FACULTY OF ENGINEERING



September 2015

Promoter: Prof. Dr. Coen De Roover

Advisors: Dr. Laure Philips

Dr. Jens Nicolay

Thesis submitted in partial fulfilment of the requirements for the Master :

Master of Science in Applied Sciences and Engineering: Applied Computer Science

Xinran AI

Tier splitting using Static analysis

Abstract

Acknowledgments

Content

1

[Introduction 5](#_Toc31504)

[1.1 Problem statement 5](#_Toc18012)

[1.2 Contribution 6](#_Toc3667)

[1.3 Road map 7](#_Toc32523)

[Program Slicing 8](#_Toc16509)

[2.1 Static program slicing 8](#_Toc14599)

[2.1.1 Introduction 8](#_Toc29108)

[2.1.2 Data flow analysis 8](#_Toc5626)

[2.1.3 Static slicing on PDG. 10](#_Toc24240)

[2.2 Other approach of program slicing 12](#_Toc17063)

[2.2.1 Methods for static program slicing 12](#_Toc106)

[2.2.2 Dynamic and conditional program slicing 16](#_Toc24155)

[2.3 Applications of program slicing 19](#_Toc16969)

[Abstract Interpreter 20](#_Toc4843)

[Program Slicing on State Graph 21](#_Toc2741)

[Validation 22](#_Toc2425)

[Conclusion 23](#_Toc12595)

1

Introduction

The typical three-tier architecture of web application includes client tier, server tier, and data tier. Figure 1 is an web application structure conforms to three-tier architecture introduced in [6].

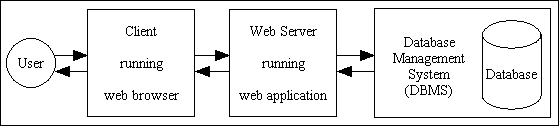


Figure 1 three-tier web application

Normally, client tier is represent for a presentation layer such as a browser installed on user’s computer, server tier is used to handle computation and event, data tier is act as an interface communicate to database system. The traditional development method is respective each tier in development and combination of each tier is required. On contrast with traditional method, tier splitting of web development is a new concept in the domain of web application development. In distributed programming, tier splitting is split a single-tier application into multi-tier application[15]. This approach used on web application try to blur the distinction between different tiers of web development such that the development is more straightforward.

Beside investment in novel programming language to realize tier splitting, extend existing technology with distribute feature, such as JavaScript to cater for tier splitting seems more beneficial. To enable tierless programming for the JavaScript language, we use static analysis based on abstract interpretation. On top of that, we need to split the actual tiers of the program which can be achieved by program slicing.

**1.1 Problem statement**

Today people can’t live without internet. The applications built in internet provides a lots of functionalities to convenience for people’s life. This kind of changes requires web applications provide huge information and services, and also be able to handle more interactions with user. In this case, developers have to create so-called rich internet applications, where the client contains program logic as well. Developing such

1<https://soft.vub.ac.be/~lphilips/jspdg/stip/stip-web/stip.html>

applications in the traditional three-tier architecture increases the complexity of web development, because each tier comes with its own technology stack. This leads to complex glue between each technology of the different tiers. Normally, a combination of HTML, JavaScript and CSS is used for client tier and often together with Ajax and jQuery.The server tier itself is written in another programming language, often depending on which web framework is chosen by the programmer.

Tierless programming languages reduce this complexity by developing a web application in only one programming language. In these languages, a preprocessor or the runtime of language can split source code into client, server and sometimes a database tier. The communication between the different, distributed tiers is also handled by these languages. However, investment on novel programming languages is not an efficient way to relieve developers from handling the described complexity.

By allowing tierless programming in a general-purpose language the developer can profit from existing tool support for that language. JavaScript is an event-driven language that runs on the client tier and has been already extended to server tier, thanks to Node.js. By using program slicing mechanism, the tier-splitting tool called STIP.js1 is able to split a tierless JavaScript program into client and server tier.

Program slicing[1, 2] is a mechanism to get a subset of the program on a graph presentation of program. Such a subset is a valid program itself and can be executed

independently. This mechanism is widely used to analyse and understand the behavior

of code[11,12,13,14].

Currently, the process of generating a tier split by the STIP.js tool consists of two parts. First, a State Transition Graph (STG) is generated based on abstract interpretation for JavaScript. Concretely, the Jipda2 framework is used for this analysis. This graph is a description of possible states the program can transition to. Then STIP.js constructs a Program Dependency Graph (PDG) out of the STG. PDG is constructed on data and control dependency[7]. Data dependency is used to represent for data flow relationships of a program, whereas, control dependency is corresponding the control flow relationships. A PDG is used as the input for program slicing algorithms. From the selected node in PDG, slicing algorithm is backward traversing on the basis of data dependency and control dependency, resulting in two subsets of the tierless program: the client and server program. To get the tier splitting code, the developer only needs to annotate the tierless JavaScript code with @client and @server.

**1.2 Contribution**

This dissertation introduces JipdaSlicer, a new program slicing algorithm based on the Jipda abstract interpreter framework. When slicing a program, this algorithm works on a State Transition Graph directly, instead of first constructing a Program Dependence Graph and using the classical slicing algorithms on this graph. JipdaSlicer accepts a variable pair containing a line number and a variable identifier from source JavaScript code, which are together the slicing criterion. This criterion is a representation of a point in code where the users want to cut off code not involved in this point. Based on this criterion a backward slice is computed; this is the set of instructions in the program that influence the slicing criterion. The current algorithm supports a subset of JavaScript, which we discuss in more detail in Chapter 4. To validation the new solution, we test on different cases. These cases test if JipdaSlicer can compute a correct program slice or not from different point of view. As we show, the result of JipdaSlicer is satisfactory.

**1.3 Road map**

The rest of this document is structured as follows: in Chapter 2, we describe program slicing. We introduce the basic concept of static program slicing in this chapter first and then give a general introduction of slicing on PDG. In the last of this chapter we introduce the development and different use of program slicing technology. Chapter 3 introduces abstract interpretation. Chapter 4 is about program slicing on the state transition graph. We explain our implementation of new algorithm in this chapter. Chapter 5 validates the new algorithm, by comparing the result to that of a current program slicing tool for JavaScript. This dissertation ends with a conclusion chapter, including work summary and future work.

2<https://github.com/jensnicolay/jipda/>

2

Program Slicing

In this chapter, we introduce program slicing. The original definition of program slicing is proposed by Weiser [1,2] mainly used for program debugging and maintenance. Weiser defined a slice of program *S* is a reduced program obtain from program *P* by removing statements based on data flow and control flow analysis. *S* can be executed independently and keeps the same behavior when it’s involved in *P*. Program slicing is the process to obtain the slice *S*.

So far, program slicing has been extended in variety method to adjust different use and properties in different applications. Tip [3] conducted a survey about all these types of program slicing technologies. Static slicing is distinct from dynamic slicing without input assumption. The survey summarizes static and dynamic slicing, starting from the basic algorithm, procedures, unstructured flow and composite data and gives an overview of slicing technologies applied in different application areas.

This thesis focuses on static program slicing. First in this chapter, we introduce the basic notions and character of static program slicing and Weiser’s data flow algorithm. Then we explain slicing on a program dependence graph (PDG). Lastly, we discuss the development of program slicing technology.

**2.1 Static program slicing**

2.1.1 Introduction

A program slice, conform to Weiser’s theory [1], has the following two properties:

1. A slice S of program P, is obtained from a specific slicing criteria denoted as a pair of value *<i, V>*, where *i* is the line number of statement in *P*, and *V* is set of variables defined or used at *i*.
2. A slice S can be obtained by deleting zero or more statements from program *P*. Meanwhile, *P* and *S* must behave the same with respect to *<i,V>*.

Figure 2.1(a) and 2.1(b) is an example to illustrate Weiser’s program slicing. This example is supporting Weiser’s theory from two aspects: slice *S* on slicing criteria <10,{abc}> is achieved by deleting several statements; looking at the 10th statement, executing the slice S and the program P both lead to the same result on variable *ac*.

2.1.2 Data flow analysis

Beside, Weiser also put forward some constructive opinions in [1]. Weiser’s first slicing theory establishes on graph representation of program. Each node in a graph represents for a statement in program. Weiser defined a flowgraph *G*=<*N*,*E*,*no*> where *N* is the nodes, *E* is edges in set of *N×N* indicates the existing path from one node to another, an initial node *n0* as single entry where the program start. A hammock graph structure

1

2

3

4

5

6

7

8

9

10

var a=1;

var b=2;

var c=3;

var ab=0;

if (a>b){

ab=a-b;

}

else ab=a+b;

var ac = a + c;

var abc = ac + c;

var a=1;

var b=2;

var c=3;

~~var ab=0;~~

~~if (a>b){~~

~~ab=a-b;~~

~~}~~

~~else ab=a+b;~~

var ac = a + c;

var abc = ac + c;

Figure 2.1(a) a JavaScript program P; 2.1 (b) get slice S out of slicing criteria <10,{abc}>

*G=<N, E, n0, ne>* where *ne* is an exist nodewhere the program terminate is extensible definition of flowgraph. Another useful definition is *REF(n)* and *DEF(n)*. *REF(n*) indicates the set of variables whose value are used at statement n, *DEF(n)* indicates the set of variables whose value are changed at statement n. In [1], program slicing is done by flow datatype analysis [5].

Base on the original notions, Weiser proposed a data flow algorithm for program slicing in [2]. After discussinh about how to slice a program by using reader’s intuitive understanding on a flowgraph, Weiser introduced a method to find slices by tracing backwards according to dataflow analysis.

1. Introduction of flowgraph

Figure 2.2 is an example of slicing on flowgraph Weiser gave in [2]. From the figure we see that a flowgraph is an oriented graph with an initial node. The edge (*n*, *m*) in flowgraph indicates the execution progress can be from *n* to *m*, *n* is an immediate predecessor of *m*, and *m* is an immediate successor of *n*. A path of length *k* from *n* to *m* is all possible query on flow graph from node *n* to *m*. A node *n* is dominated of node *m* when *n* is on every path from *n0* to *m*. If *m* is on every path from *n* to the terminate node *ne* on flowgraph, *m* is a inverse dominator of *n*. Deleting statements in a flowgraph to calculate slices must ensure that there is no increase of immediate successor of statement during deletion. With this concerning, Weiser defined statement deletion as: a set of nodes with single successor can be seen a deleted group, for all predecessors of a deleted group, set deleted group’s unique successor as their new successor. The left part of figure 2.2 shows the result of removing statement in deleted group.

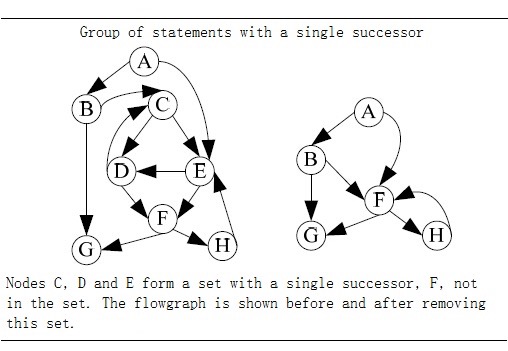


Figure 2.2 Deleting statement in a flowgraph[2].

1. Dataflow algorithm

Weiser’s dataflow algorithm finds program slices by iteratively calculating the set of the related variables of each node in the flowgraph. The calculation steps are as follows:

Input: the flowgraph of program *P*, and slice criteria *C=<i, V>*

Output: slice *S* of program *P* on slice criteria *C*.

1. Calculation the directly relevant variables and directly relevant statement.
   1. For node *n* and *m* in flowgraph, if there exist path from *n* to *m*, then the set of directly relevant variables *R[0,C](n)* is denoted as: all variables v such that either:
      1. *n = i* and *v* is in *V*: *R[0,C](n)=V.*
      2. *n* is an immediate predecessor of a node *m*: *R[0,C](n) = {v | v∈ R[0,C](m), v∉DEF(n)}{v | v∈ REF(n), DEF(n)R[0,C](m) ≠ ϕ}*



* 1. directly relevant statements is denoted as:

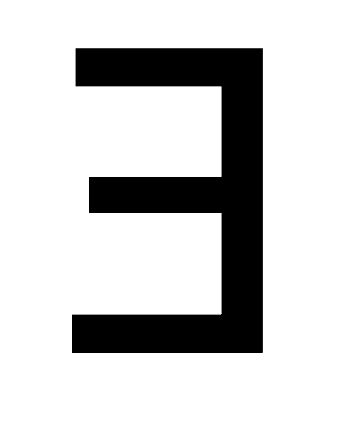
*S[0,C] = {N | DEF(n)R[0,C](n)≠ ϕ }*.



1. Iteratively calculate indirectly relevant variables and indirectly relevant statements.
   1. indirectly relevant variables set is denoted as *R[k,C](n)(k0)*, when calculate *R[k,C](n)*, we have to take account into control dependency. INFL(b) is a set used to represent for the statements influence on statement b, then:

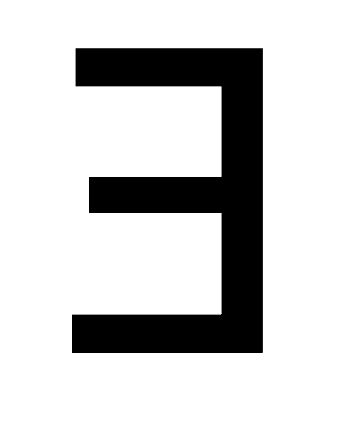


*R[k+1,C](n) = R[k,C](n){n|n∈R[k,C](n),b∈INFL(b)}*



* 1. Similarly for indirectly relevant statements S[k,C]:

*S[k+1,C] =* *{n|n∈R[k,C](n),b∈INFL(b)}{n| DEF(N)R[i+1,C](n)≠ ϕ}.*



1. Repeat step 2 until the size of set *S* doesn’t increase any more, and the statements in *S* consist of slice.

2.1.3 Static slicing on PDG.

Many program slicing approaches use a so-called program dependence graph (PDG)[4,7] as an intermediate representation of a program. Similar with a flowgraph, nodes in a PDG correspond to statements and control predicates of the program, and the edges correspond to data and control dependencies between nodes. Informally, when a variable is defined or appearing in one statement, another statement uses this variable without redefining it, then we say this dependency is data dependency. If one statement determines whether another statement will be executed, then the latter is control dependent on former. J.Ferrante et al. defined a PDG which can be use for static slicing of single-procedure programs[7]. According to their definition, the control dependency in PDG is constructed on top of a Control Flow Graph (CFG) associated with a Control Dependency Graph (CDG), and data dependency is identified through a Data Dependency Graph (DDG).

They follow the principle in [35,36] to define and construct CFG and post-dominator tree. A CFG describes the control flow of a program. A CFG is flowgraph augmented with a unique *Start* node represents for entry of the program and unique *Stop* node represents for exist of the program. Each node in CFG has at most two successors and they assume the attributes of outgoing edges get to the two successors is *“T”(true)* and *“F”(false)*. The *post-dominator* is defined as if every path from *n* to *Stop* contains *m*, then we can say *n* is post-dominated by *m*. If a node *X* is control dependent on *Y* in CFG, is must exist a path from *X* to *Y*, and *X* is not post-dominated by *Y*. In other word, if a statement *S’* is dependent on statement *S*, there will be two edges out of *S*, the “T” edge will lead to execute on *S’*, *“F”* edge will lead the program jump to another statement. *“F”* edges also ensure there always has an exist for loop in CFG. Figure 2.3 shows an example code with its corresponding CFG.

Start

14

13

12

10

F

T

F

T

**8**

**4**

**3**

**2**

**9**

**6**

**5**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

var absolute=function(x,y){

var x1=0;

var y1=0;

if(x>0&&y>0) {

x1=x;

y1=y;

}

else if(x>0&&y<0){

x1=x;

y1=0-y;

}

else {

x1=0-x;

y1=0-y;

}

return x1+y1;

}

16

Stop

(a)

(b)

Figure 2.3 (a)An example code;(b) control flow graph of (a).

In the figure, we use the line number to mark each node in CFG. Whether line 5 and 6 will be executed is control dependent on the predicate in line 4. Another information in CFG to get control dependences is in term of *region node*. This term can be understand as an entry of a block of statements with the same control conditions. In the example above, the prediction in line 4 can be seen a *region node* of statement in line 5 and 6. In addition, each block has a exist node. The control dependences can be identified by examining the edges (*A, B*) label with *“T”* or *“F”* in post-dominator tree. As the definition of control dependency, the examining is aim to skip the post-dominator of *A* which actually is the parent node of *A*. Integrate with two type of region node, we can get the CDG that contains the control dependencies inside a program.

Data dependent is defined as: there exists a path *k* from node *m* to node *n*, and a variable is defined at m and gets used at n without redefining it at any other node on path *k*, then we can say *n* is data dependent on *m*. DDG is a set of data dependencies between statements in a program. Their solution to build DDG is derived from [37,38] and perform data flow analysis[35] with additional assumption that every variable is initiated at entry. After constructing PDG by integrating the CDG and the DDG, a slice can be obtained by performing a backward traverse from an interested node in the graph, visiting all predecessors. In their later work[9], they state that slicing on a PDG is more accurate than the described, earlier method.

The below picture shows corresponding PDG of Figure 2.1(a). The gray node is the slice on <10,{abc}>. In the figure, solid line denotes for control dependences, dash line denotes for data dependences. Slicing starts by searching the node represents for line 10, and then backward traverse dependences edges from it and mark all the node you visited. In the first traversal, nodes represent for code “var b=2;” and “var ac=a+c;” will be visited. In step 2, backward traverse the dependences edges from the node you marked in first traversal. Iterative step 2 until all marked node on the path from slicing node to entry node.

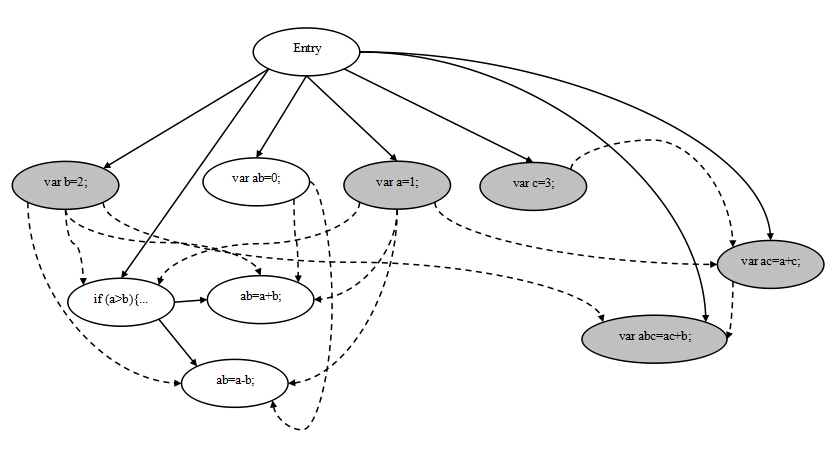


Figure 2.3 program dependency graph of program in figure2.1(a)

**2.2 Other approach of program slicing**

2.2.1 Method for static program slicing

The above approaches describe the simple case to compute slices of structured, single-procedure programs with scalar variables. The slice we get from backward static slice, is the statements has influence on the the slicing criteria. In the similar way, forward static slice[17] determine slice by computing set of statements are influenced by the the slicing criteria, it requires forward tracing on dependences. A statement is influenced by slicing criteria means the value computed in slicing criteria will effect on the statement or determine execution of it. In addition to this, program slicing also applies for inter-procedural program, unstructured control flow, composite variables and pointer, and concurrent program. The solutions to these problem will be discussed below.

***Procedures***

The main idea to solve inter-procedural static slicing is to construct a call-return structure to depict the interaction between procedures. Weiser’s approach [2,16] is able to approximate inter-procedural static slicing by taking account into *summary information* [8]: MOD(P) is a set of variables may be modified by procedure *P*, and variable may be used at P denoted as USE(P). The data flow algorithm is extended to identify a call effect with these *summary information*. Assuming a procedure *R* calls *P*, and *Q* is called by *P*, function Up(S) is defined as: a set of slice criteria (*nQ,VQ*) where *nQ*is the last statement of *Q* and *VQ* is all actual parameters from *P* pass to *Q*; function Down(S) is defined as: a set of (*NR,VR*) such that NR is statements in *R* that call *P*, and *VR* is the set of relevant variables at the first statement of *P* that substituted with formal parameters. The slice is got by iteratively generate Up(S) and Down(S) set until no new criteria are generated.

However this approach is inaccurate, because it fails to account for “calling context” problem as Horwitz, Reps, and Binkley pointed out in [18]. The shortcoming of Weiser’s approach is that it is infeasible in the case of entering a procedure *Q* fromprocedure *P* and exit *Q* to another, different procedure. To solve this, when a procedure *Q* is called by procedure *P*, all call sites that call procedure *Q* have to be considered, not only *P*. Horwitz et, al. had their PDG based on [7] with three other categories of vertices: a distinguished vertex called *entry vertex*, *initial definition of x* vertex represents initial state of variable *x* in program *P*, *final use of x* vertex represents final value of *x* compute by *P*. They define control dependency between two vertex as: if *v1* is an entry node, *v2* is control dependent on *v1* when *v2* is not nested in loop or conditional; if *v1* presents for predicate of while-loop, then *v2* is a component nested in loop body and edge *v1-> cv2* is labeled true; if *v2* represents for predicate of conditional statement, *v2* is the component in the conditional branch and edge *v1-> cv2* is labeled true or false depends on whether *v1* lead to execute it. The data-depencences edges occurs in their PDG contains two types: *flow dependences* and *def-order dependences*. Flow dependences is the path in standard CFG augmented with *loop carried* or *loop independent*. The edge *loop carried* flowdependence if the value change in vertex which inside a loop will effect on the predicate or another vertex inside the loop. If the execution of vertex have no effect on the loop, then edge is a loop independent flow dependence. Def-order dependences edges represent for reassignment a variable. Figure 2.4 is an example of the PDG define in [18].

Then they construct the *System Dependency Graph (SDG)* on top of their PDG. In the first step, they model procedure calls and parameter passing with five new kinds of vertices and three new kind of edges. A call site in a SDG is represented by a *call-site* vertex; on the calling side, *actual-in* and *actual-out* vertices are control dependent on the *call-site* vertex and copy the value of the actual parameter to temporary variables/ F*ormal-in* and *formal-out* vertices correspond to formal parameters of a procedure and are control dependent on this procedure’s entry vertex. A *call* edge start from *call-site* vertex to corresponding procedure entry vertex; a *parameter-in* edge is binding corresponding *actual-in* and *formal-in* vertex; similar, a *parameter-out* edge is binding corresponding *actual-out* and *formal-out* vertex. So the SDG includes two parts, a PDG for the main program, and a procedure dependency graph for each procedure. The second step to construct the SDG is to build transitive dependence edges. This dependent relationship due to the value changing of variables after the calling of a procedure. *Subordinate characteristic graphs* are used to compute transitive dependencies. This graph depicts the transitive flows between the procedure’s input and output, and also, the call and return relations between the call site and procedure. To perform an inter-procedural slicing, their algorithm includes two phases that can be derived by starting vertex *s*: first, identify the vertices which can reach *s* without descending into procedure calls, it’s either in *P* itself or in the procedure that calls on *P*; second, identify the vertices that can reach s from procedures called by *P* or from procedures called by procedures that call *P*. In addition, they also use a data flow analysis when building the procedure dependency graph to identify modified variables and referenced variables. This results in a more accurate slice.

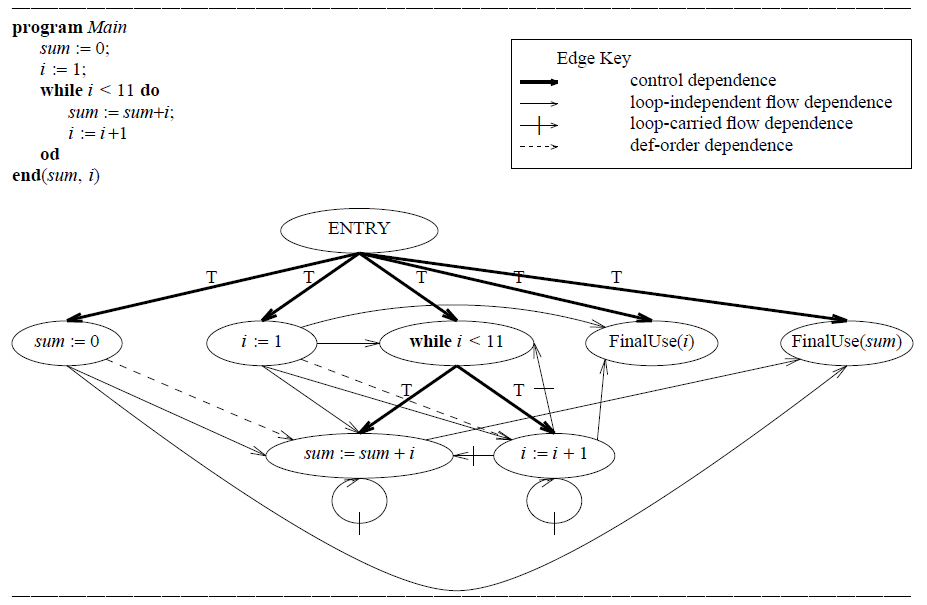


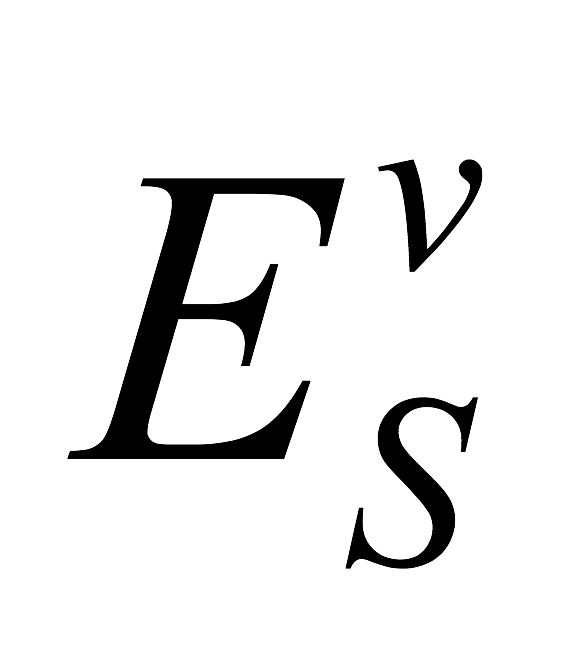
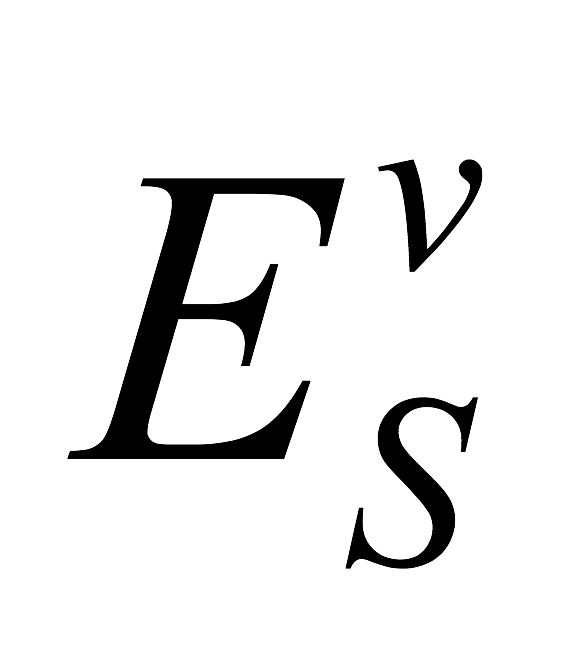
Figure 2.4 An example program, which sums the integers from 1 to 10 and leaves the result in the variable sum, and its program dependence graph. The boldface arrows represent control dependence edges, solid arrows represent loop independent flow dependence edges, solid arrows with a hash mark represent loop-carried flow dependence edges, and dashed arrows represent def-order dependence edges[18].

Later, Lakhotia [22] presents an alteration algorithm for inter-procedural static slicing on a SDG. Lakhotia’s approach labels vertices in the SDG with a three-valued vertex tag: “” marks node has not been visited; “Τ” marks the visited node; “*β* ” indicate a visited node and some of its reachable node should be visited. These tag permits to traverse each edge at most once to get slices. The node in slice labeled with “Τ” at beginning and all other node is “”, and backward slicing algorithm traverse on the node and change the tag value based on operation they configured in algorithm where:

 *β = β* = *β, x*  Τ = Τ *x*, and *x* *x* = *x*. All the nodes labeled *β* and Τ in the slice when the algorithm stops. However, it may require to change the value change twice when slice program. In a subsequent study, the SDG is also extended into object-oriented programs [20] to identify reference between classes. Donglin and Mary discuss a SDG [19] based on existing approach[20, 21] that can distinguish data members for different instances of the same class.



Another approach is also suitable for inter-procedural static slicing proposed by Bergerretti and Carre [17], they define a information-flow relations to identify the information transition statements that can be used to compute slices. In their approach set of variable denoted by *V* and *E* denote the set of predict expressions in program. For each statement *S*, they define three relations between V and E: *(e,v)λs* if for the value of *v* on entry to *S* potentially affects the value computed for *e*; *(e,v)μs* if the value computed for *e* potentially affects the value of *v* on exit from *S*; *(e,v)ρs* if the value of *v* on entry to *S* may affect the value of *v*’ on exit from *S*. They also introduced a notion partial statement denoted as which indicates those value may be used in obtaining the value of *v* on exist from *S*. Slices can be got by replacing the those statements doesn’t contain some member of by an empty statement.



***Unstructured control flow***

The approach of program slicing introduced above compute the projection for structured control flow. In [23], Ball and Horwitz point out the previous approach[1, 2, 7] is failure to deal with unstructured control flow, such as a unconditional jump statement **break**. Their solution to this problem is construct a PDG on top of an augmented CFG in which a unconditional jump (i.e, **break**, **goto**, **halt**) is represented as a *pseudo-predicate* vertex. The value of *pseudo-predicate* vertex is always set true, and its true-successor is the target of jump; its false-successor indicate the jump statement is continuation. Similar with Ottenstein’s approach, the slicing is graph reachability problem.

Choi and Ferrante [25] represented two methods to deal with **goto** statement. As same as [23], their first method is to build augmented program dependency graph base on augmented CFG. Choi and Ferrante extend augmented CFG with *fall-through* edge. Informally, a *fall-through* edge between two statement Si and Sj indicates a relationship: if Si is replaced by **null** statement, the program control flow coming into Si would be forwarded to Sj. The notion *fall-through* edge is renamed by *lexical successor* in [28]. A *fall-through* edge is used to capture the relevant control dependent when the **goto** statement is deleted. Their testing example shows the augmented slice may contain statements that don’t affect the statement for which the slice is computed. After analysis the disadvantage of this method, Choi and Ferrante proposed the second method that computes smaller slice as the executable slice by remove redundant cascaded **goto** statement. In this method, they define a new kind of slice named *CFG slice*. *CFG slice* is extract from CFG that includes nodes from which there is a path to the statement to compute slice. The executable slice is out of classical PDG and *CFG slice* by replaced with each statement’s immediate post-dominator.

Another PDG-based static slicing algorithm for unstructured flow is proposed by Agrawal [28]. Agrawal demonstrated the conventional slicing algorithm (refers to slicing algorithm for classic PDG) is easy to be extended with conditional jump statement in the form of **if** P **then goto** L: if the predicate P is in included in the slice because there is other statement control dependent on it, then the associated jump statement L must be included in the slice. In the case of unconditional jump statement,

Agrawal found the way to determine a statement in a slice by: if and only if the the nearest post-dominator and nearest lexcial successor of statement J is different, then J must be included in the slice. A statement S’ is post-dominator of another statement S, when every path from S to exist contains S’[35]. Considering a sequence of statements S1, S2, S3 in original program, if nearest post-dominator of S2 is different with nearest lexcial successor, it means exclude S3, S2 cause control jump to elsewhere. In this case S2 is unconditional jump statement in original program, omit S2 from the slice will cause the control always transfer from S1 to S3. Agrawal distinguished structured jump statement if its target statement is also its lexical successor. The property of structured jump ensure a single traversal of the post-dominator tree is sufficient to obtain correct slice. Second, if a predicate statement is in slice by conventional slicing algorithm, the jump statement directly dependent on it will be also included in the slice.

***Composite variables and pointer***

In order to construct flow dependences for pointers, aliasing which refers to the phenomenon of two or more variables point to the same memory location has to be determined. The James R. and David proposed [29] algorithm use symbolic execution to construct aliasing identical list. They define two type of address: address for static object and address for dynamic object. Address for dynamic object enable to keep indirect assignment through dynamic address can be treated as assignment to array. Their algorithm consists of two phases. In the fisrt step, extract Pointer State Subgraph (PSS) by deleting the nodes in CFG without pointer value, and also deleting the branch node under the node being deleted, then repeat propagate address from address creation node to all successor on PSS. Second, the point state information we get in first step is used to determine which statements should be included in a slice[34].

**Concurrency**

Multiple flows, synchronization, communication, non-deterministic selection cause identify primary program dependency more difficult in concurrent program. Cheng introduced an graph-theoretical approach[31] for concurrent program slicing based on his previous work. Nondeterministic parallel control-flow net introduced in [31] is used to represent control flows, and can be bundled with information concerning definitions and the uses of variables and communication channels. In addition to data and control dependency, Cheng defines selection dependency, synchronization dependency, communication dependency to construct *Program Dependence Net* (PDN). Selection dependency is a kind of control dependency involves non-deterministic selection statement; synchronization dependency reflect the synchronous execution between two statement; communication dependency corresponds to modification of a variable at a point directly has influence on the value of a variable at another point via interprocess communication. Program slicing is a depth-first or breadth-first graph traversal algorithm on PDN. In final, Cheng also discuss how to alter his algorithm into dynamic slicing and forward slicing.

### 2.2.2 Dynamic and conditional program slicing

So far, static program slicing we introduced is independent of input value. On the contrast, dynamic and conditional program slicing have to take account into input at slicing criterion. The difference between dynamic and conditional slicing is that the former only considers one specific input when compute slices, but the latter computes slices with a range of possible input.

Dynamic program slicing is first proposed by Korel and Laski [21,26]. Their approach depends on data flow analysis. They redefine flowgraph of program *P* is a tuple *<N, A, en, ex>* where *N* is a set of nodes represent for statement, *A* is a set of binary relationships between nodes in *N*, *en* and *ex* are used to label the unique entry and exist node of program. The notion of *arc(n,m)*  *A* identifiesthe control transition from node *n* to *m*. They distinguished the path whose execution invoked by input feasible path. After the execution of feasible path derived by a specific input, the tracing history through the path is referred as program trajectory *T*. The slicing criterion is defined as a triple *(x, Iq,V)* where x is the input, *V* is subset of program variables, and *Iq* is the *qth* element of the program trajectory. The dynamic flow they introduce includes three type: Data-Data relations(DD), Test-Control relations(TC) and Identity Relation(IR). DD models the relationship of definition and usage of same variable, and TC is used to determine which statements will be executed upon the predicate in **while** or **if-then-else**. IR refers to duplicate elements in *T*. Furthermore, they subdivide the use of a variable with its last definition as definition-use(DU). The slice is computed by iterative finding directly and indirectly relevant statement. The role of three type of relation ensure traverse on dynamic flow. They formalize the process as follow[21]:



S0 = A0 = LD(q,V) + LT(Iq),

Si+1 = Si + Ai+1,

where

Ai+1={ Xp | Xp  Si, (Xp, Yt)(DDTCIR) for some YtSi,p < q}

In the formalization, Xp is referred to statement X at execution position p; LD(q, V) is the set of last definitions of variables in *V* at execution position *q*, and LT(Iq) is the set of test instructions which have control influence on the execution of Iq. The example of this approach is shown in figure 2.5, also the figure gives the comparison of static program slicing.

1

2

3

4

5

6

7

8

9

read(n);

i := 1;

**while** (i<=n) **do**

**begin**

**if** (i **mod** 2 = 0) **then**

x := 17

**else**

x := 18;

z := x;

i := i + 1

**end**;

write(z)

**(a)**

11 read(n)

22 i := 1

33 i <= n

44 (i **mod 2** = 0)

65 x := 18

76 z := x

87 i := i + 1

38 i <= n

99 write(z)

**(b)**

DU = { (11,33), (11,37), (22,33), (22,44), (22,76), (22,87), (65,98), (87,38), (65,76) }

TC = { (33,44), (33,65), (33,76), (33, 87),

(44,65), }

IR = { (33,38),(38,33) }

**(c)**

read(n);

i := 1;

**while** (i<=n) **do**

**begin**

**if** (i **mod** 2 = 0) **then**

x := 17

**else**

z := x;

i := i + 1

**end**;

write(z)

**(d)**

read(n);

i := 1;

**while** (i<=n) **do**

**begin**

**if** (i **mod** 2 = 0) **then**

x := 17

**else**

x := 18;

z := x;

i := i + 1

**end**;

write(z)

**(e)**

Figure 2.5 (a) Example code; (b) Trajectory of (a) for input n =1; (c) Dynamic flow concept for this trajectory; (d) Dynamic slice for slicing criterion <1, 99, z>; (e) Static slice for slicing criterion <9,z>.

In the example, dynamic slice on slicing criterion <1, 99, z> is computed. In the process of computation, A0 is {76} and produced in subsequently A1={33, 22, 65}, A2={11,38, 44} A3={87,37}. From figure 2.5(d) and (e) we can see the different result from dynamic slice and static slice. In static program slicing without assumption of input value, the result slice always contains statement 6, on contrast, dynamic slice get rid of statement 6 because this statement won’t be executed when variable *n* equals to 1. In the sake of take account into a specific input, dynamic generate a smaller slice of program. However, Korel and Laski don’t give an proof about sufficient of their approach. The trajectory of when input value *n* equals 2 is shown in Figure 2.6 (a). In this time, the dynamic concept follow for this trajectory is :{ (65,76), (510,711) }DU, (76, 711)IR. Statement 6 will be included in slice, however, this statement is not necessary to slice cause by reassignment of variable *z* on statement 5.

Based on Korel and Laski’s method, Agrawal and Horgan discussed dynamic program slicing on top of Dynamic Dependency Graph(DDG)[32] where DDG is an extension of PDG that represents for execution history with a specific input. The slice is get from traversal on all vertex that can be reached from criterion. Figure 2.6 gives the trajectory of for input *n* = 2,corresponding PDG and DDG of the program of in figure 2.5 (a). Gray nodes in figure 2.6 (c) is the slice result on < 2, 914, z>.

11 read(n)

22 i := 1

33 i <= n

44 (i **mod 2** = 0)

65 x := 18

76 z := x

87 i := i + 1

38 i <= n

49 (i **mod 2** = 0)

510 x := 17

711 z := x

812 i := i + 1

313 i <= n

914 write(z)

**(a)**

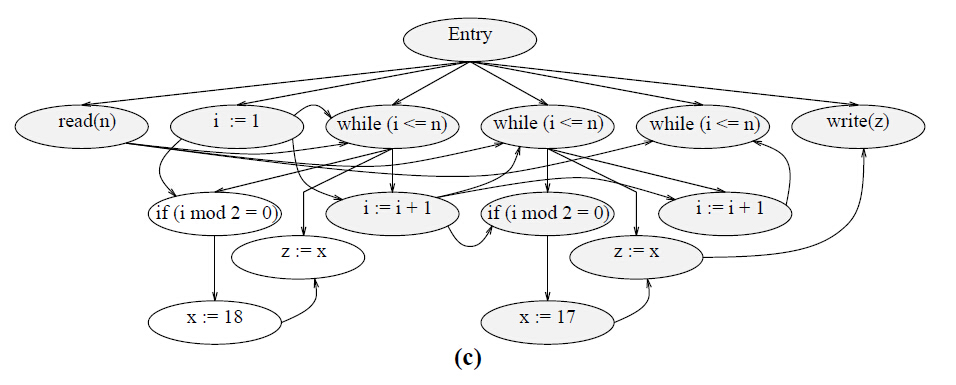
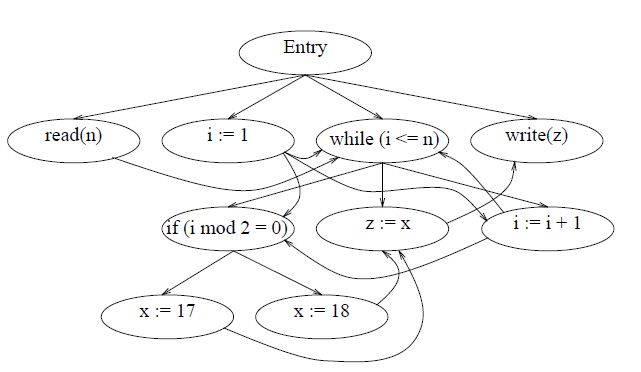


Figure 2.6 (a) Trajectory of the program of figure 2.5 (a) for input n =2; (b) PDG of the program of figure 2.5 (a); (c) DDG of the program of figure 2.5 (a) [3].

ConSIT is a conditional program slicing tool introduce by Danicic et al.[24]. The slicing criterion is a tuple <*V, n, π*>, where *V* is a set of variables, n is the position and *π* is some condition. Their algorithm consists of three phase. The first phase is symbolically execute to propagate state information to each statement in program. The second phase is a theorem proving phase to determine which statement should be eliminate from slice under the input condition. The last phase is carry out traditional static program slicing on conditional program extract from second phase. As they state in conclusion, their approach on conditional is limited on single-procedural program.

2.3 Applications of program slicing

The original purpose of program slicing is for debugging[1, 12]. In later, the application of program slicing is also proposed in testing[29], software maintenance[19], software architecture[14].

3

Abstract Interpreter

4

Program Slicing on State Graph

5

Validation

6

Conclusion

Reference

[1] M. Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, Ann Arbor, 1979.

[2] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[3] Tip F. A Survey of Program Slicing Techniques.[J]. Journal of Programming Languages, 1994, 3(93):121--189.

[4] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In Conference Record of the Eighth ACMSymposium on Principles of Programming Languages, pages 207–218, 1981.

[5]"Experience with a data flow datatype” J. Comput. Languages, to be published, 1983.

[6] http://www.tonymarston.net/php-mysql/3-tier-architecture.html

[7] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–349, 1987.

[8] J.M. Barth. A practical interprocedural data flow analysis algorithm. ommunications of the ACM, 21(9):724–736, 1978.

[9] Ottenstein K, L. Ottenstein: "The program dependence graph in a software development environment[J]. Acm Sifsoft, 1984.

[10] Wang L M, Xian Y, Zhang L, et al. JSSlicer: A Static Program Slicing Tool for JavaScript[J]. Applied Mechanics & Materials, 2013, 241-244:2690-2695.

[11] Rodrigues N F, Barbosa L S. Component Identification Through Program Slicing[J]. Electronic Notes in Theoretical Computer Science, 2006:291–304.

[12] Weiser M. Programmers Use Slices When Debugging.[J]. Communications of the Acm, 1982, 25(7):446-452.

[13] Mohapatra D P, Mall R, Kumar R. An Overview of Slicing Techniques for Object-Oriented Programs[C]// INFORMATICA. 2006.

[14] Zhao J. Applying Slicing Technique to Software Architectures[C]// In Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems. 1998:87--98.

[15] Meijer, Henricus Johannes Maria, Manolescu, Dragos. Tier splitting for occasionally connected distributed applications: WO, WO2011100171 A2[P]. 2011.

[16] M. Weiser. Private communication, 1994.

[17] J.-F. Bergeretti and B.A Carre. [Information-flow and data-flow analysis of while-programs](http://dl.acm.org/citation.cfm?id=2366" \t "http://xueshu.baidu.com/_blank). ACM Transactions on Programming Language and Systems, 7(1):37-61, 1985.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 12(1):26–61, 1990.

[19] Liang D, Harrold M J. Slicing objects using system dependence graphs[C]// Software Maintenance, 1998. Proceedings., International Conference on. IEEE, 1998:358-367.

[20] Larsen L, Harrold M J. Slicing object-oriented software[C]// icse. IEEE Computer Society, 1996:495.

[21]Korel B, Laski J. Dynamic program slicing[J]. Information Processing Letters, 1988, 29(88):155–163.

[22] Lakhotia A. Improved Interprocedural Slicing Algorithm[J]. Report Cacs Tr, 1992.

[23] Ball T, Horwitz S. Slicing Programs with Arbitrary Control Flow[C]// 1ST CONFERENCE ON AUTOMATED ALGORITHMIC DEBUGGING. 1993:206--222.

[24] G. Canfora, et al. Conditioned program slicing. Information and Software Technology, 40(11-12):595~607 1988.

[25] Choi J D, Ferrante J. Static Slicing in the Presence of GOTO Statements[J]. Acm Transactions on Programming Languages & Systems, 1994, 16(4):1097--1113.

[26] Korel B, Laski J. Dynamic program slicing[C]. Information Processing Letters, 1988, 29(88):155–163.

[27]T. Reps and W. Yang. The semantics of program slicing and program integration. In Proceedings of the Colloquium on Current Issues in Programming Languages, volume 352 of Lecture Notes in Computer Science, pages 60–74. Springer Verlag, 1989.

[28] Agrawal H. On Slicing Programs with Jump Statements[J]. Acm Sigplan Notices, 1994, 29(6):302-312.

[29] Landi W, Ryder B G. A safe approximate algorithm for interprocedural pointer aliasing[J]. Acm Sigplan Notices, 2004, 39(7):473-489.

[30]R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In Proceedings of the Conference on Software Maintenance, pages 299–308, 1992.

[31] Cheng J. Slicing Concurrent Programs - A Graph-Theoretical Approach[C]// Proceedings of the First International Workshop on Automated and Algorithmic Debugging. Springer-Verlag, 1993:223--240.

[32] Agrawal H, Horgan J R. Dynamic Program Slicing[J]. Sigplan Notices, 1990, 25(6):246-256.

[33] GB/T 7714 Weihl W E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables.[J]. POPL '80 Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1980.

[34] J.R. Lyle. Evaluating Variations on Program Slicing for Debugging. PhD thesis, University of Maryland, 1984.

[35] R. Compilers : principles, techniques, and tools[J]. Compilers Principles Techniques & Tools, 1988.

[36]ALLEN, F. E. Control flow analysis. In Proceedings of a Symposium on Compiler Optimization. SZGPLAN Not. 5, 7 (July 1970), 1-19.

[37]Ottenstein K J. Data-Flow Graphs as an Intermediate Program Form[J]. Purdue University, 1978.

[38]Ottenstein K J. And Ottenstein L M.[An intermediate program form based on a cyclic data-dependence graph](http://citeseerx.ist.psu.edu/showciting?cid=2115552" \t "http://xueshu.baidu.com/_blank). CS-TR 81-1,Dept. Of Computer Science, Michgan Technological Univ., Houghton, MI, Oct. 1981; July 1982, errata.