Three Approaches to Interprocedural Dynamic Slicing

Mariam Kamkar, Peter Fritzon, Nahid Shahmehri
Department of Computer and Information Science
Linköping University, S-581 83 Linköping, Sweden
Email: marka@ida.liu.se, petfr@ida.liu.se
Phone: +46-13-281000, Fax: +46-13-282666

The need of maintenance and modification demand that large programs be decomposed into manageable parts. Program slicing is one method for such decomposition. A program slice with respect to a specified variable at some program point consists of those parts of the program that may directly or indirectly affect the value of that variable at the particular program point. This is useful for understanding dependences within programs.

A static program slice [Wei84] is computed using static data- and control flow analysis and is valid for all possible executions of the program. Static slices are often imprecise, i.e., they contain unnecessarily large parts of the program. Dynamic slices [KL90] [AH90] however, are precise but are valid only for a single execution of the program. Interprocedural dynamic slices can be computed for programs with procedures.

This paper presents the first three techniques for interprocedural dynamic slicing which deal with procedures/ functions at the abstract level. All three methods first generate summary information for each procedure call (or function application), then represent a program as a summary graph of dynamic dependences. A slice on this graph consists of vertices for all procedure calls of the program that affect the value of a given variable at the specified program point. The amount of information saved by these methods is considerably less than what is needed by previous methods for dynamic slicing [KL90] [AH90], since it only depends on the size of the program's execution tree, i.e., the number of executed procedure calls, which is smaller than a trace of all executed statements.

The interprocedural dynamic slicing methods introduced here are applicable in at least two areas, program debugging [SKF90] [KSF90] and data flow testing.

1 Introduction

This paper presents three approaches for computing interprocedural dynamic slices of a program based on:

- Collection and computation of interprocedural control and data dependence information and representation of this information as temporary graphs.
- Abstraction of interprocedural information for each procedure and construction of an interprocedural summary graph.
- Computation of interprocedural slices.

A slice is extracted from the interprocedural summary graph that represents the control and data dependences of variables that extend across procedure boundaries. Thus, the essential portion of our slicing technique is the construction of the interprocedural summary graph.

The control dependence information is derived from the program structure. This information is static, since for a given statement s, the statements that s is control dependent on do not change in different executions. However, the statements that s is data dependent on may vary in different executions. Hence, the data dependences among program statements are dynamic. The computation of control dependences in arbitrary programs is described in [FOW87].

We present three alternative methods for collecting data dependence information: statically over all executions, dynamically with respect to a specific execution, and in a combined static/dynamic manner. Using any of these methods, we obtain an interprocedural summary graph for computing interprocedural dynamic slices. The interprocedural summary graph is constructed dynamically at run-time, so a slice on this graph is a dynamic slice. However, if the data dependence information in the interprocedural summary graph is computed statically, the resulting slices are dynamic slices based on static information. Similarly, if dynamic (combined static/dynamic) data
dependence information is used, the slices are dynamic slices based on dynamic (combined static/dynamic) information.

The next section introduces the dynamic interprocedural summary graph. Section 3 explains the process of extracting a slice on this graph, and Section 4 describes a mapping function from a slice into the source program. Section 5 describes the construction of the summary graph and Section 6 presents the conclusions of this work.

2. Dynamic Dependence Summary

Graph

In this section we introduce the dynamic interprocedural summary graph. This graph is a condensed representation of two other graphs. These two graphs represent, respectively, the calling relationships and the interprocedural data dependences that exist among procedures in a program during a particular execution.

Here, we give some definitions which are used later in this section to introduce the dynamic dependence summary graph. These definitions may represent static, dynamic or combined static and dynamic information according to their application in the corresponding analysis method.

Definition: An incoming variable to a procedure is a variable defined outside the procedure, and which is used before being defined in the procedure. An outgoing variable is a variable which is defined in the procedure and whose value is accessible outside the procedure. An incoming/outgoing variable refers to a variable which is an incoming variable or an outgoing variable or both. Examples of these variables are reference parameters, which can be used as both incoming and outgoing in these variables, and value parameters, which can only be used as incoming. Global variables are treated as incoming variables, outgoing variables or both according to their usage within the procedure. If this information is collected statically, then they represent potential incoming/outgoing variables, while in dynamic collection of this information they are actual incoming/outgoing variables.

Definition: InComing(p) is the set of all incoming variables to procedure p, and OutGoing(p) is the set of all outgoing variables of p. A variable can belong to both InComing(p) and OutGoing(p).

Definition: An activation edge from procedure p1 to procedure p2 represents an activation of procedure p2 from the body of p1. In the case of static information this is a possible activation, and in the case of dynamic information this is an actual execution.

Definition: A summary dependence relation on the set of incoming/outgoing variables is the set of all ordered pairs <x, y> such that x and y are both incoming/outgoing variables and x has a transitive dependence on y through combined data and control dependences. The pair is called a summary dependence edge. It represents a possible dependence when considering static information, and an actual dependence otherwise.

Definition: A procedure instance is a pattern which contains the procedure name and the incoming and outgoing variables of the procedure. Different executions of a procedure may give rise to different procedure instances, due to different values of its incoming and outgoing variables, since execution can follow different paths inside the procedure.

procedure instance = <inp, p, outp>, where:

p represents a procedure,

inp = {x | x ∈ InComing(p)}

and

outp = {x | x ∈ OutGoing(p)}

Here, we define two graphs, SummaryG1 and SummaryG2, which for a particular program execution represent the calling relationships and the interprocedural data dependences among procedures, respectively.

Let SummaryG1 be a directed graph with a set of vertices V and a set of edges E as follows:

SummaryG1 = <V, E> where

V = {s | s is a procedure instance: <inp, p, outp>}

E = a set of activation edges s1 → s2 between two procedure instances. Such an edge shows that procedure p2 in s2 is activated from the body of procedure p1 in s1 during program execution. Procedure instances s1 and s2 are in a parent-child relation.

Let SummaryG2 be a directed graph with a set of vertices V and a set of edges E as follows:

SummaryG2 = <V, E> where

V = {v | v is a variable instance: <p, x> where p represents a procedure, and

x ∈ InComing(p) and/or OutGoing(p)}

E = a set of summary dependence edges v1 → v2
between variable instances. Such an edge shows that the computation of the value of \( x \) in \( v_1 \) is dependent on the value of \( y \) in \( v_2 \). An edge represents a transitive dependence provided by summarizing control and data dependences between two variables. We distinguish these two kinds of dependence edges:

- **Intra-summary dependence**, when an edge connects two variables belonging to the same procedure instance. This is caused only by local dependences within a procedure; the effect of the calls from the body of the procedure are excluded.

- **Inter-summary dependence**, when an edge connects two variables belonging to procedure instances that are either in a parent-child relationship or have the same parent.

Intuitively, these two graphs together represent an execution tree of a program, decorated with dependence information for all incoming/outgoing variables of its procedure instances. As we are going to use both graphs in parallel in the remainder of the thesis, for the sake of simplicity we represent them as one single graph. This graph is called the **dynamic dependence summary graph** or simply the **summary graph**. We further simplify the illustration of the summary graph by using simple lines instead of arrows to represent the activation edges. This is due to the fact that the execution tree of a program is acyclic. Thus, the intuitive direction of the activation edges in figures is top-down.

As an example, consider Figure 1. where procedure \( P \) calls procedures \( P_1, P_2, \) and \( P_3 \). Procedure \( P_3 \) in turn calls procedure \( P_4 \) and \( P_4 \) calls \( P_5 \). The value of the outgoing variable \( y \) of \( P \) is a result of activation of two procedures \( P_1 \) and \( P_2 \). This value is affected by the outgoing variable \( d \) of \( P_2 \) while \( d \) in turn is affected by the value of \( c \) of \( P_2 \) and so on. These affect relations create a chain as follows:

\[
y \rightarrow d \rightarrow c \rightarrow b \rightarrow a \rightarrow P_1 \rightarrow x \rightarrow P
\]

The value of \( P \)'s outgoing variable \( z \) is dependent on the activation of \( P_3 \) and not on \( P_1 \) or \( P_2 \). The chain of affects is:

\[
z \rightarrow f \rightarrow i \rightarrow h \rightarrow e \rightarrow a \rightarrow P_3 \rightarrow x \rightarrow P
\]

The value of outgoing variable \( w \) of \( P \) is also dependent on activation of \( P_3 \). This value is affected by outgoing variable \( g \) of \( P_3 \) with the chain of affect as follows:

\[
w \rightarrow g \rightarrow j \rightarrow l \rightarrow k \rightarrow h \rightarrow e \rightarrow a \rightarrow P_3 \rightarrow x \rightarrow P
\]

A dynamic dependence summary graph is constructed at run-time during execution of a program.

![Figure 1. A summary graph representing dependence relations between procedure instances.](image)

### 3 Interprocedural Execution Slice

Once the interprocedural summary graph of a program has been constructed, the computation of an execution slice on an incoming/outgoing variable is straightforward.

We define a **slice criterion** \( C \) to be a pair of a vertex \( s \) representing a procedure instance \(<in_{p_1}, p, out_p>\) and a variable instance \( v = <p, x> \) where the incoming/outgoing variable \( x \) belongs to \( in_p \) or \( out_p \) or both:

\[
C = <s, v> 
\]

A slice on the summary graph with respect to the slicing criterion \( C \) is a subgraph containing all vertices with a variable \( y \) such that there is a transitive dependence between \( x \), observed at the slicing criterion \( C \), and \( y \). The interprocedural execution slice consists of all procedure instances which have affected the computation of the value of \( x \). It is possible that only some of the incoming/outgoing variables of a selected procedure instance, \(<in_{p_1}, p, out_p>\), are involved in computation of \( x \). We refer to such a procedure instance as a **partial procedure instance** \(<in'_{p_1} \)
Thus, only these partial procedure instances and their dependence edges will be in the slice. Let a partial procedure instance of a procedure instance \( \langle \text{in}_p, p, \text{out}_p \rangle \) be:

\[
\langle \text{in}'_p, p, \text{out}'_p \rangle
\]

such that

\[
\text{in}'_p \subseteq \text{in}_p \text{ and/or } \text{out}'_p \subseteq \text{out}_p, \text{where } \text{in}_p \text{ and } \text{out}_p
\]

belong to \( \langle \text{in}_p, p, \text{out}_p \rangle \).

We define an execution slice on the two graphs \( \text{SummaryG1} \) and \( \text{SummaryG2} \) (see Section 1) with respect to the slicing criterion \( C = \langle s, v \rangle \) as two subgraphs \( \text{SummaryG1}' \) and \( \text{SummaryG2}' \).

\[ \text{SummaryG1}' = \text{execution slice}_{\text{SummaryG1}}(C) = \langle V', E' \rangle \]

where

\[ V' = \{ s \} \cup \{ s' \mid s' \text{ is a partial procedure instance of a procedure instance in } \text{SummaryG1} \text{ with at least one incoming/outgoing variable } y \text{ in } s' \text{ reachable from } x \text{ in } s \text{ through some edges in } \text{SummaryG2} \} \]

\[ E' = \{ e \mid e \text{ is an activation edge, in } \text{SummaryG1}, \text{ between two vertices belonging to } V' \} \]

\[ \text{SummaryG2}' = \text{execution slice}_{\text{SummaryG2}}(C) = \langle V', E' \rangle \]

where

\[ V' = \{ v \} \cup \{ v' \mid v' \text{ is a variable instance in } \text{SummaryG2} \text{ with an incoming/outgoing variable } y \text{ in } v' \text{ reachable from } x \text{ in } v \text{ through some edges in the graph} \} \]

\[ E' = \{ e \mid e \text{ is a summary dependence edge, in the } \text{SummaryG2}, \text{ between two variables such that the edge is part of the path starting from slicing variable } x \} \]

For the sake of simplicity, the two slices are represented as a single interprocedural execution slice on the summary graph. Figure 2 shows an interprocedural execution slice on the summary graph of Figure 1., with the slice criterion:

\[ C = \langle s, v \rangle \]

\[ s = \langle \text{in}: x, P \text{ (out: } y) \text{ (out: } z) \text{ (out: } w) \rangle \text{ and} \]

\[ v = \langle P, \text{out: } z \rangle \]

![Figure 2](image.png)

4 Mapping from Execution Slice to Program Slice

The execution slice on a summary graph contains all procedure instances which affect the value of a specified variable. These procedure instances represent execution of call statements in a program. Note that several procedure instances which denote different executions of a procedure can arise from the same call statement.

A program slice is the set of call statements in a program whose execution and hence execution of procedures called by them affect the value of a specified variable.

A many-to-one mapping function \( M \) from an execution slice to the source program will provide a program slice. The mapping function \( M \) maps one or several vertices in the execution slice to a single call statement in the source program.

\[ M(s) = cs \text{ where} \]

\[ s \text{ is a vertex in the execution slice and} \]

\[ cs \text{ is a call statement in the source program} \]

and

**program slice** \( (C) = \bigcup_{s \in \text{execution slice}(C)} M(s) \)

where \( C \) is the slicing criterion.

5 Construction of the Summary Graph

This section describes the construction of the summary graph introduced in the previous section. The method is based on collecting and summarizing
dependence information. The type of collected dependence information gives rise to the three approaches we have mentioned earlier, i.e., the static-analysis based approach, the dynamic, and the combined static/dynamic analysis based approach. For each approach, an algorithm describes the construction process of the summary graph. We use an example program to explain the algorithms.

Since during the process of constructing the summary graph, we use a number of auxiliary graphs called temporary procedure-dependence graphs, we first give a definition of these graphs.

5.1 Temporary Procedure-Dependence Graphs

A temporary procedure-dependence graph or, simply, a temporary graph, represents static dependence information for a single procedure, or dynamic dependence information for a specific execution of a procedure in the case of the static or dynamic approaches respectively. A temporary graph is a condensed representation of two other graphs, i.e., TempG1 which represents the flow of control of a procedure and TempG2 which shows the data and control dependences that exists between statements in a procedure. Before defining these graphs, we introduce some concepts.

Definition: A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. A call statement is considered to be a single statement block, referred to as a call block [ASU86]. We use the term block to denote both basic blocks and call blocks when the difference is not essential. We use two special block instances, entry and exit to represent start and end of a temporary graph.

Definition: An incoming variable to a block is either a variable which is used before being defined in the block or is a control variable whose evaluation affects the decision about execution of the block. An outgoing variable is a variable which is defined in the block and whose value is accessible outside the block. An incoming/outgoing variable refers to a variable which is an incoming variable or an outgoing variable or both. For a basic block this information is unconditional. However, for a call block, if this information is collected statically then it represents potential incoming/outgoing variables of the called procedure otherwise it represents actual information.

Definition: An activation edge from block b₁ to block b₂ shows that b₂ is activated after b₁ during a program execution. This activation may be caused by a branch statement at the end of b₁. In the case of static information, this is a possible activation, and in case of dynamic information, this is an unconditional activation.

Definition: A block dependence relation on the set of incoming/outgoing variables is the set of all ordered pairs <x,y> such that both x and y belong to incoming/outgoing variables and x is data or control dependent on y. The pair is called a block dependence edge. It represents a possible dependence when considering static information, otherwise it represents an actual dependence.

Definition: We define a structure called a block instance representing a basic block or a call block.

\[ \text{block instance} = \langle \text{in}_b, \text{b}, \text{out}_b \rangle, \text{where:} \]

- \text{b} represents a block
- \text{in}_b = \langle \text{in-data}_b, \text{in-control}_b \rangle \text{where}
  - \text{in-data}_b = \{ x \mid x \text{ is an incoming variable to block } b \}
  - \text{in-control}_b = \{ x \mid x \text{ is an incoming variable to block } b, \text{ whose value affects the execution of } b \}
- \text{out}_b = \{ x \mid x \text{ is an outgoing variable of } b \}

We now define the two graphs TempG1 and TempG2 as directed graphs:

\[ \text{TempG1} = \langle V, E \rangle \text{ where} \]

\[ V = \{ s \mid s \text{ is a block instance: } \langle \text{in}_b, \text{b}, \text{out}_b \rangle \} \text{ and} \]

\[ E = \text{the set of activation edges, between two block instances } s_1 \rightarrow s_2, \text{ which shows that if block } b_2 \text{ in } s_2 \text{ is activated, then it is activated after block } b_1 \text{ in } s_1 \text{ during program execution. This activation may be caused by a branch statement at the end of } b_1. \]

\[ \text{TempG2} = \langle V, E \rangle \text{ where} \]

\[ V = \{ v \mid v \text{ is a variable instance: } < b, x > \text{ where } b \text{ represents a block, and } x \in \text{ incoming and/or outgoing variables of } b. \} \]

\[ E = \text{the set of block dependence edges } y \leftarrow x \text{ between variable instances. Such an edge shows that the computation of the value of } x \text{ in } v_1 \text{ is dependent on the computation of the value of } y \text{ in } v_2. \text{ An edge represents either a control dependence or a data dependence between two variables. We distinguish these two kinds of block dependence edges:} \]
• **Intra-block dependence**, when an edge connects two variables belonging to the same block instance representing a basic block.

• **Inter-block dependence**, when an edge connects two variables belonging to different block instances, not necessarily in a parent-child relation between vertices.

Note that intra-block dependences are considered only for incoming/outgoing variables of basic blocks and not for the call blocks. The reason is that dependences between incoming/outgoing variables of a basic block are known directly from the body of the block. In contrast, for a call block, computation of intra-block dependences requires analyzing the body of the called procedure and the procedures called by it. This will be handled in the called procedure when there exists a local dependence between two variables. Thus, the dependence graph of a procedure shows the dependences that are directly derived from the body of the procedure without considering the effect of calls from inside this procedure.

As in the previous section we represent the two graphs TempG1 and TempG2 as a single graph called a temporary procedure-dependence graph (temporary graph).

Figure 3 illustrates a basic block $b_1$ with four assignment statements, a data dependence graph representing data dependences between variables inside the block, the summarized version of the data dependence graph and finally the block instance of the basic block together with intra-block dependences between its incoming/outgoing variables.

![Figure 3](image)

**Figure 3.** (I) basic block $b_1$, (II) data dependence graph of $b_1$, (III) summarized dependence graph, (VI) the block instance for $b_1$ with intra-block dependences for incoming/outgoing variables of $b_1$.

### 5.2 The Static-Analysis Based Approach

In this approach, data dependences are determined statically. First, for each procedure $p$ in a program, a temporary procedure-dependence graph is constructed. Data flow analysis is used to determine incoming and outgoing variables for each basic block and call block, the intra-block and the inter-block dependences. Then, by summarizing the dependence information within the temporary graph of $p$, a new graph will be constructed which represents the call relationship in addition to data and control dependences between $p$ itself and procedures which are/ may be immediately called by this procedure. We refer to such a graph as a static procedure-call linkage graph or simply a static linkage graph. A static linkage graph is valid for all executions of a procedure. The temporary graph can be deleted now, since its summarized information at procedure level is available in the linkage graph. Finally, the dynamic summary graph is constructed during execution of the program by copying the static linkage graphs for each procedure which is called, and combining these into the full summary graph.

The static linkage graph introduced here is similar to the linkage grammar, an attribute grammar that
models procedure-call structure, used in [HRB90].
The differences are in the construction step and the use each.

The three-step construction of the dynamic summary graph using the static-analysis based approach is as follows.

**Step 1: Construction of Static Temporary Graphs.**
As an example consider the program `number_generation` in Figure 4 which for a given value \( n \) generates a prime number by the formula \((n^n) - n + 41\) when \( n < 41\), otherwise the square of \( n \). This formula is correct only when \( n < 41\).

Temporary graphs which are constructed for procedures in the example program of Figure 4 are shown in Figure 5. For the sake of simplicity, we do not show exit blocks when it is not necessary. In this example, we model procedure `read` as an assignment statement which assigns a value to variable \( n \), and procedure `write` as an operation that only uses the value of the variable \( n \). We do not show temporary graphs and their corresponding static linkage graphs for these two procedures in this example. Nor do we show the dependences regarding the file variables `input` and `output`. Construction of temporary graphs for procedure `square` and `prime`, Figure 5 (I) (II), are straightforward since they contain only one basic block.

The temporary graph of procedure `the_number` in Figure 5 (III) contains five block instances. Because of control flow within this procedure, only one of the blocks \( b_4 \) and \( b_5 \) will be activated during an execution of the procedure. The reason is the if-statement in the source procedure. Thus, the activation edges between block instances \( b_2, b_4 \) and \( b_3, b_5 \) are possible activation edges. Another comment on this temporary graph is about control variables. In the source code of the procedure, the segment "if \( x < 41 \) then \( y := i \) else \( y := j \)" shows that the execution of "\( y := i \)" or "\( y := j \)" is dependent on evaluation of predicate "\( x < 41 \)" and hence on the value of \( x \). For this reason block instances for \( b_4 \) and \( b_5 \) contain \( x \) as in-control variable i.e., \(<(\text{in: } x)\>(\text{in: } i)\> b_4 \text{ (out: } y\) \> for "\( y := i \)" and \(<(\text{in: } x)\>(\text{in: } j)\> b_5 \text{ (out: } y\) \> for "\( y := j \)".

**Figure 4.** An example program for generation of a prime number or a square number.

**Step 2: Construction of Static Linkage Graphs.**
For each procedure \( p \) in the program, a static linkage graph is constructed through (1) building a call graph which shows the calling relationships between procedure \( p \) itself and its called procedures, and (2) decorating the call graph with the dependence information provided by the temporary graph of \( p \).
Figure 5. Static temporary graphs for procedures square, prime, the_number and number_generation. For the sake of simplicity the file variables input and output are not shown here.

Figure 6 shows static linkage graphs constructed from temporary graphs of Figure 5.

Step3: Construction of Dynamic Summary Graph.
A dynamic summary graph is constructed using static linkage graphs obtained in Step2, for a particular execution of the program under consideration. Every time a procedure is executed, a subgraph will be added to the summary graph. The subgraph is a copy of whole or parts of the procedures’s static linkage graph, dependent on execution of the procedure.

Figure 7 shows the summary graph constructed from static linkage graphs in Figure 6, for an execution of the program number-generation.
Shortcomings of the Static-Analysis Based Approach

This approach requires a pre-step of static intra- and inter-procedural data flow analysis. The information gathered by data flow analysis is static and hence is not precise. For example, the question of which variables are incoming or outgoing to a procedure yields all possible variables which can be candidates in some execution of the procedure. For illustration, consider procedure test which during an execution either uses i and defines y or uses j and defines z, while procedure instance computed using the static approach contains both i and j as incoming and y and z as outgoing variables.

\[
\text{procedure test; begin} \\
\quad \text{if } x > 0 \text{ then } y := 1 \text{ else } z := 0
\]

Thus, temporary graphs which are based on information obtained from data flow analysis also contain imprecise information. Static linkage graphs receive information from static temporary graphs and transfers it to the summary graph. Consequently the dynamic summary graph will contain unnecessarily conservative information. The summary graph in Figure 7 illustrates this matter, the outgoing variable y from procedure the_number is connected to both outgoing variable d of procedure square and outgoing variable b of procedure prime by two dependence edges. However, y is actually dependent on one of these variables at a time.

The degree of imprecision increases when using array references and pointer variables in a program. As a result, slices will contain procedures that have nothing to do with the computation of a variable of interest.

5.3 The Dynamic-Analysis Based Approach

In this approach all information is collected dynamically. The construction of the summary graph occurs gradually together with construction and deletion of dynamic temporary graphs. If there are no loops and no conditionals in a procedure, the activation structure of the procedure’s dynamic temporary graph of the procedure will be identical to activation structure of its static temporary graph introduced in the previous approach. During execution of a program, whenever a procedure is activated: (1) a new temporary graph for that procedure activation will be constructed, (2) as soon as the construction of the temporary graph is completed, its dependence information will be summarized as a subgraph and added to the summary graph, (3) the temporary graph will be deleted.

Here, a temporary graph is valid only for a particular execution of a procedure. There is no pre-analysis for finding blocks and their dependences. Each statement in the procedure is considered to be a single statement block. Trace routines create block instances and connect block dependences. We do not use linkage graphs in this approach since information is passed directly from temporary graphs to the summary graph.

The dynamic-analysis based approach constructs a dynamic summary graph through the two following steps.

Step 1: Construction of Dynamic Temporary Graph.

As an example, assume an execution of the program in Figure 4 when the value of n obtained from read statement is greater than 41. Then the number generated by the program will be a squared number. Construction of dynamic temporary graphs for this execution is shown in Figure 8.
with attached numbers indicate the transfer of information from temporary graphs to the summary graph. The arrow with number 1 shows the creation of the procedure instance for prime and its intra-summary dependence, i.e., the dependence between incoming and outgoing variables of the procedure instance itself. This is the first vertex created in the summary graph. The next transformation creates a second vertex which is the procedure instance for square and its intra-summary dependence. The third transformation results in creation of the procedure instance for the_number and its inter-summary dependences, i.e., the dependences between variables belong to the procedure instances of the_number, square and prime. As this approach shows, construction of the dynamic summary graph occurs bottom up.

Figure 9. The dynamic summary graph of an execution of the example program in Figure 4, constructed according to the dynamic-analysis based approach. As mentioned earlier file variables are not shown on these examples.

A slice on the summary graph in Figure 9 with slice criterion:

\[ C = \langle s, x \rangle \text{ where } s = \langle \emptyset \text{ number_generation (out: } m) \rangle \text{ and } x = (\text{out: } m) \]

will contain procedure instances of square and not of prime, besides other procedure instances. The actual dependence chain is as follows:

\( m \text{ of number_generation } \rightarrow y \text{ of the_number } \rightarrow d \text{ of square} \)

Comments on the Dynamic-Analysis Based Approach

This approach provides a summary graph contain-
ing precise information. Consequently, slices extracted from this graph are precise too.

Its main disadvantage is that generating traces all the time during execution can be costly in time and space. The dynamic temporary graphs are constructed during execution of procedures. Although they do not consider non-executed branches in the procedure, they contain information about all iterations of loops in the procedures. The number of existing temporary graphs at each moment is equal to the number of activated procedures in the current call chain.

5.4 The Combined Static/Dynamic-Analysis based Approach

The goal of this approach is to compute a "reasonable" amount of information at compile time and leave the rest to run-time. As in the first approach, a procedure is partitioned into basic blocks and call blocks. The information which is collected at compile time for each procedure consists of the data dependence information within basic blocks and the control dependence information between basic blocks. The computation of inter-block dependences and call block information is delayed until run-time. The following steps describe how the dynamic summary graph will be constructed using this approach.

Step1: Collection of Static Information.
Basic blocks which have the property that their static and dynamic data dependences are the same can be computed precisely at compile time.

Step2: Construction of Dynamic Temporary graph.
This step is similar to Step1 of the dynamic-analysis based approach. The differences are (1) working with basic blocks of consecutive statements instead of single statement blocks; (2) during construction of a temporary graph, whenever it is time for creation of a block instance for a basic block, the available information from Step1 is used directly. The block instances for call blocks and inter-block dependences will be computed during execution in the same way as we did in the previous approach.

Step3: Construction of Dynamic Summary graph.
This step is similar to Step2 of the dynamic-analysis based approach.

Figure 10 illustrates Step1 of this approach. Information which can be computed precisely at compile time is shown for simplicity as vertices and edges in temporary graphs. The information which is not known precisely and hence is not computed at compile time is shown as a question mark. Note that temporary graphs are actually constructed at run-time using both compile time and run-time information. Figure 8 and Figure 9 are also valid for the combined approach.

Figure 10. Static information for temporary graphs obtained at compile time.

Comments on the Combined Approach

The degree of precision in this approach is the same as in the dynamic approach. However, execu-
tion-time overhead here is lower than the dynamic approach by transferring part of the cost in generating traces from execution time to compile time. On the other hand, for every program execution the static information which is collected for the unexecuted program components remains unused.

6 Conclusions

We have introduced three methods for interprocedural dynamic slicing, which are applicable in at least two areas: program debugging and data flow testing [Kam93]. We conclude that interprocedural dynamic slicing is superior to other slicing methods when precise dependence information for a specific set of input data values at the procedural abstraction level is relevant.

The first of our methods gives less precise information than the other two but is less costly in terms of tracing. The second, purely dynamic method gives precise information but is more costly in terms of tracing. The third method is more efficient in terms of tracing and is as precise as the purely dynamic method.

References


