffolding for the provenance construction. The program state is abstracted using a Constant Copy Machine. This is a decidable machine which is able to compute the program path. We demonstrate a use case for our new path provenance approach: for this purpose, we assume an Andersen-style abstraction (Andersen, 1994) for object-oriented

programs. The abstraction also considers the data flow information between program state for that it preserves the order of instructions for a program path. A reported bug in such abstraction can be scrutinised whether a program path exists from the start node to the reported path. Such path gives us an explanation to the end-user. We call such a program path an (abstract) "Trace Witness" that resembles the provenance of a bug. The trace is enriched with the abstract semantics explaining the root-cause of a bug.

Besides explaining the bug, the Trace Witness also can be used to sharpen a flow-insensitive analysis result. For example, if a bug is reported, but no Trace Witness can be produced, we can say that the analysis that produced the bug report introduced a false positive. The existence of a trace is a necessary but not sufficient condition for the existence of the bug. Hence, the Trace Witness can also be seen as a post-mortem analysis (Manevich et al., 2004) that introduces flow-sensitivity post-mortem. We have implemented our new bug provenance approach in Soufflé, which is a modern Datalog synthesis framework. Out of Datalog specifications, efficient parallel C++ code can be generated.

1.1 Contribution

The main contribution of this work are as follows

- **Constant Copy Machine:** We produce a trace-witness based on a novel translation scheme reducing the problem to a constant-copy machine that model constant and copy assignments simulating the input program.
- **Single Static Assignment Form in Datalog:** Single Static Assignment form is used to enable a flow-sensitive analysis. In this paper, we introduce an efficient Single Static Assignment implementation in Datalog.
- **Trace Witness Algorithm in Datalog:** A Trace Witness algorithm is proposed and implemented in Datalog to generate the Trace Witness for the constant-copy machine, leveraging Floyd-Warshall Algorithm in Shortest Path problem.

1.2 Outline

This thesis will start by introducing background knowledge such as Datalog and Soufflé in Chapter 2, highlighting the syntax of Datalog. The related work will be discussed in Chapter 3, explore the existing techniques used in Points-To analysis, trace reconstruction and shortest path problem.

Our approach for explaining bug provenance with Trace Witness are introduced and discussed in Chapter 4, 5 and 6. In Chapter 4, we propose the syntax and semantics of our Constant-Copy-Machine, followed by a formal problem definition. Chapter 5 and Chapter 6 investigate the Single Static Assignment Form with detailed implementation in Datalog and Path Construction via our Trace Witness algorithm in Datalog, respectively. We apply our approach to set up benchmarks in Chapter 7, followed by a conclusion and future work in Chapter 8.

CHAPTER 2

Background

2.1 Datalog & Soufflé

Traditionally, static program analysis is troublesome and expensive to develop because the choice of imperative language is facing challenges of input semantics modelling, the precise abstraction of analysis and lengthy code crafting. *Datalog* as a domain-specific language is prospected to develop static analysis in the form of a logical specification to overcome these challenges.(Dawson et al., 1996)

In specific, Smaragdakis and Balatsouras has surveyed a broad range of pointer analysis techniques specified in Datalog.(Smaragdakis and Balatsouras, 2015) In order to realise the analysis in Datalog, the input program must be represented as relations that encodes its elements (facts). A Datalog program is then executed by repeated applying monotonic logical inferences (rules) to produce more facts until reached a fixpoint so that it is strictly declarative: the order of execution does not affect the final result. The use of Datalog allows concise expression of the precision aspect of pointer analysis, yet not need to treat the implement of algorithm separately.

Although tempting as its simplicity and declarativity, Datalog is not widely adopted in practice due to its incompleteness and inefficiency. The traditional evaluation strategy of the datalog program is through naïve evaluation that computes all the knowledge based on the current iteration, which also includes all the previous knowledge that is generated and inevitably becomes too expensive to compute as the relation sets grow.(Bancilhon, 1986) A recent advance has been made to address this issue of Datalog in (Scholz et al., 2016) Introduced a novel technique to execute Datalog programs by adapting semi-naïve evaluation to synthesise Datalog specification to a low-level imperative parallel code such as C++ to achieve an efficient and scalable result. The experiment is also conducted on analysing millions of variables under a minute to support the claim.

In the scope of trace reconstruction problem, Datalog has attracted more attention than the other imperative languages not only because its simplicity of exhibiting static analysis implementation but also its well-handling in recursion when answering queries for a graph problem.

2.1.1 Datalog Syntax

Datalog extends the syntax style from Prolog, expressing the computation in the form of first-order predicate logic, known as Horn clause.(Horn, 1951) For a Horn clause in the form of $u \Leftarrow p \wedge q \wedge \cdots \wedge t$, it can be interpreted as predicate u holds if $p \wedge q \wedge \cdots \wedge t$ is true, where u is the head of the clause and the conditions are the body.

Relations are a set of ordered tuples that bound to its declaration. Generation of relations in Datalog can be categorised into two styles of logic expression, one known as Extensional Database(EDB), and the other as Intensional Database(IDB). EDB acts as input for a Datalog program that only consists of facts without rules, whereas IDB is on the contrary. Facts strictly speaking are a subset of rules that generate the relation without condition, normally are imported from files. A rule in Datalog is a horn clause whose head is a relation to be generated, and the body is a composite of other relations as a condition for the generation.

2.1.2 Soufflé Syntax

The language in Soufflé introduced a few more features to Datalog, including

- **Type System:** Soufflé enforces type over all attributes in a relation to enable early error detection. The types can be symbol, number or primitive type that is a user defined subset under either symbol or number type. For example, $.number\_typeNode$ declares a primitive type that can be deduced to a number.
- **Relation:** Soufflé requires declaration of each relation used in the program. For example, $.declEdge(n : Node, m : Node)$ declares the realtion of $Edge$ with two attributes of $Node$ type.
- **Rules:** Soufflé enables flexible logic expression in the body of a rule, where , is used between conditions as conjunction and ; as disjunction. Soufflé also includes other useful features such

as negation and functors including arithmetic operations. For example, the rule

$$OtherEdge(n, m) :- Edge(n, m), !Start(n)$$

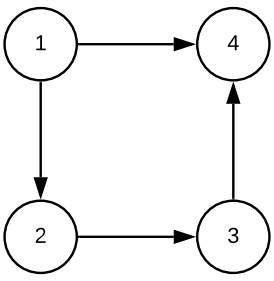
indicates $OtherEdge$ will includes all the $Edge$ tuple excpet the edges are from $Start$ node.

2.1.3 Datalog Example

Take the following Datalog program for example

```
1  .decl Edge(n: Node, m: Node)
2  .decl Reachable(n: Node, m: Node)
3
4  Edge(1, 2).
5  Edge(1, 4).
6  Edge(2, 3).
7  Edge(3, 4).
8
9
10 Reachable(n, m):-
11     Edge(n, m).
12
13 Reachable(n, o):-
14     Reachable(n, m),
15     Edge(m, o).
```

This program declares two relations $Edge$ and $Reachble$, $Edge(n : Node, m : Node)$ denotes an edge exists from node n to node m, and $Reachble(n : Node, m : Node)$ as node m is reachable from node n. $Edge$ is fully genereted from facts (i.e. IDB), representing the graph below



The first and second rules for $Reachable$ can be read as

$$\exists\, n, m : Edge(n, m) \Rightarrow Reachable(n, m)$$

and

$$\exists\, n, m, o : Reachable(n, m) \wedge Edge(m, o) \Rightarrow Reachable(n, o)$$

Informally sepeaking, a $Reachable(n, m)$ will be generated for every $Edge(n, m)$ and $Reachable(n, o)$ will be generated for every $Reachable(n, m)$ and $Edge(m, o)$. The second rule indicates $Reachable$ is a recursive relation, by which means the relation will keep propogating itself until no new tuple can be generated. At the end of the program, $Reachable$ should hold tuples of (1, 2), (1, 4), (2, 3), (3, 4) and (2, 4).

CHAPTER 3

Related Work

This research is proposing an automated trace witness discovery approach via program synthesis and graph abstraction for a high-level Object-Oriented language. At the current stage of trace witness field, the main existing gaps are

(1) Full trace is not available due to the overhead of recording and transmitting.
(2) Execution trace may be long and contain irrelevant information
(3) No general solution exists for all types of errors
(4) The materialisation of concrete traces is difficult

There have been several approaches proposed aiming to bridge the gaps above, such as *Postmortem Symbolic Execution(PSE)* formulating the problem as backward data flow analysis and can be solved in polynomial time.(Manevich et al., 2004) Nevertheless, drawbacks exist with these methods, and this literature review will bring up these approaches in the perspective of their strengths and weaknesses. Moreover, supplemental techniques and tricks that are related to this research area will be presented and compared against.

3.1 Points-to Analysis

The trace construction problem we are facing can be fundamentally phrased as bug condition base on the program heap in an object-oriented language like Java. Modelling these underlying semantics is vital for finding out the trace because the choice of branches is merely depended on the program state at the program point. However, unlike stack and static data, heap data cannot be abstracted exactly to a fixed set of variables due to the halting problem addressed by Landi.(Landi, 1992)

Points-to analysis or *pointer analysis* is a subclass of static program analysis delivering a solution abstracts the heap information of a program as a static model, i.e. which pointer can point to which object. Since such analysis is an approximation due to the undecidable behaviour of a general model for a program as stated by Ramalingam, the lack of soundness is inevitable for pointer analysis.(Ramalingam, 1994) In order to return a theoretically feasible trace, it is crucial that a fairly precise approach of points-to analysis to be taken, yet a trade-off between precision and scalability must also be evaluated. The area of pointer analysis has been thoroughly studied in various articles, and this section will review some schemes and techniques mentioned in these papers.

3.1.1 Andersen-style Analysis

A simple but useful points-to analysis algorithm is proposed by Andersen, which can be intuitively interpreted as generating subset constraints to construct the points-to set.(Andersen, 1994) The author points out that all pointer operations except memory allocation in a C-like language can be categorised into four instruction types: pointer assignment, copy assignment, pointer reference and dereference, and each of these instruction types can be perceived as a subset constraint. Subset constraints are able to capture the points-to information of variables using set relation, and they can be propagated throughout the whole program via several simple rules, at the occasion of copy, assign and dereference to be exact.

3.1.1.1 Performance

Andersen's points-to analysis is a flow-insensitive analysis. Namely, the execution order of program statements is out of the consideration of analysis, in contrast to a dataflow analysis which keeps a state at each program point. Such analysis tends to produce imprecise result due to the lack of points-to set at the statement level. However, the time consumption is greatly reduced. Additionally, being a context-insensitive analysis is also a major advantage for Andersen's analysis in terms of performance by disregarding the context information during an inter-procedural analysis.

The efficiency of Andersen-style analysis is evaluated in the study by Sridharan and Fink as $O(n^3)$ in theory but quadratic in practice base on the theorem published by *McAllester*.(Sridharan and Fink, 2009)

3.1.1.2 Precision

Despite its adequate performance, Andersen-style analysis does not fall short for an exact analysis. As mentioned above, the flow-insensitivity of this style may hamper with precision. For instance, Andersen's analysis implies that objects in the points-to set can represent the variable simultaneously at any point of the program, whereas in the actual execution each variable can only point to a single heap location at a time.(Blackshear et al., 2011) The refinement of Andersen's analysis has been made to cope with this matter, through the application of dependency rules a statement sequence can be derived from a statement to previous statements, and such a sequence can witness the validity of the points-to relation generated in that statement.

In the context of Object-oriented language, Andersen-style analysis may also not fully content needs for context-sensitivity and field-sensitivity as addressed and improved by Sridharan et al..(Sridharan et al., 2005)

3.1.2 Steensgaard's analysis

A polynomial runtime analysis, such as Andersen's analysis is sufficient for a small program, but it may not be so efficient for a large-scale program. Steensgaard proposed a pointer analysis that functions in near-linear runtime that deliveries unparalleled scalability in practice.(Steensgaard, 1996) Steensgaard's analysis, as opposed to Andersen's subset-based approach, is using equality constraints to construct points-to information.

In order to achieve linear run time, the points-to result must be of the linear space complexity, while inherently expressing them explicitly will take $O(n^2)$ space. Steensgaard suggests the solution of unifying some of the points-to sets instead of creating a new subset as in Andersen-style. This approach progressively merges the points-to sets through assignment/copy instructions, by which it infers that no abstract object can belong in more than one points-to set, and therefore result in linear space complexity. The set relation can be then represented using a union-find tree, and the operation on sets can be performed in near-constant run time complexity by doing so. Thus, the run time performance of a Steensgaard analysis is close to linear and is able to run 1-million-line program in 1996 (Microsoft Word), that is an order of magnitude higher than any other pointer analysis known at the time.

However, the problem emerges when applying Steensgaard analysis on object-oriented languages that the performance gain does not seem to cover the loss of precision, specifically, the field-sensitivity is not an option for Steensgaard's analysis if the performance needs to be maintained.

3.1.3 Field Sensitivity

Field sensitivity is a property of pointer analysis often possesses on an object-oriented language, referring to the ability to differentiate fields of the same abstract object.(Ryder, 2003) Gaining Field sensitivity can be greatly beneficial to the overall precision, while also imposes the problem of the growing size of variables and hurts the performance. To balance this issue, Heintze and Tardieu proposed an approach of field-based analysis, which only models every instance of particular fields of same typed base objects, so that different fields sharing same base type are segregated while identical fields of these individual objects are merged, in contrast to distinguishing fields of each individual object as in field sensitive analysis.(Heintze and Tardieu, 2001) As much improvement to its performance, the type-based analysis is still considered as imprecise compared with field-sensitive analysis.

In a recent study, Sridharan et al. has suggested an algorithm that mimics a field-sensitive Andersen-style analysis under a demand-driven environment.(Sridharan et al., 2005) Their algorithm extends the previous work of *Context-Free Language (CFL) reachability* formulation of Andersen's analysis for C to the formulation for Java.(Reps, 1997) Such a formulation exposes Java analysis as a balanced-parentheses problem enabling their approximation and refinement technique to be applied. A match edge notation is adopted to match loads and stores on the same field, and by iteratively refine the match edge, it can gradually regain the precision loss from the approximation until the query is answered. Empirical evaluation in the paper also supports that higher precision can be achieved via this method than fully field sensitive analysis within a short time-bound.

3.1.4 An Efficient Strong update

As discussed above, a flow-sensitive analysis can be more precise than its counterpart, and one of the most important characteristics is that it allows *strong update*, by which the abstract memory object in the points-to relationship can be overwritten with new value, contrary to *weak update* in flow-insensitive analysis that only unions the new value with the existing set.

Since enabling full flow-sensitive could significantly increase the complexity of analysis, a hybrid strong update algorithm is presented by Lhoták and Chung in a recent study.(Lhoták and Chung, 2011) The paper proposed a strong update analysis is efficient like flow-insensitive analysis with the worst-case of quadratic space and cubic time, yet the precision also benefits from the flow-sensitive analysis. The key observation that facilitates this good comprise is that strong update can only be applied to the singleton points-to set and they are cheap to propagate flow-sensitively. By leveraging this observation, Lhoták's analysis models flow-sensitive singleton sets, while also maintains sound flow-sensitive points-to sets for all pointers in case of falling back.

3.2 Trace Reconstruction

With the solid abstraction of underlying program semantics using points-to relation, further analysis can be now conducted to find the trace in the program. There are few methods have achieved a similar result, and this section we will introduce them briefly.

3.2.1 Symbolic Execution

Symbolic executionexecution has raised the popularity in the field of software security in the last four decades. It can test whether a piece of software can violate certain properties, so it can also be applied to check the existence of a trace that holds the bug condition.(Baldoni et al., 2018)

Symbolic execution abstractly represents inputs as symbols, resort to constraint solver to construct an actual instance, it allows exploring multiple paths under different inputs simultaneously, which promises a sound analysis that results a strongly guarantees on checked properties. To be specific, a condition that needs to be satisfied along the path is described as a first-order Boolean formula. A symbolic memory store maps variable to symbolic expression, while branch execution updates the formula and assignments update the symbolic store. Eventually, a model checker based on *Satisfiability Modulo Theories (SMT) solver* is used to verify any violations/infeasibility of the properties.

Liblit and Aiken first identified the postmortem symbolic execution as a CFL-reachability problem by representing all possible failure trace with a finite state automaton.(Liblit and Aiken, 2002) However, their trace exploration ignores value flow that leads to a large amount of superfluous traces. While a classic exhaustive symbolic execution exploring all the execution states returns a sound and complete

result, challenges also raise with state space explosion and complex constraint solving. Baldoni et al. introduces a few techniques that can mitigate these impacts on scalability by sacrificing soundness, such as the adoption of *Concolic executionexecution* to avoid frequent call on constraint solver as well function and loop summary to avoid repetitive work on the same block of code. Although symbolic execution can give us a fairly accurate approximation of the bug trace existence, it is difficult to materialising the concrete trace solely relying on the concrete input if it is provided by the constraint solver.

3.2.2 Postmortem Analysis

Targeting to handle the performance problem raised in symbolic execution, Sridharan et al. describes an approach that reproduces a trace only based on a failure position and typestate information in polynomial time.(Sridharan et al., 2005) Fundamentally, the author formulates a backward dataflow analysis as an existing Tabulation algorithm on IFDS problem (Reps et al., 1995) to with solely respect to the number of variables and abstracted memory location. The insight enabling the analysis is that some common programming errors can be expressed as sequences of operation that transits an object to a special error typestate. In which sense, the typestate can be viewed as the program state and finding the trace of the typestate information will derive the program trace. By traversing backwards from the failure point, the typestate information can be propagated using the effect of the assignment until the object creation site is found or reaching to a contradiction. During the states updating, in order to handle the memory aliasing problem, a may-alias oracle is used resulting from pre-computed Andersen's points-to analysis. A few other techniques are also adopted in this approach to gain a better performance such as program slicing that pruning a portion of the program that will not contribute to the relevant variables to achieve demand-driven.

Observed evidence shows that the tool developed based on the algorithm has a practical usage, it can find the trace witness for a known Null pointer exception in Windows system within a second incorporating with error call stack information. Whereas being efficient, this approach can only explain a certain group of failures with simple type. Our approach is based on the summarisation of the program control flow graph, having a much wider application of error types. Besides, the techniques been used in this paper to trade for better scalability might hurt the precision and soundness of this analysis such as its dependency on an inaccurate alias analysis stemmed from the flow-insensitivity and finite heap partition.

3.3 Shortest Path Algorithm

In our trace reconstruction problem, one of the issues is that there could exist multiple valid traces such that the bug condition holds, while only one instance of this set of traces needs to be materialised in the final result. One of the easiest ways to determine which trace to choose is via finding the shortest one.

The shortest path problem describes finding the path between any two nodes in a graph such that the sum weights of edges in the path is minimised. In this section, we will introduce some of the most popular extant algorithms solving this problem.

3.3.1 Dijkstra Algorithm

Dijkstra proposed a greedy shortest path solution in a single-source graph by sequentially exanimate through all nodes in the graph.(Dijkstra, 1959) Dijkstra's algorithm states that starts from the entry node, each neighbour nodes will be updated with the minimum between known distance and new distance calculated based on the current working node. Though selecting the smallest distance and unvisited node, all nodes will be visited, and the shortest distance to the start node will be corrected accordingly. The correctness of this algorithm can be proved from two observations:

(1) The subpath of any shortest path is itself a shortest path.
(2) If $S(u, v)$ is a shortest path then $S(u, v) <= S(u, x) + S(x, v)$.

More importantly, the run time of Dijkstra's algorithm can be bounded with $O(M + NlogN)$ with the implementation using minimum priority queue where M is the number of the edges and N is the number of the node.

However, as mentioned above Dijkstra's algorithm is only applicable when the graph has one single source, but the trace reconstruction problem certainly does not fulfil this prerequisite since only generating the path from the bug node to the start node will not guarantee the validity of the trace. Hence, a more general solution is required to provide the shortest path between any two nodes, so we can set intermediate nodes to maintain the correctness of the trace.

3.3.2 Floyd-Warshall algorithm

Floyd has come up with an algorithm that solves all pair shortest path problem with dynamic programming.(Floyd, 1962) Floyd-Warshall algorithm describes a relation $ShortestPath(i, j, k)$ that computes the shortest possible path from node i to node j through only nodes in the node set consist of $1, 2, 3, \ldots, k$, by setting edges as the base case and the final optimal solution can be naturally derived when k is the number of nodes in the graph. Computing the $ShortestPath(I, j, k)$ relation can be considered with two cases where one is the optimal path does not go through node k that can be easily obtained from the $ShortestPath(I, j, k\check{}1)$ in the previous iteration. The second case is when actually go through node k gives a better solution than the previous iteration, we know that it must be the concatenation of path using $ShortestPath(i, k, k-1)$ and $ShortestPath(k, j, k-1)$ since they are already the optimal solution using only node $1, 2, 3, \ldots, k-1$.

Furthermore, Floyd-Warshall algorithm can not only give us the shortest distance between any two nodes, but it also allows a path reconstruction of a given nodes pair which is important to form trace witness for our problem. While other shortest path algorithms also allow such reconstruction, this feature is free of additional costs in Floyd-Warshall. By keeping a $Next$ relation along find the shortest path, the chosen node can be easily tracked at the event of an update. Reconstruction of the trace can be then realised by iteratively updating start node with the chosen one.

Let n be the number of nodes in the graph, it is clear that n iteration is required to loop through each node as an intermediate node, and within this loop, another n^2 iteration is needed in order to compute any two nodes in the graph. Run time complexity of $\theta(n^3)$ and space complexity of $O(n^3)$ is required for this algorithm. Despite the extreme simplicity of the Floyd-Warshall algorithm in logic expression, the inherent accumulativeness of Datalog may trigger growth in space usage.

It is found that Floyd-Warshall algorithm is most suited with dense graphs since its runtime complexity is irrelevant to the number of edges in the graph. However, a large-scale program normally results in a sparse control flow graph with numerous nodes, in which Floyd-Warshall algorithm is not deemed adequate in runtime bound. A possible modification on the algorithm maybe could save it out of the dilemma, since we are only interested in a specific trace of the program, there is a large redundancy of nodes that can be possibly pruned without affecting the optimality of algorithm since no negative weight is involved in the graph.

3.4 Summary

This literature review has explored the analysis and techniques used in finding trace witnesses. As the validity of trace witness heavily depends on the precision of program state abstraction, forms of points-to analysis are included to illustrate the trade-offs between precision and performance. Traditional static analysis is implemented in imperative programming languages which are normally suffering from the difficulty of algorithm realisation and cumbersome code size. The domain-specific language Datalog is evaluated to be best suited in our research area. Finally, two precedented methods are described to construct a trace from the failure point, whereas both of methods facing the challenges of full coverage of the problem. Finally, the shortest path algorithms are briefly introduced and compared determining that Floyd-Warshall Algorithm might be the most appropriate to materialise the actual trace with slight modification.

CHAPTER 4

Constant Copy Machine

The objective of this research topic is to find a path that leads to a bug in a program. The bug condition is phrased as a condition at a given point of the program, denoted as an assertion. The trace witness of such program identifies a subset of nodes, from which a unique path from the entry point to the assertion of the program can be reconstructed whilst the bug condition holds true. However, due to the non-determinism introduced by constant conditions evaluation along the path in a standard imperative language, the semantics of the language must be weakened to a point such that the analysis performed on will be deterministic.

This chapter will introduce a simplified intra-procedural machine, Constant Copy Machine(CCM), with an elementary language to simulate the behaviour of a common imperative language such as C while still under the domain of determinism in the analysis. This type of machine is in the form of a Control Flow Graph(CFG), such that the flow of the program is capable of diverging/converging along with the control flow, including loops. However as mentioned before, the conditions are introduced at these fork/merge points of the program in an imperative language, these conditions will be eliminated at such points to sustain the computability, which entitled this machine a non-deterministic and it should only be used for analysis purposes and not in actual execution. In addition to the condition eradication, some of the features such as memory management and predicate evaluation will be omitted, for that will impose some unnecessary complication to solving the basic problem.

A program in CCM can be represented as a CFG $G = (V, E, r, f)$, where V is the set of program statements and $E \subseteq V \times V$ is the set of control-flow edges. We use (u, v) to represent a flow edge, denoting the transfer of control from program statement $u \in V$ to program statement $v \in V$. There are two distinguished nodes r and f where r is the start node of the CFG, and f is the final statement.

A program path in CFG G is a finite sequence of statements, denoted by $p = \langle u_0, u_1, \ldots, u_k \rangle$, such that $(u_i, u_{i+1}) \in E$ for all $i \in [0, k-1]$. The empty program path is denoted by ϵ and the set of all program paths in a CFG G is denoted by P_G. The set $P_G(u, v) \subseteq P_G$ stands for the set of all program paths that emanate in statement u and terminate in statement v. We have a labelling function $\ell : V \to$ CCM that assigns a statement to each node in the CFG. We have the following statements in the CCM:

$$\begin{aligned} \text{CCM} ::= x := n \quad & \text{(constant assignment)} \\ | x := y \quad & \text{(copy assignment)} \\ | \texttt{nop} \quad & \text{(no-operation)} \end{aligned}$$

CCM has a finite set of variables *Var* whose values can be assumed as natural numbers. It also has a fixed set of constants that can be assigned to variables but no computations can be performed, i.e., the statements transfer variable values only. We use x and y to represent variables, which belong to the finite variable set Var, whereas $n \in \mathbb{N}$ stands for constant. A statement in CCM can be either a constant assignment $x := n$ that assigns a constant value n to the variable x, or a copy assignment $x := y$ that copies the value stored in variable y to the variable x, or a no-operation statement that no operation will be executed.

In order to illustrate the semantics of CCM, we introduce the program context $c : Var \to \mathbb{N}$, which is a function that maps variables to values. The semantics function σ for statements in CCM σ : CCM $\to (c \to c')$ is given below:

- $\sigma[\![x := n]\!] \equiv \lambda c \,.\, c_{[x \leftarrow n]}$: the constant assignment updates the value of variable x with constant n in the program context c and keeps other variables unchanged.
- $\sigma[\![x := y]\!] \equiv \lambda c \,.\, c_{[x \leftarrow e(y)]}$: the copy assignment $x := y$ updates the value of variable x with the value of variable y in the program context c, and keeps other variables unchanged.
- $\sigma[\![\texttt{nop}]\!] \equiv \lambda c \,.\, c$: similar as the assertion statement, the no-operation statement also keeps the program context $c$unchanged.

For sake of simplicity, we abuse the notation of semantics function σ and extend it for program paths, i.e., $\sigma : P_G \to (c \to c')$. The semantics function σ for program paths is defined as

$$\sigma[\![p]\!] \equiv \begin{cases} \lambda c \,.\, (\sigma[\![p']\!])\sigma[\![\ell(u)]\!], & \text{if } p \neq \epsilon \text{ and } p = u \cdot p' \\ \lambda c \,.\, c & \text{otherwise (i.e., } p = \epsilon) \end{cases}$$

An example of this machine can be viewed in the graph below

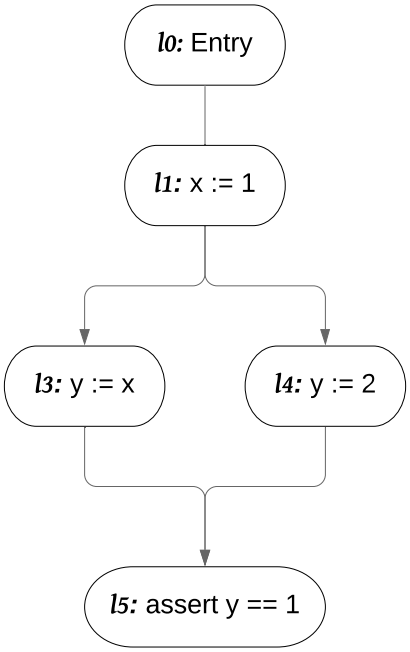


FIGURE 4.1. Example in CCM

The evaluation of a program path p is the semantics evaluation $\sigma[\![p]\!]c_0$ where c_0 represents the initial mapping for the variables in the program, i.e., all variables in the program are assumed to point to 0 initially. A bug condition can be expressed as an assertion denoted by $\mathcal{A}$, which is used to check whether variables have some specific values at the checking point for a context c that $\mathcal{A}(c)$ holds.

Problem Statement. A trace witness is, in fact, a program path that emanates from the start node r and terminates in the final node f, i.e., the assertion statement. The trace witness problem asks for a trace witness (particular the shortest one) for which the assertion $\mathcal{A}$ holds under the context c for a given input program.

DEFINITION (Trace-Witness-Problem). *An instance of the trace witness problem is given by the quadruple $(G, \mathcal{A}, Var, \ell)$ where G is the control flow graph of the program, $\mathcal{A}$ the assertion, Var the set of variables in the instance, and ℓ the labelling function. The solution of the trace-witness problem is the shortest program path p_s such that $\mathcal{A}(\sigma[\![p_s]\!]c_0)$ holds.*

CHAPTER 5

Single Static Assignment Form

5.1 Definiton

Single Static Assignment (SSA) Form (Rosen et al., 1988) is a property of an Intermediate Representation, which provides an efficient solution to address the problem of encoding data-flow information at compile time.(Reif and Lewis, 1977) As per required, each variable in SSA form must only be assigned once throughout the whole program, and its name carries the context information at the definition.

A motivating example is given through the following program

```
1 X = 1;
2 X = 2;
3 Y = X;
```

A flow-insensitive analysis could not differentiate the order between the first and the second assignments to X. When assigning X to Y, the analysis would have to take both potential values (i.e., 1 and 2) into account, which leads to an imprecision that may grow substantially from the snowball effect. Just like the motivating example program, variables with multiple assignments will introduce ambiguity when performing a flow-insensitive analysis.

In contrast, a flow-sensitive analysis can determine the execution order of instructions of a given program. There exist certain ways to perform flow-sensitive analysis such as setting up data flow equation in each node in the graph and solve it locally.(Kildall, 1973) SSA is another technique we can adopt to perform such an analysis.

The property of SSA form infers that any use of a variable uniquely maps to one single definition

through eliminating assignments to the same variable. (Alpern et al., 1988) The conversion from the original program into the SSA form one can be found as below.

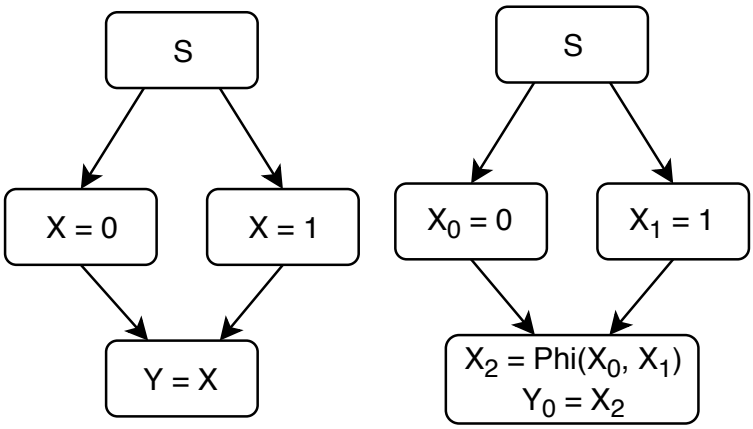
SSA Form Example

```
1 X0 = 1;
2 X1 = 2;
3 Y0 = X1;
```

In the above code, each definition of X is renamed with a subscript to ensure unique variable names. Thus, the value of X can be determined at compile time when it is assigned to Y. use of a variable refers to access/read the value holden by the variable, while def refers to definition/initialisation of a variable. The $use - def$ chain maps the use of a variable to its def. Generally speaking, the $use - def$ chains are considered implicitly during backward probing, i.e look for the def of a use.

It is intuitive to rename variables at definition point to create unique variable names, whereas the problem arises when distinct definitions of a variable reach a join point. Complying with the property of SSA that the use of a variable only refers to a single definition, a special phi-function needs to be inserted at a joint point, and the node inserted is often referred as phi node. A phi-function can be considered as a pseudo-assignment since it introduces a new variable definition that takes all the incoming variable names of a variable as arguments and denotes the value of this new variable could be any value from the predecessors.

An example is given below illustrating the conversion from an original program into the SSA form program with a phi-function inserted.



(A) Original (B) SSA

FIGURE 5.1. Conversion to SSA Form

Note that, each variable with multiple definitions is required to place a phi-function at the top of the merging block that should execute simultaneously.

5.2 SSA Construction

Constructing SSA form has an established and efficient algorithm, proposed in (Cytron et al., 1991). There are two steps in SSA construction, including placing phi-functions and renaming. The placement of phi-functions is crucial to the performance of SSA construction and corresponding analysis. Naïvely thinking, one could place a phi-function at the joint point for each variable used in the following block. However, introducing superfluous phi-functions may increase the analysis cost drastically, even though without affecting the correctness of produced SSA form.

An algorithm has been devised to build a Semi-Pruned SSA form by computing a dominance frontier set (Briggs et al., 1998), which can guide the insertion of phi-functions to only certain points where are needed. The number of phi-functions can be further reduced based on the observation that a definition is redundant if no use of the variable followed. The details of the algorithm and implementation will be discussed in the next section in the form of first-order logic.

5.3 Algorithm & Implementation

The key to reducing the number of extraneous phi-functions lies in the problem of identifying where the phi-function is needed. Consider a definition node n in the CFG, the dominance relation $m \in Dom(n)$ represents all paths that reach m must pass through node n (Prosser, 1959). The dominance information indicates that for all m dominated by n, a phi-function does not need to be inserted since the definition at n will not be interfered by another definition from other branches.

5.3.1 Dominance

The dominance set for a node $m \in Dom(n)$ in the CFG can be computed by considering it recursively.(Cooper et al., 2006)

Algorithm 1: Compute Dominance Set

if *m = Start* **then**
  foreach *Node n in CFG* **do**
    m := Dom(n);
  end
else if $\forall_p$ *m = Dom(p)* $\wedge$ *p* $\in$ *Parent(n)* **then**
  m := Dom(n);
end

However, the recursive case above infers a universal quantifier("for all"), which cannot be easily interpreted to Datalog rule without a few tweaks since each rule implicitly applies an existential quantifier("for each") in conjunction when relations are joined in a rule. While Dominance relation cannot be directly represented in rules, the negation of it can be effectively computed and result in the dominance set eventually.

The rule below sets a base case for the $NonDom$ relation that none of the nodes m in CFG dominates the start node n except for n itself. Note that the start node dominates all the other nodes in CFG.

```
1  % m does not dominate n
2  .decl NonDom(n: Node, m: Node)
3  NonDom(n, m):-
4      Start(n),
```

```
5       Nodes(m),
6       n != m.
```

The following rule describes that $NonDom$ can be propagated through edges of CFG from the start node. It guarantees that n is either the parent of m or there exists at least one path to reach m without going through n.

```
1   NonDom(n, m):-
2       NonDom(o, m),
3       Edge(o, n),
4       !Start(n),
5       n != m.
```

With the help of $NonDom$ relation, Dom can be obtained easily by negation. The $StrictDom$ set is the same as the Dom set but excluding the current node itself.

```
1   % m dominates n
2   .decl Dom(n: Node, m: Node)
3   Dom(n, m):-
4       Nodes(n),
5       Nodes(m),
6       !NonDom(n, m).
7
8   % m strictly dominates n
9   .decl StrictDom(n: Node, m: Node)
10  StrictDom(n, m):-
11      Dom(n, m),
12      n != m.
```

5.3.2 Immediate Dominance

An Immediate Dominance $IDom(n)$ denotes the closest node that strictly dominates n (i.e. $Dom(n) - \{n\}$), that is 1) $IDom(n)$ strictly dominates n and 2) it is also dominated by all other nodes strictly dominates n.

Assume $\{d_0, d_1, d_2, \ldots, d_k\}$ as the strict dominators of a node m, it can be inferred that $Dom(d_n) +$

$\{d_n\} = Dom(IDom(d_n)), for\, n \in [1, k]$. This relation can be used to chain the dominator set

$$Dom(d_0) \subset Dom(d_1) \subset Dom(d_2) \subset \cdots \subset Dom(d_n) \subset Dom(m)$$

An order of dominance can be built from this observation by comparing the size of the dominance set of each dominator, and the immediate dominance will be the one with the largest size.

```
1  % c = count(SDom(n))
2  .decl StrictDomSize(n: Node, c: number)
3  StrictDomSize(n, 0) :-
4      Start(n).
5  StrictDomSize(n, c) :-
6      StrictDom(n, _),
7      c = count : StrictDom(n, _).
8
9  % m strictly dominates n, and domianted by c nodes
10 .decl StrictDomSizes(n:Node, m:Node, c:number)
11 StrictDomSizes(n, m, c) :-
12     StrictDom(n, m),
13     StrictDomSize(m, c).
14
15 % m = IDom(n)
16 .decl ImmediateDom(n: Node, m: Node)
17 ImmediateDom(n, m) :-
18     StrictDomSizes(n, m, d),
19     d = max c : StrictDomSizes(n, _, c).
```

$StrictDomSize(n, c)$ stores the size c of nodes that strictly dominates node n and $StrictDomSizes(n, m, c)$ associates the count with each strict dominance relation. The immediate dominance set can be determined by finding the dominator of a node with the maximum size of the strict dominator.

5.3.3 Dominance Frontier

The Dominance Frontier of a node n is a set of closest nodes that are reachable from n and not strictly dominated by n. Because Dominance Frontier outlines the border of dominance, a phi-function is safe to be placed at the dominance frontiers of a definition without introducing much redundancy.

Formally speaking, a definition in node n forces a phi-function at each $m \in DF(n)$, where

(1) n dominates a parent of m ($q \in Parents(m) \wedge n \in Dom(q)$)

(2) n does not strictly dominate m ($n \notin StrictDom(m)$).

The motivation for using Strict Dominance instead of Dominance is because Strict Dominance allows placing a phi-function at the same node with the definition, such as the beginning of a loop when we consider a loop case.

The computation of DF is based on the following three observations

(1) Node n must be the join point to be considered in Dominance Frontier set.

(2) Join node n must be the Dominance Frontier of each parent p.

(3) If some node m dominates p, n must of be the Dominance Frontier of p, unless p dominates n.

The algorithm below shows the computation in an imperative language from (Cooper et al., 2006)

Algorithm 2: Compute Dominance Frontier For Every Node in Graph

```
foreach Node n ∈ CFG do
    DF(n) := ∅;
end
foreach Node n ∈ CFG do
    if count(Parent(n)) > 1 then
        foreach Parent p ∈ Parent(n) do
            runner := p ;
            while runner ≠ IDom(n) do
                DF(runner) := DF(runner) ∪ n;
                runner := IDom(runner);
            end
        end
    end
end
```

The algorithm starts with every join node n in the CFG (Observation 1), which is also the DF of its parent p (Observation 2), it then gathers more nodes that have node n as DF by walking through the Dominance chain using $IDom$ relation along each parent p, until it reaches the node that dominates n

(Observation 3). Intuitively, there would not exist another node between $IDom(p)$ and p, which has n as dominate frontier since $IDom$ is the closest node that dominates p. Similarly, since $IDom(n)$ would not have n as its Dominance Frontier, neither would the predecessors of $IDom(n)$. This algorithm can also be expressed more succinctly in Datalog with three rules corresponding to each observation above.

The following rule computes $JoinNodes(n)$ by counting incoming edges for all nodes. A join point can be expressed as a node with more than one incoming edges. (Observation 1)

```
.decl JoinNodes(n: Node)
JoinNodes(n):-
    Nodes(n),
    c = count : Edge(_, n),
    c > 1.
```

The rule below serves as the base case to compute Dominance Frontier. For each joint point n obtained above, n will be the Dominance Frontier of every parent of n. (Observation 2)

```
% n = DF(m)
.decl DomFrontier(n: Node, m: Node)
DomFrontier(n, m):-
    JoinNodes(n),
    Edge(m, n).
```

Finally, the rule $DomFrontier(n, runner)$ is the recursive case of computing Dominance Frontier. For each n as the Dominance Frontier of m currently, if the Immediate Dominance $runner$ of m is not the Immediate Dominance of n, then n will also be the Dominance Frontier of $runner$. Since it's recursive, this rule will keep adding the newly generated DF as an input, until no more facts can be produced. (Observation 3)

```
% DF(m) = n, IDom(m) = runner, IDom(n) != runner
% =>  DF(runner) = n
DomFrontier(n, runner):-
    DomFrontier(n, m),
    ImmediateDom(m, runner),
    !ImmediateDom(n, runner).
```

5.3.4 Phi-Function Placement

The computation of Dominance Frontier has narrowed down the set of nodes where phi-function is needed in the CFG, yet the set could be further reduced through the $use - def$ chain, namely Semi-Pruned SSA.(Briggs et al., 1998) This optimisation is based on an observation that the definition of a variable is not required if it is not used in CFG after its definition. The algorithm can be devised accordingly to compute the set of nodes to insert phi-function where at least one use of that variable will follow after.

Firstly, the potential set of nodes to insert phi-function needs to be chosen through the Dominance Frontier set obtained in the previous section. The rule takes every definition node of a variable and the Dominance Frontier of which should be the node we are looking for. Note that since phi-function defines a new variable, it should also be interpreted as a special case of definition and be included alongside with Def.

```
.decl T_PhiNode(n: Node, var: Var)
T_PhiNode(df , var):-
    (Def(def, var);
    T_PhiNode(def , var)),
    DomFrontier(df, def).
```

The key to pruning a node in $T\_PhiNode$ is to find if a use of the variable followed. The search of it, however, does not need to probe until the endpoint of the program. The Dominance relation we calculated before now comes handy, because a definition only governs the $uses$ within the dominance range, and any use out of the range will be covered by the dominance frontier of which definition. This knowledge bounds the search of use only to the dominance set of a definition, as suggested in the rule below. Similar to the definition, each phi-function is also a special use since it takes variables as its arguments and it should be included along with use.

```
% Phi-node at n for var
.decl Pruned_Phi(n: Node, var: Var)
Pruned_Phi(m, var):-
    T_PhiNode(m, var),
    Dom(n, m),
    (Use(n, var);
    Pruned_Phi(n, var)).
```

5.3.5 Renaming

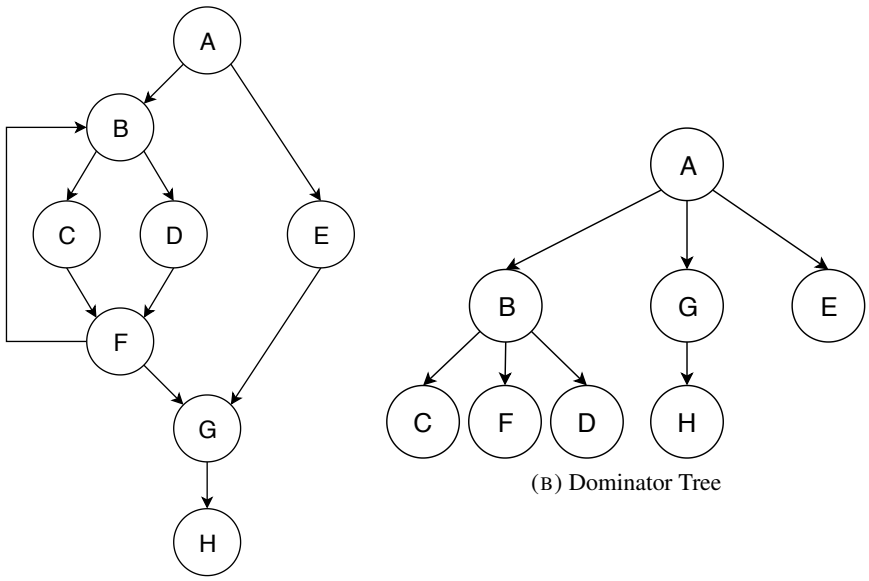
To differentiate each definition in SSA form, a sequential subscript is required to be added to the original variable name. For example, $x = 1$ in the original language may be converted to $x_0 = 1$. Since our analysis is intra-procedural at the stage and adheres to one statement each line, the sequential subscript can be simply replaced by the line number without affecting the uniqueness of the variable name.

The process of renaming the variable at the definition point is shown in the following rule.

```
1 .decl N_Def(n: Node, var: Var)
2 N_Def(n, cat(cat(var,"_"), n)):-
3     Def(n, var).
```

With variable names updated at the definition point, variable names at the use should be replaced as well. Despite the simplicity of renaming variables at the definition, finding the definition corresponding to the use could be a challenging task, considering the involvement of phi-function.

Before delving into the renaming algorithm, an important concept needs to introduced here. As discussed in the previous sections, an Immediate Dominance describes the closest node that dominates a node, and therefore a chain of dominance can be formed by following the $IDom$ relation. Considering the whole CFG, it can also be interpreted as an aggregation of Dominance chains, namely a Dominator Tree. A Dominator Tree contains every node in the CFG, and the nodes can connect via $IDom$ relation, i.e. if $m = IDom(n)$, then there is an edge in the Dominator Tree from m to n. Dominator Tree also encodes Dom as every node along the path from the root to the node in query. An instance of such conversion is illustrated below, and the Dominator Tree can be efficiently calculated using Lengauer–Tarjan algorithm.(Lengauer and Tarjan, 1979)



(A) CFG

(B) Dominator Tree

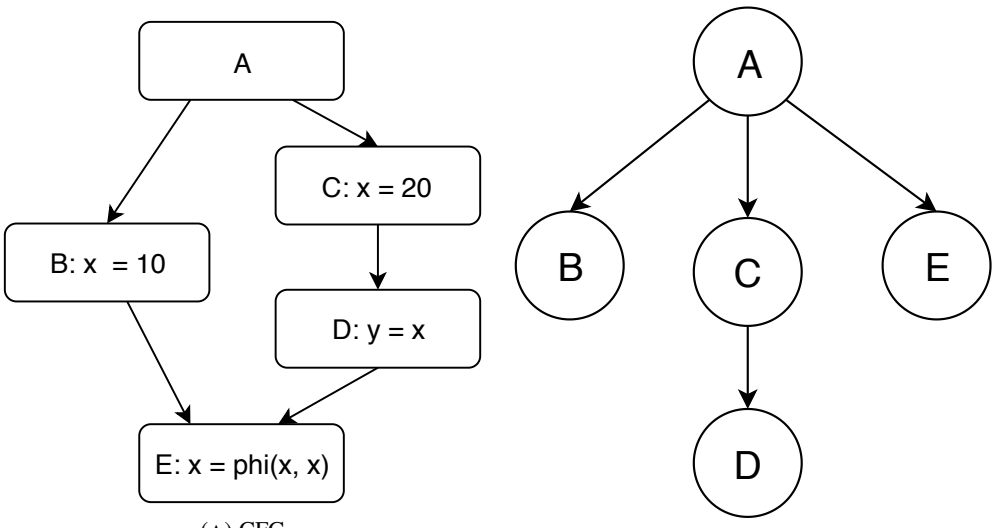
FIGURE 5.2. Example CFG and Its Dominator Tree

The task of renaming variable name at use point is to find the closest node with the variable defined.(Torczon and Cooper, 2011) Because the property of SSA form after the phi-function insertion, it's easy to induce that for a use of variable x at node n, there must exist a node $m$such that $m \in Dom(n)$ and m defines variable x in either assignment or phi-function. This observation guarantees that there always exists a node defines the variable to use in the path from the root node to every use node in the Dominator Tree. (Brandis and Mössenböck, 1994) A backward search can then be adopted to find the closest definition node in the path starting from every use node.

A normal variable use such as an assignment can be dealt with the method described above. Nevertheless, a phi-function as a peculiar use node does not fit in with the algorithm. The reason behind this aberrant behaviour is because the observation does not apply to the phi-function as a use since the purpose of placing phi-function is to gather non-dominant precedents, so it is expected that the arguments of a phi-function are defined outside of the dominators of the phi-node. On top of which, the search algorithm will stop once it finds the closest node with the variable definition, but the phi-function requires multiple variables with the same name in the source language.

For instance, apply the backward searching algorithm to the following CFG with its Dominator Tree

works well for the use in node D, the corresponding definition C can be determined by walking backwards in the Dominator Tree. While for the phi-function in node E, it does not return the correct result that should be in node B and C.



(A) CFG (B) Dominator Tree

FIGURE 5.3. Example For Backward Searching Algorithm

A solution to this problem is spawning a series of searching tasks with each represents a parent of the phi-function. To implement this approach, one can assign each parent of the phi-function with the use of the variable in phi-function. This approach will not only set a bound to the number of arguments in the phi-function as well as the searching direction regarding each argument.

$MayUse$ is defined to store the use of variables as well each parent of pruned phi-functions from obtained from the last section.

```
1  .decl MayUse(n: Node, var: Var)
2  MayUse(n, var):-
3      Use(n, var);
4      (Pruned_Phi(m, var),
5      Edge(n, m)).
```

$ShortestDef$ is defined to store the intermediate($bool$=0) as well as final($bool$=1) result during the searching progress depending on $bool$, for var used at n and its dominator m. The rules below describe

the base case the shortest definition should be itself when a variable is used and defined in the same node, otherwise mark the node as intermediate.

```
% use node n is dominated by m
% bool indicates if m is a definition for var
.decl ShortestDef(n: Node, m: Node, var: Var, bool: number)
ShortestDef(n, n, var, 1):-
    MayUse(n, var),
    (Def(n, var);
    Pruned_Phi(n, var)).

ShortestDef(n, n, var, 0):-
    MayUse(n, var),
    !Def(n, var),
    !Pruned_Phi(n, var).
```

For every intermediate $ShortestDef$ fact, a further probing along the dominator tree is required. By looking up the Immediate Dominance m of an intermediate node o, the use var in n is defined in m if m contains its definition, otherwise, it sets m as another intermediate result until it reaching to the start node, where we presume every variable is defined.

```
ShortestDef(n, m, var, 1):-
    ShortestDef(n, o, var, 0),
    ImmediateDom(o, m),
    (Def(m, var);
    Pruned_Phi(m, var)).

ShortestDef(n, m, var, 0):-
    ShortestDef(n, o, var, 0),
    ImmediateDom(o, m),
    !Def(m, var),
    !Pruned_Phi(m, var).
```

With the $use - def$ mapped for each use, every variable is used in a use node can be renamed accordingly.

```
.decl N_Use(n: Node, var: Var)
N_Use(n, cat(cat(var,"_"), def)):-
    Use(n, var),
    ShortestDef(n, def, var, 1).
```

The final form of phi-function can be also defined and each fact in $PhiNode$ represents one argument taken by a phi-funtion.

```
1 .decl PhiNode(n: Node, var: Var, arg: Var)
2 PhiNode(n, cat(cat(var,"_"), n), cat(cat(var,"_"), def)):-
3     Pruned_Phi(n, var),
4     Edge(m, n),
5     ShortestDef(m, def, var, 1).
```

CHAPTER 6

Path Construction

Path construction refers to the process of building a complete path from the assertion, where is typically the possible bug point, back to the entry point of the program concerning both the control flow and the data flow. This chapter will introduce a sequence of algorithms that helps to find such a path, given the input in the form of CCM and SSA that introduced in chapter 4 and chapter 5.

6.1 Chain of Data Dependencies

Data dependency denotes the data of a program statement refers to the data of the preceding statements (Patterson and Hennessy, 1990), and a chain of data dependencies consists of a set of assignments where the right-hand side of the assignments is either a constant or a variable defined by a statement in the set, forming a topological order between assignments.(Ferrante et al., 1987) The construction of chains of data dependencies abstracts the data flow in the program as a tree structure, with which the traversal becomes much easier to perform because the property of SSA constrains that there must be a viable path for each variable from its usage to its definition.

For example, in the following program, the value of the variable a_2 is asserted to be 1 at line 4, which will be the start point of the data dependencies of a_2 through a backward search. The data dependency can be determined for a_2 by looking back to the point (i.e., line 3) where a_2 is defined by a copy assignment with the value of a_1, while a_1 is also defined by a copy assignment at line 2. The search does not end until it reaches a_0 at line 1, which is defined by a constant assignment. Now the chain of data dependencies of the variable a_2 can be formed as $a_2 = a_1$, $a_1 = a_0$, $a_0 = 1$.

```
1 a0 = 1;
2 a1 = a0;
3 a2 = a1;
4 assert a2 == 1;
```

A special case to be considered in the chain is at the presence of the confluence point (i.e. phi-node), in which case the chain of data dependencies has to split into sub-streams, in consideration of multiple possible sources of a variable defined in a phi-node. The presence of a phi-node propagates the chain of data dependencies, and it also manifests that the path from assertion to the entry point might not be singular.

While chains of data dependencies list all the possibilities of a path, the feasibility of them in respect to the assertion has yet acknowledged, by which it means not all variables can contribute to the path construction. The assertion in a program is assumed to be in the form of $X == constant$, which can be regarded as a filter to sift the constant assignments with different sources of the assertion.

```
.decl PruneVars(x:Variable)

% Constant assignments whose values differ from
% assert statement, can be pruned.
PruneVars(x) :-
    ConstAssign(_, x, c1),
    Assert(_, _, c2),
    c1 != c2.
```

With the variables pruned at the constant assignment, any further utilisation of these variables should also be restricted. Further pruning can be conducted by following the data dependency chains of extant pruned variables. As the rule describes below, any variable that takes a pruned variable as a source in a copy assignment will also be pruned.

```
% Copy assignments whose source variable is pruned
PruneVars(x) :-
    PruneVars(y),
    CopyAssign(_, x, y).
```

A phi-assignment is also an assignment but with multiple sources and needs to be considered differently. The variable assigned at the phi-node can be pruned only when all variables in the phi-function arguments are pruned. Take the following graph as an example, x_0 clearly should be pruned due to the value of variable x_0 is in incompatible with the required value (i.e., 1) in the assertion. Despite taking

x_0 as a source in the phi-assignment, x_2, however, will not be pruned because of the existence of the other source x_1 which stays intact. In summary, a variable at a phi-node can only be pruned when all arguments in the phi-function are pruned.

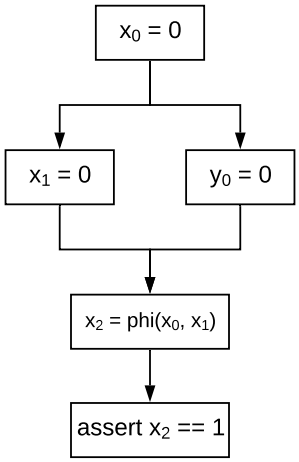


FIGURE 6.1. Exmaple For Not Pruning Phi Node

However, the placement of phi-node may be taken even before the definition of arguments in the phi-function (e.g. a program contains a loop structure as shown in the graph below), which may lead to a convoluted relationship between variables in phi-node and jeopardise the pruning process. Consider the phi-function in the following graph $x_2 = phi(x_0, x_1)$ and we assume x_0 is pruned due to the mismatching value with assertion. However, x_2 is not yet pruned because of the presence of x_1, which is also unpruned because of the existence of x_2. A cyclic dependency hence can be formed in a chain of data dependency chains when more than one variables are responsible for the definition of the preceding variables.(Lakos, 1996) One can argue that when pruning the arguments in phi-node, it is imperative to present the consideration of cyclic dependency.

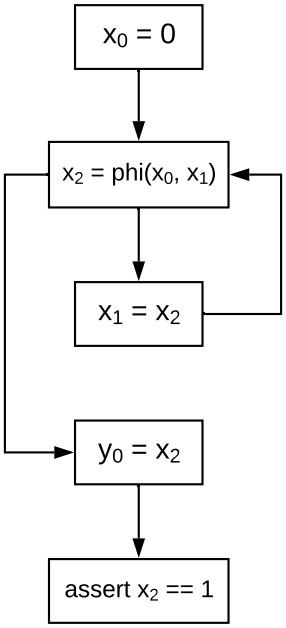


FIGURE 6.2. Cyclic Dependency

Finally, all the observations above can assemble to the Datalog code below where the data dependencies can be determined by applying transitive closure to the def-use chain and the variables are cyclic dependent are simply the variables that depend on each other.

```
1  .decl DataDependency(source: Variable, destination: Variable)
2
3  % Immediate transfer of values
4  %  x = y => (y, x)
5  DataDependency(y, x) :-
6      CopyAssign(_, x, y);
7      PhiArgs(x,y, _).
8
9  %  Intermediate transfer of values
10 %  x = y; y = z; => (z, x)
11 DataDependency(z, x) :-
12     (CopyAssign(_, x, y);
13     PhiArgs(x, y, _)),
14     DataDependency(z, y).
15
16 .decl CyclicDependency(u: Variable, v: Variable)
17 CyclicDependency(x, y) :-
```

```
18      DataDependency(x, y),
19      DataDependency(y, x).
```

To prune variables x at phi-node n, one must fulfil that $\forall a \in args(n) a \in Pruned \vee a \in CyclicDep(x)$, nevertheless, Datalog may experience the difficulty of expressing logic in $\forall$ as discussed in the previous chapter. In order to counter this issue, a sequential method can be adapted to mimic universal quantifier. Consider a predicate $PrunePhiArgs(x, n)$ where x is the assigned variable in a phi-function, n stands for the number of arguments pruned so far that increment by 1 for each pruned argument. Assume the number of argument at this phi-function is k, and x can be pruned once n reaches to k since it implies every argument has been pruned or is cyclic dependent with x.

```
1   .decl PrunePhiArgs(x:Variable, i:number)
2
3   % For all arguments (up to k) the predicate PrunePhiArgs
4   % must hold.
5   PruneVars(x) :-
6       PhiAssign(_, x, k),
7       PrunePhiArgs(x, k).
8
9   % Base case for first argument, i.e,
10  % argument can be pruned or is cylic dependent
11  PrunePhiArgs(x, 1) :-
12      PhiArgs(x, y, 1),
13      (PruneVars(y);
14      CyclicDependency(x, y)).
15
16  % Inductive case for remaining arguments
17  PrunePhiArgs(x, i + 1 ) :-
18      PrunePhiArgs(x, i),
19      PhiArgs(x, y, i + 1),
20      (PruneVars(y);
21      CyclicDependency(x, y)).
```

Now everything can be patched together to generate the chain of data dependency that is associated with the variable in the assertion.

```
1   % Variable in assert statement is always reachable
2   BackwardReachable(x) :-
3      Assert(_, x, _).
```

For every assignment/phi-function, if the assigned variables are not yet pruned, it can be classified as reachable.

```
% Copy and Phi assignment
% Reachable if the source is not pruned
BackwardReachable(y) :-
   BackwardReachable(x),
   (CopyAssign(_, x, y);
    PhiArgs(x, y, _)),
   !PruneVars(y).
```

Note that every variable in $BackwardReachable$ indicates that it does not invalidate assertion, it is not guaranteed that they will form a single path/chain of data dependency. In order to achieve this, a new algorithm has to be used to compute a single valid path from the assertion to the start of the program. When considering a single instance of a pool of paths, one may intuitively seek the most efficient solution, in this case, the shortest path amongst a group of paths.

6.2 Shortest Path Algorithm

Floyd-Warshall algorithm is known as an efficient solution for finding the shortest path among all pairs of nodes in a directed weighted graph.(Floyd, 1962) The motivation behind choosing Floyd-Warshall algorithm despite of its $\theta(n^3)$ time complexity compared with $O(n^2)$ of other shortest path algorithms like Dijkstra's algorithm is that Floyd-Warshall is able to compute the shortest path between every pair of nodes in the graph, while Dijkstra determines only the given source-sink pair. The extra time complexity allows the result of the algorithm to be queried multiple times without any recomputation, which may occur quite frequently during program analysis.

Floyd-Warshall algorithm leverages the principle of dynamic programming to achieve its simplicity and efficiency. The nature of dynamic programming indicates that for finding the optimal solution, a sub-problem must be defined and solved based on the previous step. In the case of Floyd-Warshall algorithm, this sub-problem can be defined on the observation that if the shortest path from node i to j passes an intermediate node k, the path must compose of the shortest path from i to k and k to j.

The observation above enables us to establish the solution to the shortest distance problem, which can

be later extended to the shortest path problem. For a directed weighted graph G with n nodes, let $D_{ij}^{(k)}$ be the shortest distance between i and j at k^{th} iteration using intermediate nodes set $\{1, 2, \ldots, k\}$. For the base case, when $k = 0$, assign the distance from i to j to the edge weight w, which is 1 in the case of CFG for the simplicity, for every pair of i, j with a direct edge, and ∞ for other pairs of nodes that without a direct edge in between. The algorithm terminates at the iteration n since the subproblem has saturated the problem when all nodes can be used as intermediate nodes, and $D_{ij}^{(n)}$ represents the shortest distance from node i to j. This procedure can formalise to an equation below

$$D_{ij}^{(0)} = \begin{cases} 1 & \text{if Edge(i, j) exists} \\ \infty & \text{else} \end{cases}$$

$$D_{ij}^{(n)} = min(D_{ij}^{(n-1)}, D_{ik}^{(n-1)} + D_{kj}^{(n-1)})$$

The algorithm so far has only considered the shortest distance between each pair of nodes, while the ultimate goal is to construct a path out of the optimal distance. For addressing this issue, no extra computation is needed apart from adding another relation to hold the next node of a path in the optimal solution. To further illustrate the update of $Next$ relation in each iteration, the equation below declares that $Next_{ij}^{(n)}$ as the next node after i in the shortest path from i to j in iteration n.

$$Next_{ij}^{(0)} = \begin{cases} j & \text{if exists Edge(i, j)} \\ i & \text{if} \quad i = j \\ null & \text{else} \end{cases}$$

$$Next_{ij}^{(n)} = \begin{cases} Next_{kj}^{(n-1)} & \text{if } D_{ij}^{(n-1)} > D_{ik}^{(n-1)} + D_{kj}^{(n-1)} \\ Next_{ij}^{(n-1)} & \text{else} \end{cases}$$

For implementing the Floyd-Warshall algorithm in Datalog, the notion of iteration must be addressed in advance since it is the core of dynamic programming. As Datalog program executes declaratively, the strategy of the loop is no longer available here. Nevertheless, an additional attribute in the relation can help to mimic the loop in an imperative language. Take the $ShortestKPath(u : Node, v : Node, k : Node, dist : number)$ for an example, this relation represents the shortest distance $dist$ from node u to v at k_th iteration. Similarly, $Next(u : Node, v : Node, k : Node, succ : Node)$ also uses k as the iteration index. The following rules are used to describe the base cases for $ShortestKPath$ and $Next$ when $k = 0$ (i.e. No intermediate node in the shortest path)

```
% Shortest path from u to v in iteration k
.decl ShortestKPath(u:Node, v:Node, k:Node, dist:number)

% succ is the next node from u to v in iteration k
.decl Next(u: Node, v: Node, k: Node, succ: Node)

% set self-loops to zero distance
ShortestKPath(1, 1, 0, 0).
Next(1, 1, 0, 1).
ShortestKPath(i+1, i+1, 0, 0),
Next(i+1, i+1, 0, i+1) :-
    ShortestKPath(i, i, 0, 0),
    NumNodes(n),
    i < n.

% set Edges in the graph to unit distance
ShortestKPath(i, j, 0, 1),
Next(i,j,0, j) :-
    Edge(i, j).

% set remaining connections to n+1 distance,
% representing that they are too expensive to traverse.
ShortestKPath(i+1, j, 0, n+1),
Next(i+1, j, 0, -1) :-
    ShortestKPath(i, j, _, _),
    !Edge(i+1, j),
    i+1 != j,
    NumNodes(n),
    i < n.
```

In recursive rules, k as the index of iterations incremented from 0 to n, where n is the number of nodes in the graph as well as the number of iterations required to compute the optimal solution in Floyd-Warshall algorithm.

```
% Shortest path via k is longer
ShortestKPath(i, j, k+1, d),
Next(i,j, k+1, succ):-
    ShortestKPath(i, j, k, d),
    ShortestKPath(i, k+1, k, d1),
    ShortestKPath(k+1, j, k, d2),
    Next(i, j, k, succ),
    d <= d1 + d2,
    NumNodes(n),
    k < n.
```

```
11
12 % Shortest path via k is shorter
13 ShortestKPath(i, j, k+1, d1+d2),
14 Next(i, j, k+1, succ) :-
15     ShortestKPath(i, j, k, d),
16     ShortestKPath(i, k+1, k, d1),
17     ShortestKPath(k+1, j, k, d2),
18     Next(i, k+1, k, succ),
19     d > d1 + d2,
20     NumNodes(n),
21     k < n.
```

The final result of this algorithm should contain sets of nodes that each set can construct the shortest path for a pair of nodes in the graph unless they are unreachable. For example, to obtain the set of nodes that construct the shortest path from i to j, the computation can be done via traversing $Next$ relation. If the $Next$ points to k at n_th iteration, k will be included as one of the intermediate node, and the traversal can continue by replacing the source i to k and searching for $Next$ from k to j until $k = j$. The rules that perform this searching can be written as below.

```
1  % base case
2  ShortestPathWitness(i, j, i) :-
3      Next(i, j, n, _),
4      ShortestKPath(i, j, n, d),
5      d != n + 1,
6      NumNodes(n).
7
8  % inductive case
9  ShortestPathWitness(i, j, succ) :-
10     ShortestPathWitness(i, j, u),
11     Next(u, j, n, succ),
12     NumNodes(n).
```

6.3 Trace Witness Generation

The chain of data dependencies provide a set of variables in CFG with no violation of the assertion, and the shortest path algorithm can build up a concrete path using a subset of nodes in the graph. This section will dedicate to merging the two sides of the trace witness and produce the actual trace without violating the assertion.

The task of finding a trace in a chain of data dependencies in fact is to determine the key points and their orders to pass through. For example, the definition($y = x$) for the variable y in assertion ($y == 1$)can be considered as a key point, and the next key point should be the assertion itself while the previous one should be the definition of $x = 1$. The identification and ordering of key points allow the elimination of invalid key points and reconstruction of paths between key points with the help from Floyd-Warshall. $ShortestTrace(v : Node, prev : Node, d : number)$ used to describe the order of key points, where $prev$ is the previous key point of v and the distance from the entry point to v as d to keep track of distance travelled so far.

For a variable evaluated in assertion, it is guaranteed that the value comes from a constant assignment in CCM, so the initial action can be done to find the key points of the valid(reachable) constant assignment.

```
.decl ShortestTrace(v:Node, prev:Node, d:number)

% Base case: const assignments
% (find the shortest path from
%  start node of CFG to constant
%  assignment)
ShortestTrace(v, s, d) :-
   ConstAssign(v, x, _),
   BackwardReachable(x),
   Start(s),
   fw.ShortestPath(s, v, d).
```

Of course, the valid copy assignments that transfer values in constant assignments should also be considered as key points, as well as the assertion, which should always be the last key point in a path.

```
% shortest trace computation for copy assignment and assertion
ShortestTrace(v, u, d1 + d2 ) :-
   ShortestTrace(u, _, d2),
   VarDef(y, u),
   ((CopyAssign(v, x, y),
    BackwardReachable(x));
    Assert(v, y, _)),
   fw.ShortestPath(u, v, d1).
```

Unlike copy assignment with single-source variable, the phi-function as the confluence point of the CFG takes multiple variables as a source. Amongst these source variables, some of them are reachable by the

definition of assertion while some of them do not. Furthermore, the possible existence of multiple valid paths may jeopardise the goal of path singularity. Thus, at most one source is needed at the presence of multiple valid source variable, the selection can abide by the comparison of the shortest distance from the entry point to the phi-node.

Very similar to the computation of the Floyd-Warshall algorithm, the comparison of distances carried by source variables is an iteration problem as well. The relation $ShortestTracePhiArgs(x : Variable, arg : number, miny : Variable, distance : number)$ represents a case when $x = phi(y_1, y_2, ...)$, x is the variable assigned at phi-node, $miny$ is the source variable with shortest $disance$ from entry to where x is defined(the phi-node) at arg_th iteration. arg is the index of iterations and should end at the number of source variables when the solution is optimal across all sources.

The base case for the phi-node is when only consider the first argument, which may result in two cases: one is the first argument will be taken as the sub-optimal at the first iteration when it is backward reachable. While the other case when the first argument is not backward reachable, it is still considered as the "sub-optimal", but act as a place holder since the distance is now $n + 1$, which is the maximum distance for a path in acyclic graph of n nodes. In this case, any reachable subsequent argument should be able to overwrite this placeholder.

```
1  .decl ShortestTracePhiArgs(x:Variable, arg:number,
2  miny: Variable, distance: number)
3  % Base Case:
4  % First argument is backward reachable
5  ShortestTracePhiArgs(x, 1, y, d1+d2) :-
6     ShortestTrace(u, _, d1),
7     VarDef(y, u),
8     PhiArgs(x, y, 1),
9     VarDef(x, v),
10    fw.ShortestPath(u, v, d2).
11
12 % First argument is not backward reachable
13 % 10000... very large constant - better to use max int number
14 ShortestTracePhiArgs(x, 1, y, n+1) :-
15    fw.NumNodes(n),
16    PhiArgs(x, y, 1),
17    !BackwardReachable(y).
```

The recursive rules can split into three cases, one is when the distance carried by the next argument is smaller than the previous sub-optimal distance, and then the best argument will be replaced by the next argument. When the next argument is not backward reachable, or the distance is greater than the previous argument, the sub-optimal solution remains unchanged and increments the iteration index.

```
% Shortest path of the next argument is larger than currently found
% so far.
ShortestTracePhiArgs(x, y, k+1, ymin, d) :-
   ShortestTracePhiArgs(x, _, k, ymin, d),
   PhiArgs(x, y, k+1),
   BackwardReachable(y)
   VarDef(y, u),
   VarDef(x, v),
   ShortestTrace(u, _, d1),
   fw.ShortestPath(u, v, d2),
   d <= d1 + d2.

% Shortest path of the next argument is smaller than currently found
ShortestTracePhiArgs(x, y, k+1, y, d1+d2) :-
   ShortestTracePhiArgs(x, _, k, _, d),
   PhiArgs(x, y, k+1),
   BackwardReachable(y),
   VarDef(y, u),
   VarDef(x, v),
   ShortestTrace(u, _, d1),
   fw.ShortestPath(u, v, d2),
   d > d1 + d2.

% There is no shortest path of next argument
ShortestTracePhiArgs(x, y, k+1, ymin, d) :-
   ShortestTracePhiArgs(x, _, k, ymin, d),
   PhiArgs(x, y, k+1),
   !BackwardReachable(y).
```

$PhiAssign(u, x, k)$ represents a phi-function assigned to x at the node u with k arguemnts. The $ShortestTracePhiArgs$ for x has the optimal argument $ymin$ after k_th iteration, so that the previous key point can be determined as the defintion node of $ymin$.

```
% choose the argument with the shortest path
ShortestTrace(u, v, d) :-
   PhiAssign(u, x, k),
   BackwardReachable(x),
```

```
5     ShortestTracePhiArgs(x, k, ymin, d),
6     VarDef(ymin, v).
```

$ShortestTrace$ relation identifies all the valid key points starting from the start point of the program until the end of the program, transferring the value along with the constant assignment, copy assignment and phi-node assignment. Essentially the $ShoretstTrace$ defines a set of nodes that can guarantee the correctness of the path in respect to the assertion, and the last step is to connect all the key points in $ShoretstTrace$ together to form a complete path.

The $TraceWitness(w : Node)$ relation is a set of nodes in CFG that can form a complete and valid path from the assertion to the entry point of the program if one exists, where each node is bound to have an edge to one of the other node in the relation. The generation of this relation can be viewed in the following rules by taking the key points in $ShortestTrace$ and filling the missing nodes in between via $ShortestPathWitness$ from the Floyd-Warshall algorithm.

```
1  .decl TraceWitness(w: Node)
2
3  % Base Case from Node u to the assertion
4  TraceWitness(w) :-
5      Assert(v, _, _),
6      ShortestTrace(v, u, _),
7      fw.ShortestPathWitness(u, v, w).
8
9  % Recursion until the start node
10 TraceWitness(w) :-
11     TraceWitness(v),
12     ShortestTrace(v, u,  _),
13     fw.ShortestPathWitness(u, v, w).
```

CHAPTER 7

Experiments

7.1 Use Case

Considering the simplicity of CCM, one might question its real-world application, whereas CCM is very efficient for expressing variables that bound to a finite state. For instance, Null Pointer Exception(NPE) in Java program is thrown when the program trying to access an object that points to null and may cause the program to crash if it is not handled properly. NPE is a perfect use case for the algorithms that we proposed because the variables at the point of the exception can only be either point to null or non-null, which can be represented as distinct immutable constants in CCM.

7.1.1 Gadget

To find a path for the provenance of a Null Pointer Exception in Java, We can devise a gadget that translates the original Java program into CCM and perform the pathfinding algorithm upon the translated program. The input for CCM can be translated from Java from the following rules

- **Constant Assignment:** For every variable initialisation, we can use 0 to denote null, and 1 for allocating heap memory(i.e. new ...()). This analysis only considers reference type in Java and omit other instructions involved with primitive type, since it is unlikely they will cause Null Pointer Exceptions.

$$x = \begin{cases} 0 & x = null \\ 1 & x = new() \end{cases}$$

- **Copy Assignment:** Every value trnasfer from one variale to another will be preserved as copy assignment in CCM. In order to enable inter-procedural analysis, variables returned from method calls(denote as $call()$) will also be included as Copy Assignment. Since the analysis

we are performing is context insensitive, this action will introduce some over-approximation because the method is unaware of the caller's context information and the return value will consider all the possible output from the same method in different contexts(i.e. different parameters).

$$x = y \begin{cases} x = y \\ x = call() \wedge return\ y \in call \end{cases}$$

- **Phi Assignment:** We assume the program is already in SSA before the gadget translation, but some additional phi nodes need to be included in this translation. Firstly of all, every method that contains parameters in its signature must include a phi node as its first instruction, taking all callers' actual parameters as phi arguments and assign to its corresponding formal parameter in the method signature(denote as $defcall()$). Secondly, for every load field instruction in the Java program, the gadget will look for the store instruction whose base object shares the same pointers in $VarPointsTo$ relation, and take those as phi arguments. Note that Array in java is a special case for loading and storing, but we can apply a similar technique as for fields. Since this transformation is dependent on $VarPointsTo$ relation, this may inevitably introduce some over-approximation into the analysis. Both strategies of inserting phi node will not break the rules in SSA form since we do not introduce any new variable along the way but simply stitches the existing variables together and replace the original assignment with another assignment. Thus, the properties of unique variable names and single assignment in SSA still holds.

$$x_j = \begin{cases} \phi(x_j^1, x_j^2, \ldots, x_j^i) & \forall_i\ call(x_1^i, x_2^i, \ldots, x_j^i) \wedge def\ call(x_1, x_2, \ldots, x_j) \\ \phi(x_1, x_2, \ldots, x_i) & \forall_i\ x_j = base_1.f \wedge base_2.f = x_i \wedge \\ & VarPointsTo(base_1) \cap VarPointsTo(base_2) \neq \varnothing \end{cases}$$

- **Assertion:** Assertion obviously should be where the Null Pointer Exception locates, and the form of it will be $assert(var == 0)$, where var is the variable that in question and 0 represents null.
- **Nop:** There are other instructions that included in the Java program such as method calls, return and if statements etc. Since their semantics are implied via conversions above and edge relation in Control Flow Graph, they are no longer required as part of the CCM and instead will be just represented by a Nop Node.

7.1.2 Digger & Doop

In this chapter, we will look into some actual program and apply the algorithm discussed in the previous sections to find the path from the start of the program to Null Pointer Exception reported by Digger. Digger is a Java Null Pointer Exception checker written in Datalog that finds the location of the exception at the static time. The execution Digger relies on Doop, a state of the art points-to analysis framework.(Smaragdakis and Balatsouras, 2015)

Doop provides points-to analysis with a range of flavours, including

- **Context Sensitivity:** Context sensitivity is a technique used in points-to analysis that can greatly enhance the analysis precision while maintaining its scalability. It proffers the context information to variables so they can be treated accordingly based on their context.
- **Reflection:** Reflection is an excessively used API in JDK that can determine and modify the behaviour of methods, classes at runtime. Doop has this feature covered by creating a special object for the variables assigned by reflection.
- **Flow Sensitivity:** Doop can enable flow sensitivity in Soot, a static analysis framework built by Mcgill University, Canada.(Vallée-Rai et al., 1999) Specifically, it converts each variable in Jimple to Static Single Assignment(SSA) form. By doing so, each SSA variable will only be assigned once, and the points-to analysis will be more precise from its consideration of instruction order.

In Doop, jar file (compiled from Java if not provided) will be translated to an intermediate representation(Jimple) by embedded Soot, and then the fact writer in Doop will extract the information of the program from Jimple to fact files. All the facts are then used by Doop's points-to analysis as well as Digger analysis and executed by Soufflé. The NPE results reported include the variables, instructions and code lines that will cause null pointer exception. The execution flow for Digger can be illustrated in the following flow chart

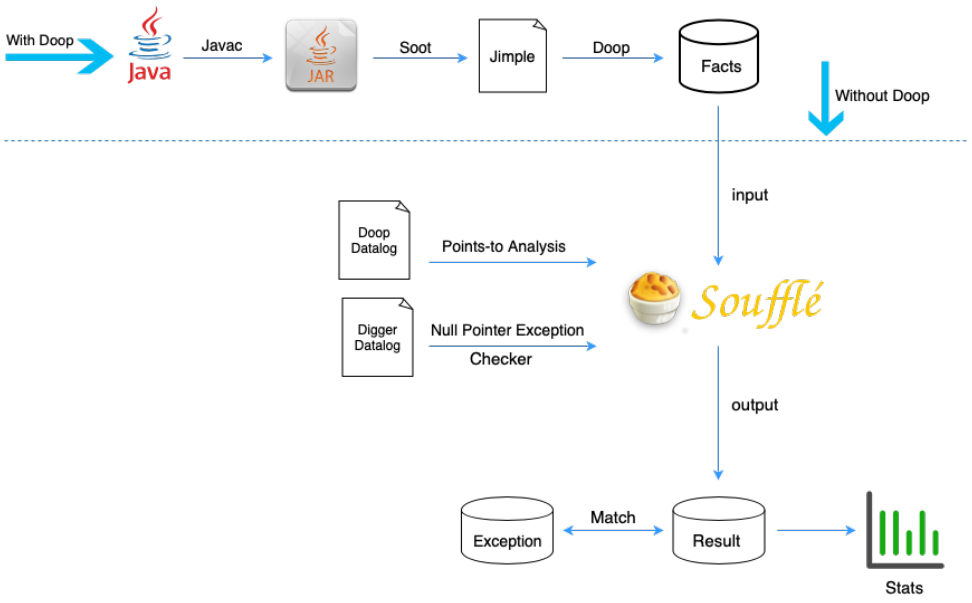


FIGURE 7.1. Digger Execution Flow

7.1.3 Case Study

The Java program below contains a Null Pointer Exception at line 13. By looking into the program, it is not hard to find the value causes NPE returned from the method call $returnNull$ in class NPE, which returns null when the parameter passed is null.

Main.java

```
1  class NPE{
2     public String returnNull(String param){
3        String res = new String();
4        if(param == null){
5           res = null;
6        }
7        return res;
8     }
9  }
10 public class Main {
11    private static void NPECall(NPE npe, String str){
12       String result = npe.returnNull(str);
13       result.toString();
14    }
```

```
15
16     public static void main(String[] args) {
17         NPE npe = new NPE();
18         NPECall(npe, null);
19     }
20 }
```

If we try to run this program, the stack trace printed indicates that the Null Pointer Exception located at line 13 in $Main.NPECall$ and called by line 18 in $Main.main$ as expected. The goal of this case study is to replicate the same result as in stack trace but instead in static time.

Stack Trace

```
Exception in thread "main" java.lang.NullPointerException
    at Main.NPECall(Main.java:13)
    at Main.main(Main.java:18)
```

First, we pass the Java program to Doop, and the translation to Jimple is done within Doop via Soot, the Jimple program looks like the following

NPE.class

```
1  class NPE extends java.lang.Object{
2
3      void <init>(){
4          NPE this#_0;
5          this#_0 := @this: NPE;
6          specialinvoke this#_0.<java.lang.Object: void <init>()>();
7          return;
8      }
9
10     public java.lang.String returnNull(java.lang.String){
11         NPE this#_0;
12         java.lang.String param#_0, $r0, res#_3, res_$$A_1#_5, res_$$A_2#_6;
13         this#_0 := @this: NPE;
14         param#_0 := @parameter0: java.lang.String;
15         $r0 = new java.lang.String;
```

```
16        specialinvoke $r0.<java.lang.String: void <init>()>();
17        res#_3 = $r0;
18
19  (0)   if param#_0 != null goto label1;
20
21  (1)   res_$$A_1#_5 = null;
22
23     label1:
24        res_$$A_2#_6 = Phi(res#_3 #0, res_$$A_1#_5 #1);
25
26        return res_$$A_2#_6;
27    }
28 }
```

Main.class

```
1  public class Main extends java.lang.Object{
2
3     public void <init>(){
4        Main this#_0;
5        this#_0 := @this: Main;
6        specialinvoke this#_0.<java.lang.Object: void <init>()>();
7        return;
8     }
9
10    private static void NPECall(NPE, java.lang.String){
11       NPE npe#_0;
12       java.lang.String str#_0, result#_12;
13       npe#_0 := @parameter0: NPE;
14       str#_0 := @parameter1: java.lang.String;
15       result#_12 = virtualinvoke npe#_0.<NPE: java.lang.String
             returnNull(java.lang.String)>(str#_0);
16       virtualinvoke result#_12.<java.lang.String: java.lang.String toString()>();
17       return;
18    }
19
20    public static void main(java.lang.String[]){
21       java.lang.String[] args#_0;
```

```
22          NPE npe#_17, $r0;
23          args#_0 := @parameter0: java.lang.String[];
24          $r0 = new NPE;
25          specialinvoke $r0.<NPE: void <init>()>();
26          npe#_17 = $r0;
27          staticinvoke <Main: void NPECall(NPE,java.lang.String)>(npe#_17, null);
28          return;
29      }
30  }
```

The translation performed in Jimple breaks down lines with multi-instructions into single instruction on its own, eliminates the ambiguity of instructions for analysis by doing so. It also converts the original Java program to SSA form , which is the base for the TraceWitness algorithm, by reassigning variable names(the actual names are also prefixed with class/method names to keep the uniqueness, but it is omitted here for the simplicity), and introducing phi node at the converge point (line 24 in NPE.class). The representation of the Jimple program in Control Flow Graph should look like below.

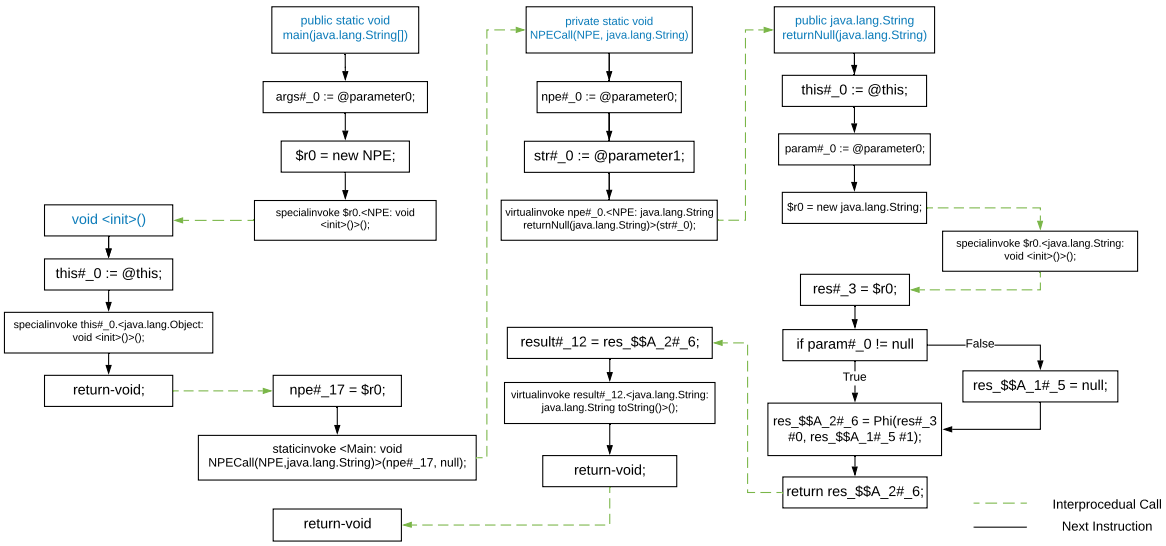


FIGURE 7.2. Control Flow Graph in Jimple

The gadget we devised then takes this CFG and convert it to the CCM that is the actual input for the TraceWitness algorithm, the conversions made in the process including inserting an additional phi node as the first instruction for parameters mapping, and other conversions to CCM specific style as described

in the gadget section above. The Control Flow graph in CCM output from gadget should look like the graph below.

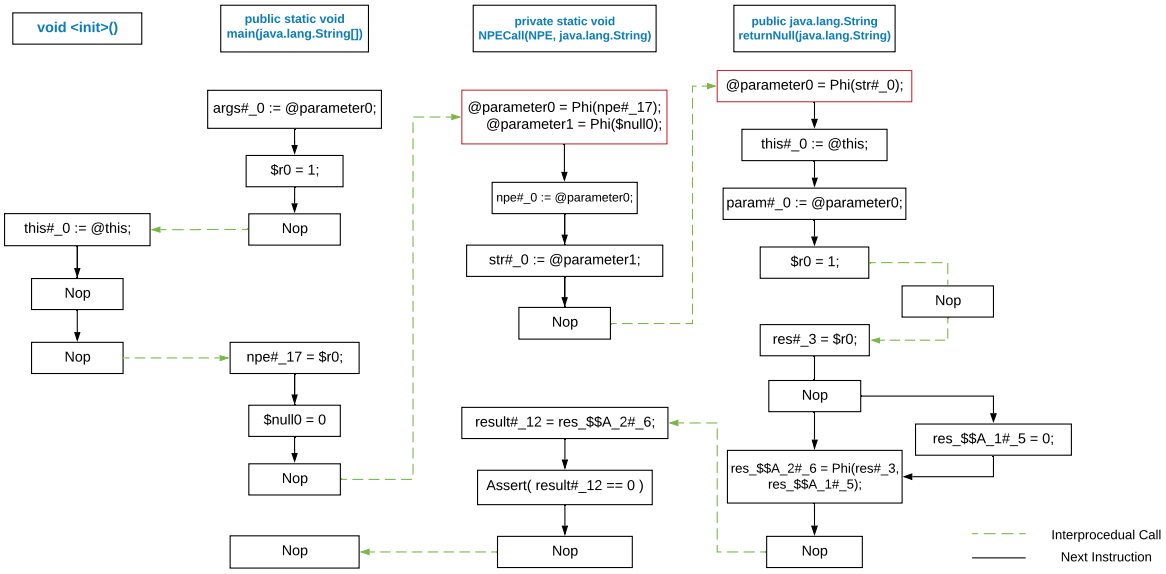


FIGURE 7.3. Control Flow Graph in CCM

Now that every instruction in the CFG is under the syntax of CCM, the TraceWitness algorithm can then be applied. $PruneVars$ relation first will look for the variables assigned to a constant that is not equivalent to the assertion, so in our program, it will kill off all the variables that are assigned to 1, which is <Main: main>/$r0 and <NPE: returnNull>/$r0. The relation recursively adds variables via all copy assignment originate from the existing pruned variables, as well as the variable in Phi assignment if all the Phi arguments are pruned. Besides, variables that without any assignment will also be pruned since this analysis does not include JDK library, and some variables are assigned from on the return value in JDK method calls, therefore without assigned value.

The unpruned variables are then gathered in $BackwardReachable$ starting from the assertion, indicates the trace of null values from some null source to the variable in the assertion. For our program, the variables in $BackwardReachable$ are

- **<Main: main>/$null0**: Assigned to 0 in Constant Assignment(null Source)
- **<NPE: NPECall>/@parameter1**: Assigned to <Main: main>/$null0 in Phi Assignment

- **<NPE: NPECall>/str#_0**: Assigned to <NPE: returnNull>/@parameter1 in Copy Assignment
- **<NPE: returnNull>/@parameter0**: Assigned to <NPE: NPECall>/str#_0 in Copy Assignment
- **<NPE: returnNull>/param#_0**: Assigned to <NPE: returnNull>/@parameter0 in Copy Assignment
- **<NPE: returnNull>/res_$ $A_1#_5**: Assigned to 0 in Constant Assignment
- **<NPE: returnNull>/res_$ $A_2#_6**: Assigned to <NPE: returnNull>/res_$ $A_1#_5 in Phi Assignment
- **result#_12**: Assigned to <NPE: returnNull>/res_$ $A_2#_6 in Copy Assignment(Assertion)

Since the variables are all in SSA form, which means they can only have one single assignment and their assignments are always before their usage. The $ShortestTrace$ relation that is generated based on this by finding all the assignment node for variables in $BackwardReachable$, and those are a set of fixed nodes to traverse in order to hold the condition in the assertion. For the program we are analysing, it has the following nodes to be fixed.

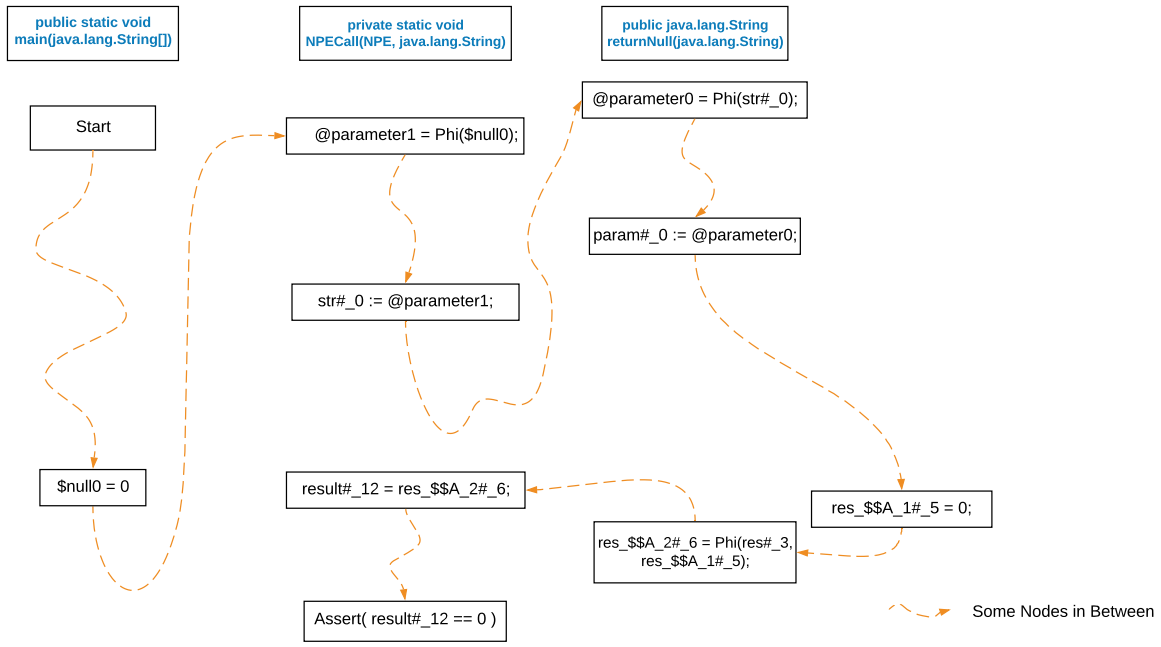


FIGURE 7.4. Shortest Trace

It is important to note that the dashed line between nodes in the graph is drawn based on the $ShotestPath$ from Floyd-Warshall algorithm based on the distance from $Start$ node, since the nodes in $BackwardReachable$ are unordered, and $ShotestPath$ provides a way to sort out the nodes by distance. Even though it does not apply in this particular case, but there is another purpose of considering the $ShortestPath$ relation. It is rather common that the null source for the variable in assertion is not single and it can be originated from one of the multiple variables. This implies there are multiple sets of key nodes that exist with some intersections near the assertion node, and we only need to select the one that can get us to assertion the fastest.

With all the key points fixed, we need to leverage the $ShortestPath$ relation again, but instead of laying out the hopping points in the graph, it is now used for finding out the actual path between each hopping points. Because Floyd-Warshall algorithm does not only provide the shortest distance between every pair of nodes in the graph but also the actual path between them. The final path computed are our trace witness for the program, as shown below.

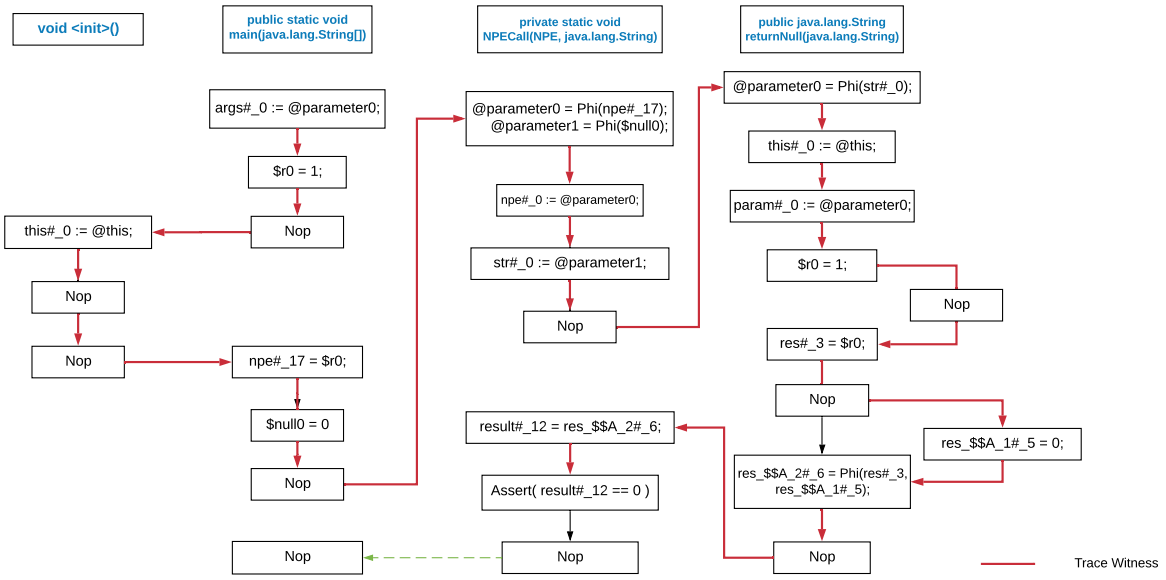


FIGURE 7.5. Trace Witness

Every node in the trace witness is an instruction in Jimple, and it can be mapped back to the actual code lines by consulting Doop. And the final output for the analysis is very much similar to the produced Stack Trace, including lines and file names.[1]

```
Trace Witness Lines:
-1 - Main.java
1  - Main.java
3  - Main.java
4  - Main.java
5  - Main.java
7  - Main.java
12 - Main.java
13 - Main.java
17 - Main.java
18 - Main.java
```

7.2 Emperical Study

7.2.1 Motivation

Digger as a static program analysis tool, it suffers from inherently high false-positive rate.(Landi, 1992) It is difficult to trace the reported bug or identify whether it is a false positive. Trace Witness does not only provides a visualisation of bug traces, but it can be also be utilised to filter the false positive bugs reported from Digger, for the ones with trace can be treated as true positive and the ones without as false positive.

In this section, we will conduct an empirical study on some open-source Java programs to find valid traces of Null Pointer Exceptions reported by Digger to fully evaluate

(1) the performance of our algorithm in large-scale programs
(2) the application of the algorithm in false-positive invalidation.

[1] The 1 and -1 are instructions from void init() that does not have the actual line in Java code

7.2.2 Experimental Setup

To fully evaluate the effectiveness of out algorihtm, the emperical experiment will be conducted on a list of open source Java programs. All subjects of the experiment are obtained as Jar files from SourceForge(`http://www.sourceforge.net/`), at versions where Null Pointer Exceptions are reported. The programs are chosen in varaious sizes to better evaluate the preformance in relation to the size. The enlisted programs include JSP-3.0.0, JBoss-1.1.1 and Sling-11 as shown in Table 7.1. The metrics of program size includes number of classes, number of method and method calls, number of variables, number of object creation sites, number of instructions in Jimple and number of main methods, with all statistics reported from Doop and Soot.

Benchmark	**JSPWiki-3.0.0**	**JBoss-1.1.1**	**Sling-11**
Class	490	143	19
Method Call	931	1049	284
Methods	345	390	121
Variables	2230	2697	664
Object Creation Sites	7628	1274	377
Instructions	90556	22134	5652
Main Methods	8	1	1

TABLE 7.1. Problem Size

For each subject, our analysing tool will first pass it to Digger and the trace witness analysis will take every Null Pointer Exceptions reported by Digger as assertion, and try to produce a trace from the assertion back to the entry point of the program, which we can safely assume as the first instruction in main method. The consider of number of main methods as problem size is because certain instructions in a program such as JSPWiki are unreachable from one main method but is reachable at others. Taking only one main method as start point for those programs will potentially render valid trace as non-existent, so instead the experiment will run all assertions for every main method exists in the program.

The tasks of each run includes

- Soot-3.3.0 facts generation
- Doop Points-To Anlaysis in context-insensitive mode
- Digger Null Pointer Exception Checker
- Gadget Translation
- Shortest Path Computation

- Trace Witness Generation

All tasks are run by Soufflé-1.6.2 compiler in 6 threads except for Soot. All experiments are conducted on Fedora Server Version 30 with 187GB RAM and 32 Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz.

7.2.3 Results & Discussion

The experiment we are conducting is primely to evaluate the performance of our algorithm in a large-scale program environment. In particular, we are evaluting the runtime throughout stages in Gadget Translation, Shortest Path Computation and Trace Witness Generation. Since the majority of our algorihm is graph-based, the graph size are taken as the main factor that can affect our runtime performance, the graph size is measured in number of nodes and edges where nodes reprensent instructions in the CFG of the program and edges as transitions between instructions. Consider large number of instructions in certain programs, nodes in CFG that are unreachable from the given start node or to the assertion node will be pruned so that the graph size will be greatly reduced with affecting the correctness of our output.

Our program is performed on top of the results obtained from Digger, the experiment will take all Null Pointer Exceptions reported by Digger and validate the ones with valid trace to the bug point. Valid traces computed by our program are measured by its length in nodes(instructions) and length of code lines. The correctness of the trace are not in consideration of this experiment at this stage given that traces in large programs are often too long to be inspected manually, and the actaul Null Pointer Exception reported by execution may vary from the ones we reported, which causes the evaluation on correctness even mroe difficult.

Results for the experiment can be viewed in the following table

Program	NPE	Assertion (File: Line)	Nodes	Edge	Gadget Runtime	Shortest Path Runtime	Trace Witness Runtime	Trace Length(Nodes)	Trace Length(Lines)
JSPWiki	86	Import: 167	2975	3295	3m 0s	13m 27s	6m 50s	11	7
		Import: 161			3m 1s	13m 25s	6m 31s	6	3
		WikiEngine: 580			2m 41s	11m 19s	5m 56s	236	106
		WikiEngine: 581			3m 1s	13m 28s	6m 32s	329	107
		WikiEngine: 582			2m 59s	13m 40s	6m 34s	331	108
		WikiEngine: 587			3m 0s	13m 39s	6m 31s	275	88
		WikiEngine: 618			3m 1s	13m 26s	6m 36s	275	88
		WikiEngine: 614			3m 0s	12m 57s	6m 32s	275	88
		WikiEngine: 579			2m 58s	13m 37s	6m 33s	325	105
		Tag: 115	1434	1778	29s	4m 54s	54s	327	89
		Tag: 111			28s	5m 9s	52s	235	89
		Tag: 107			28s	5m 0s	52s	229	87
		JspParser: 105			28s	5m 18s	52s	232	88
		JspParser: 933			28s	5m 9s	53s	232	89
Sling	8	Main: 557	1879	2116	42s	11m 45s	2m 46s	187	45
		Main: 566			45s	12m 19s	2m 55s	197	48
JBoss	19	PathUtils: 92	5223	5980	14m 39s	92m 57s	21m 28s	222	72
		ModuleLoader: 208			14m 35s	92m 50s	21m 15s	106	40
		ModuleLoader: 281			14m 29s	94m 21s	22m 34s	138	48

TABLE 7.2. Trace Witness Result

From the results table, we can observe that all experimental subjects detected traces for their reported NPEs by Digger, with 14 out of 86 (16.2%) in JSPWiki, 2 out of 8 (25 %)in Sling and 3 out of 19 (15.7 %) in JBoss. The Null Pointer Exceptions without a valid traces are either because all nodes are pruned due to the unreachability or no trace can be found while the condition in assertion holds true. For example, JSPWiki is the target program with the largest code base with 90556 instructions before pruning and 8 main methods as mentioned in Table 7.1. However, out of all 8 main methods, only from 2 of which the trace to the bugs can be found. Even though our experiments invalidate nearly 80 % of bugs reported by Digger, it is unknown that the deemed invalid bugs are actaully false positive. For example, the unreachable case can be explained by there the existence of other entrances to the program that we are not aware of such that a dependency on a third-party application.

Figure 7.6 shows how the runtime in three stages of our experiment relating to the graph size. Number of nodes are used to represent the graph size in this experiment. The graph representation are often sparse for the Control Flow Graph of a program since the majority of the nodes(instructions) are linked together singly and linearly. The nearly linear mapping between number of nodes and number of edges can justify the measurement of a graph size using only number of nodes, and it also can be proved in Table 7.2.

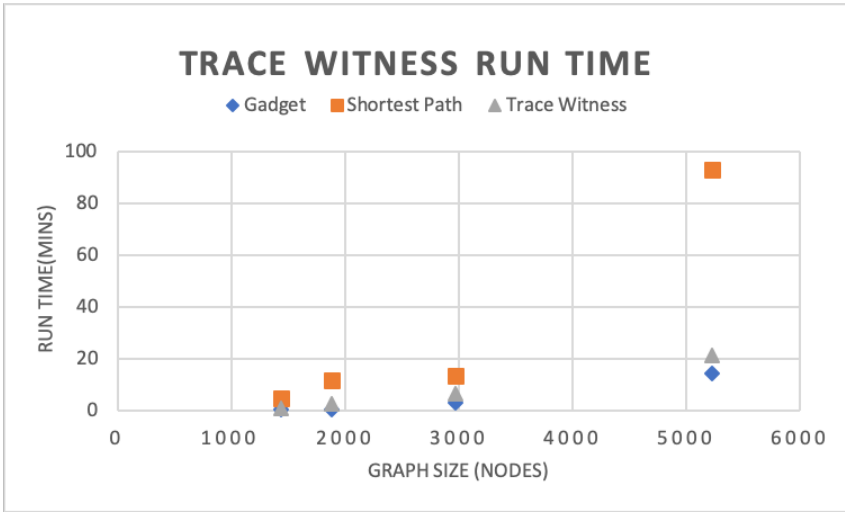


FIGURE 7.6. Trace Witness Runtime

Gadget Translation among all three processes takes the least portion of total runtime for all graph sizes, while the Shortest Path Algorithm always the slowest in terms of the performance. It is clear that the runtime grows with increasing graph size for all three processes, but the runtime of Shortest Path algorithm grows exponentially while others stay in linear. The runtime growing trend indicates that the Shortest Path algorithm will be the performance bottleneck once the graph is large enough. A potential future work for this research is to implement a more suitable Shortest Path algorithm to improve the performance in runtime.

The length of bug traces found across all three experimental subjects is shown in Figure 7.7. The trace length is measured in the number of nodes(instructions) and the number of code lines, which appear in a linear relation upon observing the figure. It can be inferred from the figure that the length of trace found the program does not vary much between different programs, so accordingly the size of a program has little impact on the bug trace length, which may belie what we normally believe.

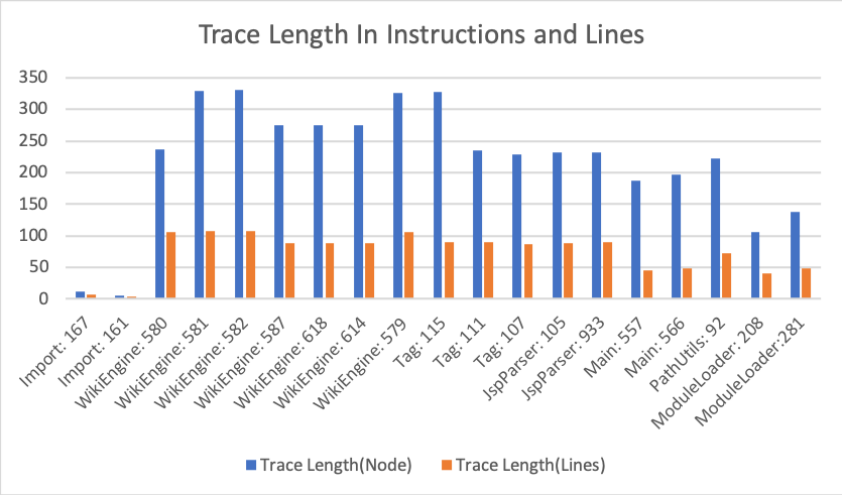


FIGURE 7.7. Trace Length in Instructions and Lines

This empirical study examined three open-source Java programs, with around 80% of bugs reported by Digger eliminated by our algorithm and the rest expressed with valid traces. It is certainly efficient for our algorithm to run on small programs, but as the program grows, a better Shortest Path algorithm may be needed to be implemented for the better performance.

CHAPTER 8

Conclusion & Future Work

In this work, we introduced Constant Copy Machine to weaken the semantics of a high-level Object-Oriented language. CCM only includes Constant Assignment and Copy Assignment in syntax to address the issue of undecidable pathfinding in a Turing machine. CCM is represented in a graph such that it can fully capture all the possible paths in the original Control Flow Graph. Finding a path in CCM is sufficient to deduce the path must also exist in the original OO language, and such a path can be materialised by mapping the trace back from CCM to OO language.

We also demonstrated Single Static Assignment as an efficient approach to provide the analysis of a flavour of flow-sensitivity. While the implementation of SSA conversion in imperative language can be laborious, we propose the implementation in Datalog with the edge of simplicity utilising Dominance Frontier relation between nodes in the graph.

Finally, Floyd-Warshall algorithm as a solution for the shortest path between every pair in the graph is implemented in Datalog as the consultant for the Trace Witness algorithm to address the existence of multiple possible valid paths in the program. Trace Witness algorithm then provides the solution for finding chains of Data dependencies and the chain with assertion variable is the trace that we sought after. The concrete path can then be materialised from the Shortest Path results obtained beforehand.

This paper also includes a case study to provide a holistic view of how the algorithm can be applied to find the trace of Null Pointer Exception in a Java program. We have also extended the experiment to a list of large-scale Java open source programs. The experiment is aiming to visualise and validate the bugs reported by Null Pointer Exception checker Digger. With the result of eliminating around 80% bugs eliminated, our algorithm is proven to be efficient.

The research on the Trace Witness problem can be extended in the following areas

- **Better Shortest Path Algorithm:** Floyd-Warshall algorithm has a great advantage of being simple and easy implementation in Datalog, however, it also faces the challenges in terms of the performance when applied in a large graph as discussed in Chapter 7. When considering a Control Flow Graph as the target graph for Shortest Path problem, it is an unweighted directed graph, and a selection of algorithms such as Dijkstra algorithm and Bread-First Search can be considered for an advance in runtime performance. However, the main challenge may be faced when implementing those algorithms in Datalog, keep track of the state is an easy task in an imperative language, but it will result in an unstratified rule in Datalog.
- **Other Applications of Trace Witness Algorithm:** Indeed, Null Pointer Exception is a successful use case for our Trace Witness Algorithm given this type of exception confines variables to two states, either null or not null. Our algorithm should also be applicable for other defects that can abstract the state of variables to a finite set of states.

Bibliography

B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11. ACM, New York, NY, USA.

Lars Ole Andersen. 1994. Program analysis and specialization for the c programming language. Technical report.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39.

Francois Bancilhon. 1986. On knowledge base management systems: Integrating artificial intelligence and d atabase technologies. chapter Naive Evaluation of Recursively Defined Relations, pages 165–178. Springer-Verlag New York, Inc., New York, NY, USA.

Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. 2011. The flow-insensitive precision of andersen's analysis in practice. In Eran Yahav, editor, *Static Analysis*, pages 60–76. Springer Berlin Heidelberg, Berlin, Heidelberg.

Marc Brandis and Hanspeter Mössenböck. 1994. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16:1684–1698.

Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. 1998. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881.

Keith Cooper, Timothy Harvey, and Ken Kennedy. 2006. A simple, fast dominance algorithm. *Rice University, CS Technical Report 06-33870*.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

Steven Dawson, C. R. Ramakrishnan, and David S. Warren. 1996. Practical program analysis using general purpose logic programming systems—a case study. *SIGPLAN Not.*, 31(5):117–126.

E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

Robert W. Floyd. 1962. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–.

Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. *SIGPLAN Not.*, 36(5):254–263.

Alfred Horn. 1951. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21.

Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206. ACM, New York, NY, USA.

John Lakos. 1996. *Large-scale C++ Software Design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

William Landi. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337.

Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to analysis with efficient strong updates. *SIGPLAN Not.*, 46(1):3–16.

Ben Liblit and Alex Aiken. 2002. Building a better backtrace: Techniques for postmortem program analysis. Technical report, Berkeley, CA, USA.

Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. Pse: Explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes*, 29(6):63–72.

David A. Patterson and John L. Hennessy. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Reese T. Prosser. 1959. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138. ACM, New York, NY, USA.

G. Ramalingam. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471.

John H. Reif and Harry R. Lewis. 1977. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 104–118. ACM, New York, NY, USA.

Thomas Reps. 1997. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, pages 5–19. MIT Press, Cambridge, MA, USA.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61. ACM, New York, NY, USA.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27. ACM, New York, NY, USA.

Barbara G. Ryder. 2003. Dimensions of precision in reference analysis of object-oriented programming languages. In Görel Hedin, editor, *Compiler Construction*, pages 126–137. Springer Berlin Heidelberg, Berlin, Heidelberg.

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*,

CC 2016, pages 196–206. ACM, New York, NY, USA.

J. C. Shepherdson and H. E. Sturgis. 1963. Computability of recursive functions. *J. ACM*, 10(2):217–255.

Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69.

Manu Sridharan and Stephen J. Fink. 2009. The complexity of andersen's analysis in practice. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 205–221. Springer-Verlag, Berlin, Heidelberg.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for java. *SIGPLAN Not.*, 40(10):59–76.

Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41. ACM, New York, NY, USA.

Linda Torczon and Keith Cooper. 2011. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press.

Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. 2005. Comparing bug finding tools with reviews and tests. In Ferhat Khendek and Rachida Dssouli, editors, *Testing of Communicating Systems*, pages 40–55. Springer Berlin Heidelberg, Berlin, Heidelberg.