Program slicing for Java 6 SE

de Jong, J.P.

Program Slicing for Java 6 SE

Johan P. de Jong

July 2011

Eindhoven University of Technology
Department of Mathematics and Computer Science

Program slicing for Java 6 SE

Johan P. de Jong

In partial fulfilment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Supervisor: dr. Alexander Serebrenik

Examination Committee:
dr. Ruurd Kuiper
dr. Serguei Roubtsov
ir. Joost Gabriels
dr. Alexander Serebrenik

Eindhoven, July 2011

Abstract

Slicing is a powerful technique which extracts from a given program, a subprogram which contains all statements that potentially affect the values at some point of interest (the slicing criterion). At this time, no good slicer exists for the Java language which can cope with the current version (Java 6). The contribution of this project is a slicing platform capable of producing slices of Java 6 SE programs in the form of Java source code that can be compiled. The slicing platform is based on system dependence graphs. The platform is modular, so that it is possible to extend the platform which new types of dependence and slicing algorithms.

Contents

List of Figures

Chapter 1

Introduction

1.1 Program slicing as an abstraction method

Nowadays software codebases tend to be so large that it is impossible for a single person to oversee the entire system and all of its features. In order to control this complexity, program slicing can be used [60]. Slicing is a powerful technique that extracts from a given program a subprogram which contains all statements that potentially affect the values at some point of interest. The point of interest is called the *slicing criterion*. The concept of program slicing was originally introduced by Weiser (in 1979 according to [54]).

Slicing can be seen as a method of abstraction on programs. Essentially, the slicing criterion specifies a subset of the behaviour of a program and slicing reduces a program to a smaller form which is still able to reproduce the behaviour chosen by the slicing criterion.

Weiser [60] performed an experiment in order to investigate the practical side of slicing. The participants of his experiment were 21 experienced programmers. Each of the participants were given three programs to debug. These programs were short programs, accompanied by a description of their purpose and some test input and output which clearly showed the bug. After finding the bugs, the participants were shown program fragments. For each program fragment they had to indicate if they thought that particular code fragment was used in one of the programs. They had to give a rating on a scale of one to four (almost certainly used, probably used, probably not used, almost certainly not used). The program fragments were each chosen in one of the following five ways:

1. A set of ten adjacent statements from one of the programs they had to debug, including the line that contained the bug.

2. The same as above, except ten adjacent statements that dit not contain the bug.

3. Ten statements which were from a slice where the slicing criterion was the output statement that showed the bug.

4. A slice unrelated to the bug.

5. Some non-adjacent statements unrelated to the bug.

For 1, 2 and 3 the participants answered positive (almost certainly used or probably used) for about half of the program fragments, where this was significantly less for 4 and 5. The conclusion that Weiser draws from this experiment is that since the programmers were able to recognise the slices related to the bug as used in the programs they probably mentally reconstructed the slices in their mind while debugging. The experiment was only performed on small programs.

Since programs can be very large (which often makes programs complicated), it is helpful to have a tool which is able to do slicing.

Slicing has many applications. For example, it can be used to reduce the state space [17] prior to model checking [35] a program. Slicing can also aid in debugging by setting the slicing criterion to the variable that contains a wrong value, which is what Weiser had originally in mind [59, 60]. Calculating a slice with such a slicing criterion leads to a slice which aids in finding the bug, since the bug should be in one of the statements of the slice. Furthermore, slicing can be used for calculating metrics [32, 48] like cohesion (measure for the relatedness with one unit) or coupling (measure of how units depend on each other). Slicing is also of value when performing typical software engineering tasks like reverse engineering or maintenance where it is important to understand the software in such a way that changes can be made without affecting the unchanged parts.

1.2 Problem description

It is rather surprising that no good Java slicer which supports the Java 6 SE currently exists, even though slicing is a well explored problem domain [54]. Therefore, the contribution of this project is the following: build a slicing platform for Java, which supports Java 6 SE and is able to slice complete and incomplete programs. The slicing platform must be able to produce slices in the form of syntactically correct Java programs. Possible extensions are extracting models for model checking, calculating cohesion and coupling metrics and visualizing the slice by showing the program text where the statements in the slice are highlighted.

1.3 Thesis outline

Chapter 2 presents the concept of program slicing, first in an intuitive way and then in a formal way. Chapter 3 shows the challenges for slicing Java programs. This chapter also discusses the existing Java slicers and the software requirements for Java slicing platform. In Chapter 4 algorithms for slicing are discussed, together with the abstractions on Java code that they need. Chapter 5 shows in which way the discussed algorithms can be applied to slicing Java, the design of the slicing platform developed during this project and the validation studies performed on the slicing platform. Chapter 6 concludes and sketches directions for future work.

Chapter 2

Program slicing

In this chapter the concept of program slicing is presented. First, the concept is shown in an intuitive way. After that program slicing is formally defined. Last, the limitations that must be taken into account when programmatically performing program slicing are discussed.

2.1 Program slicing intuitively

A *program slice* in its original definition by Weiser [60] consists of the parts of a program that potentially affect the variables at some point of interest. In order to demonstrate this intuitive notion, consider the `SumProd` program (Program 2.1). Each statement has a number; the curly bracket at the end of the line of statement `(8)` is assumed to be part of statement `(5)`.

Program 2.1 `SumProd` [54]

```
(1)   int n = readInt();
(2)   int i = 1;
(3)   int sum = 0;
(4)   int product = 1;
(5)   while (i <= n) {
(6)       sum += i;
(7)       product *= i;
(8)       i++; }
(9)   writeInt(sum);
(10)  writeInt(product);
```

The program reads a value `n` from the standard input, and computes the sum and product of the numbers 1 through `n`. Now assume that one is only interested in the product of these numbers, but not the sum. It is of course possible to run the program and ignore the first number printed and to only take a look at the second one. The program would do superfluous calculations which would cause the program to use more time (superfluous executed statements) and resources (memory space wasted for unused variables). However, it is possible to extract a subprogram (a subset of the statements of the original program) which consists of only those statements needed in order to calculate the product.

Calculating a subprogram which only contains those statements required to calculate a value of interest is called *program slicing*, or just slicing. The subprogram resulting from slicing is called the *slice*. Slicing is always done with respect to a *slicing criterion*, which specifies the point of interest. The slicing criterion consists of a program statement and a set of program variables. Calculating the slice is (intuitively) performed as follows: start with a subprogram that only consists of the statement in the slicing criterion. From this, add statements to the slice that influence values needed at the statements that are already in the slice. Continue doing this until no more statements need to be added to the slice. Since the number of statements is finite, this process always terminates.

Example 2.1. *This example shows in which way a subprogram of* `SumProd` *can be derived which only calculates the product. The slicing criterion consists of statement* `(10)` *and variable* `product`.

Starting at statement `(10)`*, with variable* `product`*, it becomes clear that statements* `(4)` *and* `(7)` *are required, since they actually assign values to the variable* `product`*. When statement* `(7)` *is included, it will also be needed to include statements* `(8)` *and* `(2)`*, since variable* `i` *is used in order to calculate the value which is assigned to* `product` *in statement* `(7)`*. For statement* `(8)` *this may seem less apparant, since it appears after* `(7)`*, but statement* `(7)` *may be executed after* `(8)` *because of the loop. Also statement* `(5)` *is required since the number of times statement* `(7)` *is executed depends on it thus influencing the value calculated at statement* `(7)`*. After including statement* `(5)`*, statement* `(1)` *will also be needed, since the condition of the while depends on* `n`*. The resulting slice is named* `SumProd′` *and listed as Program 2.2.*

Program 2.2 `SumProd′`

```
(1)   int n = readInt();
(2)   int i = 1;
(4)   int product = 1;
(5)   while (i <= n) {
(7)       product *= i;
(8)       i++; }
(10)  writeInt(product);
```

The `SumProd` program was easy to slice in an intuitive way, since it only makes use of scalar variables (no objects), only consists of a single procedure (method) and does not use parallelism (threads). It does call two external methods (readInt and writeInt) but the code behind them is not explored. Slicing can also be applied to programs that do make use of these more advanced features. The difficulties that arise when slicing programs using these advanced features are outlined below.

Since the introduction of the concept of program slicing by Weiser (in 1979 according to [54]), different types of slicing have been proposed. At present, the technique which Weiser simply called ‘slicing’, is known as *static backward slicing.* This term consists of two parts: *static* and *backward. Static* means that only information that can be derived at compile time from the program text is used. This information can optionally be augmented with extra settings or external files given to the slicer. No information that is only available during run time is used and no assumptions are made on the input that the program processes.

Other types of slicing than static slicing exist, such as *dynamic slicing*, are shown in Section 2.2.2. Dynamic slicing is not limited to only information that is statically available, but a dynamic slice only applies to only one execution of a program (for one input). Dynamic slicing can be *backward* or *forward*, just like static slicing. *Backward* slicing means that slicing determines which program statements have an influence on the slicing criterion. Note that this does not necessarily mean that all statements in the slice have to occur before the statement in the slicing criterion. This was shown in Example 2.1 where statement `(7)` depends on statement `(8)` while statement `(7)` occurs before statement `(8)`. *Forward* slicing is the opposite of backward slicing; the slice consists of all the statements that are influenced by the slicing criterion. The idea of forward slicing was first defined by Bergeretti and Carré in [15], although Reps and Bricker [52] were the first to use this terminology.

2.2 Program slicing formally

In the previous section program slicing has been introduced in an informal way by showing static backward slicing on an example. Slicing can also be performed programatically, but before algorithms for slicing can be introduced, it is necessarry to define slicing in a formal way. The following sections will give a formal definition for a static backward slice, as well as introduce other types of slicing and provide formal definitions. These definitions follow the definitions outlined in [19], and are based on *state trajectories.* State trajectories capture the states a program goes through during one execution and are defined as follows:

Definition 2.2 (State trajectory). *A* state trajectory *of length k of a program P for input I is a sequence of ordered pairs $T = \langle (p_1, \sigma_1), (p_2, \sigma_2), \ldots, (p_k, \sigma_k) \rangle$ where each pair (p_i, σ_i) represents a state of the program: p_i is a program statement and σ_i is a map which maps each program variable onto their values, just before executing p_i. The sequence T contains the states in the order that the program P passes through them during execution.*

The state trajectory as it is defined in [19], limits itself to finite sequences. The definition proposed here does not have that restriction. When defining slice we work with state trajectories of *complete runs*, which means that the state trajectory either corresponds to a halting run of program P (and is finite), or corresponds to a non-halting run of program P (and is infinite).

2.2.1 Static slicing

The definitions for a static slicing criterion and static backward slicing are as follows:

Definition 2.3 (Static slicing criterion). *A static slicing criterion of a program P consists of a pair $C = \langle p, V_s \rangle$, where p is a statement in P and V_s is a subset of the variables in P.*

In order to limit a state trajectory T to the statement p and the varables V_s given in the static slicing criterion C, the following projection function is used:

Definition 2.4 (Static slicing projection function). *Given a static slicing criterion $C = \langle p, V_s \rangle$ on a program P and a state trajectory $T = \langle (p_1, \sigma_1), (p_2, \sigma_2), \ldots, (p_k, \sigma_k) \rangle$ of an execution of program P, we define for $i = 1, \ldots, k$:*

$$\mathrm{Proj}'_C(p_i, \sigma_i) = \begin{cases} \lambda & \text{if } p_i \neq p \\ \langle (p_i, \sigma_i | V_s) \rangle & \text{if } p_i = p \end{cases}$$

where $\sigma_i | V_s$ is σ_i restricted to the domain V_s and λ is the empty string.

The projection can be extended to the state trajectory T by defining:

$$\mathrm{Proj}_C(T) = \mathrm{Proj}'_C(p_1, \sigma_1) \dots \mathrm{Proj}'_C(p_k, \sigma_k)$$

Note that the number of elements in $\mathrm{Proj}_C(T)$ *is equal to the number of times statement p occurs in a state in T.*

Using these definitions, it is now possible to define a static backward slice:

Definition 2.5 (Static backward slice)**.** *A* static slice *of a program P with respect to static slicing criterion $C = \langle p, V_s \rangle$ is any syntactically correct and executable program P' which is obtained from P by deleting zero or more statements, and for each input I and each state trajectory T corresponding to I which is a complete run of P, and for each state trajectory T' which is a complete run of program P' with input I it must hold that that $Proj_C(T) = Proj_C(T')$.*

Note that according to this definition a static slice can always to be obtained by deleting no statements at all ($P' = P$). In that case, state trajectory T' can be equal to T. A slice is however not unique. Also note that this definition, which is from [19]), differs slightly from the definition given by Weiser in [60]. Weisers definition requires that $\mathrm{Proj}_C(T) = \mathrm{Proj}_{C'}(T')$, where $C' = \langle \mathrm{succ}(p), V_s \rangle$, and $\mathrm{succ}(p)$ is the nearest successor of p in the original program or p itself if p is in the slice. The definition given here and in [19] requires p to be in the slice, while Weisers original definition did not. The reason for this is that a programmer calculating a slice might easily be confused by the fact that the statement in the slicing criterion does not appear in the slice, which can happen when slicing according to Weisers original definition.

The slicing algorithm originally defined by Weiser in [60] only gives valid slices for programs that halt (programs that have finite state trajectories that are complete runs). However, a slicing algorithm exists that is able to preserve behaviour, even for non-halting programs, which is discussed in Chapter 4. Therefore, the definitions in this thesis allow infinite state trajectories and use *complete runs* rather than *halting runs.*

Example 2.1 shows how to calculate a slice with slicing criterion $C = \langle 10, \{\texttt{product}\} \rangle$ in an intuitive way. Example 2.6 demonstrates the way the definition of static backward slices works for one specific I and one specific T.

Example 2.6. *Running* `SumProd` *with input* $I = \{(\texttt{n}, 2)\}$ *results in the following state trajectory T:*

$$\begin{aligned} T = \langle\ & (1, \emptyset), \\ & (2, \{\mathtt{n} \to 2\}), \\ & (3, \{\mathtt{n} \to 2, \mathtt{i} \to 1\}), \\ & (4, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{sum} \to 0\}), \\ & (5, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{sum} \to 0, \mathtt{product} \to 1\}), \\ & (6, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{sum} \to 0, \mathtt{product} \to 1\}), \\ & (7, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{sum} \to 1, \mathtt{product} \to 1\}), \\ & (8, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{sum} \to 1, \mathtt{product} \to 1\}), \\ & (5, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 1, \mathtt{product} \to 1\}), \\ & (6, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 1, \mathtt{product} \to 1\}), \\ & (7, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 3, \mathtt{product} \to 1\}), \\ & (8, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 3, \mathtt{product} \to 2\}), \\ & (5, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 3, \mathtt{product} \to 2\}), \\ & (9, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 3, \mathtt{product} \to 2\}), \\ & (10, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{sum} \to 3, \mathtt{product} \to 2\}) \quad \rangle \end{aligned}$$

The corresponding state trajectory T' which is the state trajectory of SumProd′*, where statements (3), (6) and (9) were deleted from* SumProd*, is as follows, again with input $I = \{(\mathtt{n}, 2)\}$:*

$$\begin{aligned} T' = \langle\ & (1, \emptyset), \\ & (2, \{\mathtt{n} \to 2\}), \\ & (4, \{\mathtt{n} \to 2, \mathtt{i} \to 1\}), \\ & (5, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{product} \to 1\}), \\ & (7, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{product} \to 1\}), \\ & (8, \{\mathtt{n} \to 2, \mathtt{i} \to 1, \mathtt{product} \to 1\}), \\ & (5, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{product} \to 1\}), \\ & (7, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{product} \to 1\}), \\ & (8, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{product} \to 2\}), \\ & (5, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{product} \to 2\}), \\ & (10, \{\mathtt{n} \to 2, \mathtt{i} \to 2, \mathtt{product} \to 2\}) \quad \rangle \end{aligned}$$

The following step is to calculate the projection of these state trajectories:

$$\begin{aligned} Proj_C(T) &= \langle (10, \{\mathtt{product} \to 2\}) \rangle \\ Proj_C(T') &= \langle (10, \{\mathtt{product} \to 2\}) \rangle \end{aligned}$$

Indeed $Proj_C(T)$ is equal to $Proj_C(T')$, which is required if SumProd′ *is to be a static slice of* SumProd *with respect to slicing criterion C. However, in order to prove that* SumProd′ *is a static slice it is required to show that $Proj_C(T) = Proj_C(T')$ for any input I and each corresponding state trajectory T while in this example we have shown that only for one specific input and state trajectory. Completing the proof can done with natural induction on the input value of variable* n*.*

2.2.2 Dynamic slicing

Korel and Laski proposed a different slicing definition [40], which is known as *dynamic slicing.* Whereas static slices are constructed with respect to each possible execution of program P, dynamic slicing only takes one particular execution into account, which is can be described in terms of the input I. Since the input is known, it is possible e.g. to treat each element of an array individually and it is possible to know which variables point to which object. This potentially allows dynamic slicing to yield smaller slices than static slicing.

A dynamic slicing criterion of a program is defined as follows:

Definition 2.7 (Dynamic slicing criterion)**.** *A* dynamic slicing criterion *of a program P consists of a triple $C = \langle I, p, V_s \rangle$, where I is the input of the program P, p is a statement in P and V_s is a subset of the variables in P.*

The projection function from Definition 2.4 can also be used for dynamic slicing. The definition of dynamic slicing is as follows:

Definition 2.8 (Dynamic backward slice)**.** *A dynamic slice of a program P with respect to dynamic slicing criterion $C = \langle I, p, V_s \rangle$ is a syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and for each state trajectory T which is a complete run of program P with input I, and for each state trajectory T' which is a complete run of program P' with input I in must hold that that $Proj_C(T) = Proj_C(T')$.*

The difference between static slices (Definition 2.5) and dynamic slices (Definition 2.8) is that for static slices the projection equality needs to hold for each input, while in dynamic slicing this is only required for one input which is given in the slicing criterion. This means that the state trajectories and projections given in Example 2.6 prove that `SumProd'` is a valid dynamic slice of `SumProd` with respect to slicing criterion $C = \langle 2, 10, \texttt{product} \rangle$.

Example 2.9. *Using Definition 2.8 it can be shown that the static slice is also a valid dynamic slice for the SumProd program, with input $I = \{(\texttt{n}, 2)\}$. However, dynamic slicing has the potential to create smaller slices than static slicing. Consider input $I = \{(\texttt{n}, 0)\}$, slicing criterion $C = \langle I, 10, \texttt{product} \rangle$ and the slice* `SumProd''` *(Program 2.3).*

Program 2.3 `SumProd''`

```
(1)   int n = readInt();
(4)   int product = 1;
(10)  writeInt(product);
```

The state trajectory T of `SumProd` *for input $I = \{(\texttt{n}, 0)\}$:*

$$
\begin{array}{ll}
T = \langle & (1, \emptyset), \\
 & (2, \{n \rightarrow 0\}), \\
 & (3, \{n \rightarrow 0, \texttt{i} \rightarrow 1\}), \\
 & (4, \{n \rightarrow 0, \texttt{i} \rightarrow 1, \texttt{sum} \rightarrow 0\}), \\
 & (5, \{n \rightarrow 0, \texttt{i} \rightarrow 1, \texttt{sum} \rightarrow 0, \texttt{product} \rightarrow 1\}), \\
 & (9, \{n \rightarrow 0, \texttt{i} \rightarrow 1, \texttt{sum} \rightarrow 0, \texttt{product} \rightarrow 1\}), \\
 & (10, \{n \rightarrow 0, \texttt{i} \rightarrow 1, \texttt{sum} \rightarrow 0, \texttt{product} \rightarrow 1\}) \quad \rangle
\end{array}
$$

The state trajectory T' of `SumProd''` *for input I:*

$$\begin{aligned} T = \langle\ & (1, \emptyset), \\ & (4, \{n \to 0\}), \\ & (10, \{n \to 0, \mathtt{i} \to 1, \mathtt{sum} \to 0, \mathtt{product} \to 1\})\quad \rangle \end{aligned}$$

The projection of these state trajectories:

$$\begin{aligned} \mathit{Proj}_C(T) &= \langle (10, \{\mathtt{product} \to 1\}) \rangle \\ \mathit{Proj}_C(T') &= \langle (10, \{\mathtt{product} \to 1\}) \rangle \end{aligned}$$

Since these two projections are equal, `SumProd''` *is indeed a dynamic slice for* `SumProd`*. As can be seen here, dynamic slicing can sometimes lead to smaller slices than static slicing since dynamic slicing limits itself to one input.*

2.2.3 Quasi-static slicing

Venkatesh [56] introduced *quasi-static slicing*. Quasi-static slicing falls between static slicing and dynamic slicing. When performing quasi-static slicing, only some of the inputs have a fixed value and the others may vary. The formal definitions are as follows:

Definition 2.10 (Quasi-static slicing criterion)**.** *A* quasi-static slicing criterion *of a program P consists of a quadruple $C = \langle V_i', I', p, V_s \rangle$, where p is a statement in P; V_i is the set of input variables of a program P and $V_i' \subseteq V_i$; partial input I' is the input for the variables in V_i' and V_s is a subset of the variables in P.*

Definition 2.11 (Input completion)**.** *Let V_i be the set of input variables of a program P and $V_i' \subseteq V_i$ and let I' be an input for the variables in V_i'. A* completion *I of I' is any input for the variables in V_i such that $I'(v) = I(v)$ for each $v \in V_i'$.*

Definition 2.12 (Quasi-static backward slice)**.** *A quasi-static slice of a program P with respect to quasi-static slicing criterion $C = \langle V_i', I', p, V_s \rangle$ is a syntactically correct and executable program P' that is obtained from P by deleting zero or more statements, and for each input I which is a completion of I' and for each state trajectory T which is a complete run of program P with input I, and for each state trajectory T' which is a complete run of P' on input I it must hold that that $\mathit{Proj}_C(T) = \mathit{Proj}_C(T')$.*

In the definitions above, V_i' is the subset of variables for which the input is chosen in I', thus corresponding to the dynamic part of quasi-static slicing. For other variables $(V_i \backslash V_i')$ no input is chosen, thus corresponding to the static part of quasi-static slicing. According to De Lucia in [43] the notion of quasi-static slicing is related to *partial evaluation* or *mixed computation* [39]. When using quasi-static slicing, some statements of the program may be unreachable for the given partial input in the slicing criterion. Which statements are unreachable is determined by attempting to calculate the value of control predicates (statements which have a predicate in them and are able to influence the flow of control) so that certain branches of the original program may be marked as unreachable. This calculation of control predicates can be performed with partial evaluation. Partial evaluation is a technique to specialize programs with respect to partial inputs, allowing the values of the variables for with the value is given to be replaced by constants. These constants may be propagated through the program by using constant propagation [18] and simplification which can cause certain predicates in the program to be calculated. This allows deleting parts of the code from the slice that will not be executed on a particular partial input.

2.3 Other slicing types and concepts related to slicing

In addition to static slicing, dynamic slicing and quasi-static slicing, other types of slicing can be found in the literature. These additional types of slicing are discussed shortly in this section. Also some concepts related to slicing are discussed here as well.

A *conditioned slice* [19] is a subset of the program statements which preserves the behaviour of a program with respect to a slicing criterion which consists of a first order logic formula over the input variables (V_i) of a program, a program statement p and a set of variables V_s. The definition of a conditioned slice is similar to the definition of quasi-static slicing (Definition 2.12), except that only inputs I that satisfy the first order logic formula are taken into account.

Simultaneous dynamic slicing [29] is a form of slicing which computes slices with respect to a set of program executions. The slicing criterion consists of a set of inputs $\{I_1, I_2, \ldots I_k\}$, a program statement p and a set of variables V_s. The definition of a simultaneous dynamic slice is similar to the definition of a dynamic slice (Definition 2.8) where the difference is that the projection equality must hold for each of the inputs $I_1, I_2, \ldots, I_k$ rather than for I.

A *union slice* [16] is the union of the dynamic slices (thus the union of the statements which are included in the slice) of a finite set of inputs. This makes them similar to simulatenous dynamic program slicing. It should however not be confused with simultaneous dynamic slicing, since the union of two dynamic slices does not necessarily form a valid slice, i.e. the behaviour of the original program is preserved in the slice with respect to the slicing criterion [44]. This can be seen by looking at program `Union` (Program 2.4). A valid slice for slicing criterion $\langle 7, \{\mathtt{y}\}\rangle$ consists of statements (1), (2), (3) and (7) and a valid slice for slicing criterion $\langle 7, \{\mathtt{z}\}\rangle$ consists of statements (1), (5), (6) and (7). However, the union of these two slices consisting of statements (1), (2), (3), (5), (6) and (7) is not a valid slice for slicing criterion $\langle 7, \{\mathtt{y}, \mathtt{z}\}\rangle$. The fact that statement (4) is missing causes variable `z` to have the value 4 instead of 3.

Program 2.4 `Union`, program for demonstrating that the union of slices does not need to be a slice [44]

```
(1)      x = 2;
(2)      x = x + 1;
(3)      y = x;
(4)      x = 2;
(5)      x = x + 1;
(6)      z = x;
(7)      ;
```

Amorphous slicing was introduced by Harman, Binkley and Danicic [31]. For the slicing types for which a formal definition was given in Section 2.2, a slice was always obtained by using only statement deletion. Amorphous slicing still preserves the original behaviour of a program with respect to a slicing criterion but allows other types of transformation on the source program. An example is Amorphous (Program 2.5) which is sliced with respect to slicing criterion $\langle 5, \{\texttt{average}\}\rangle$. When performing static slicing, which is limited to statement deletion, this will result in the slice Amorphous$'$ (Program 2.6). However, amorphous slicing allows other transformations to be performed, which leads to the slice Amorphous$''$ (Program 2.7). The bug in calculating the average in program Amorphous is more apparent in slice Amorphous$''$ than in slice Amorphous$'$. Note that slice Amorphous$''$ has a statement which is numbered (4'), since it does not come from the original program but is a modification of statement (4).

Program 2.5 Amorphous, program for demonstrating the practical use of amorphous slicing [31]

```
(1)     for (i=1, biggest=a[0]; i<20; sum=a[++i]) {
(2)         if (a[i] > biggest)
(3)             biggest = a[i]; }
(4)     average = sum/20;
(5)     writeInt(average);
```

Program 2.6 Amorphous$'$

```
(1)     for (i=1, biggest=a[0]; i<20; sum=a[++i]) { }
(4)     average = sum/20;
(5)     writeInt(average);
```

Program 2.7 Amorphous$''$

```
(4')    average = a[19]/20;
(5)     writeInt(average);
```

Constrained slicing was introduced by Field, Ramalingam and Tip [25]. A constrained slice is a slice that preserves behaviour for all inputs that satisfy a given set of constraints. A fully constrained slice is a dynamic slice, while a fully *un*constrained slice is static slice. Quasi-static slices, introduced by Venkatesh in [56], are simular to constrained slices, but no description of how to calculate quasi-static slices is given in [56]. Constrained slicing can be implemented by converting the program to be sliced into an intermediate representation PIM and makes use of graph rewriting techniques for slicing. Slices obtained from constrained slicing are in general not syntactically valid.

Hybrid slicing introduced by Gupta, Soffa and Howard in [28] is another slicing type which falls between static and dynamic slicing. Hybrid slicing makes use of both static information and dynamic information which available when executing a program. The dynamic information is collected by recording which procedure call and return points were passed and which user selected breakpoints were passed during execution until the statement in the slicing criterion was reached. This information, in addition to the information available during static slicing, has the potential to be used to create smaller slices since it may be possible to rule out some execution paths.

Dicing was introduced by Weiser and Lyle in [61]. A program dice is defined as the difference between the static slices of a variable which appears to have an incorrect value and one that has a correct value during testing, thus the statements that have an influence on the calculation of the incorrect value but do not have an influence on the calculation of a correct value. Also, a variant based on dynamic slicing, which is called *dynamic program dicing*, was introduced by Chen and Cheung in [20].

Chopping, introduced by Jackson and Rollins in [37]. Chopping is a generalization of slicing, which has a slicing criterion which consists of two parts: the source which consists of a set of definitions of variables and the sink which consists of a set of uses of variables. Chopping a program (calculating a *chop*) identifies all statements through which the source has influence on the sink. Chopping is limited to a single procedure. Chopping is a generalization of forward and backward slicing. Also the calculation of a chop can be expressed as a combination of intersections and unions of forward and backward slices. A backward slice can be calculated by chopping where the sink consists of the backward slicing criterion and the source consists of each statement and variable in the program. A forward slice, likewise, can be calculated by chopping where the source consists of the forward slicing criterion and the sink consists of each statement and variable in the program.

Closure slices are a different way to look at slicing, where closure slices can be seen as an alternative to executable slices [56]. Closure slices contain the set of statements that are related to the variable of interest (the variable in the slicing criterion) by *data dependence* (the term data dependence is defined in Section 4.2). Closure slices are calculated by performing a closure over the data dependencies, starting at the variable of interest. Closure slices are not necessarily executable programs. A definition for dynamic closure slices was given in [12] by Agrawal and Horgan, and was extended to static slicing in [56] by Venkatesh.

2.4 Applications of slicing

The application Weiser originally had in mind was debugging [59, 60]. If the value computed for a variable contains a wrong value, it is very likely that the slice with respect to that variable and the statement where that variable is calculated contains the wrong value.

Slicing can also be used to assist in the reverse engineering of the implementation of a software system [14]. *Reverse engineering* is the process of analyzing a system to identify the components of the software system and their interrelations. The result of this analysis is represented in another form at a higher level of abstraction. Slicing is not used to perform reverse engineering, but slicing is able to make reverse engineering easier because slicing can extract statements which are related to each other (since they depend on each other), which allows these statements to be replaced by one high-level statement. This will allow the program to be viewed at a higher level of abstraction.

Slicing can be used to calculate *cohesion* and *coupling* metrics in software measurement. *Cohesion* is a metric which measures the relatedness of one program unit (such as a Java class), where high cohesion is seen as a desirable property of a program [48]. *Coupling* is the measure of how one unit depends on or affects another unit. In [32] Harman et al. proposed a method to employ program slicing to calculate coupling metrics.

Finally, slicing can be used as an abstraction method used to combat the state space explosion problem in model checking prior to applying model checking on programs [34, 57].

More information about the applications of slicing can be found in [54] and [23].

2.5 Undecideability of slicing

In order to get slicing as useful as possible as an abstraction method, it is interesting to calculate a *minimal slice*, which is a slice that is as small as possible. The meaning of "as small as possible" can be defined in several ways, for example a *statement-minimal slice*, which is a slice with the minimum number of statements. Calculating a statement-minimal slice is unfortunately undecidable [45,50,60]. Statement-minimal slices do not need to be unique [44]. In program `Minimal` (Program 2.8) there are multiple minimal slices when slicing with respect to slicing criterion $\langle 6, \{x\} \rangle$. One valid slice consists of statements (1), (2) and (6); another valid slice consists of statements (1), (5) and (6). These two slices are not the only two, but this is already sufficient to show that a statement-minimal slice does not have to be unique.

Program 2.8 `Minimal`, a program used to demonstrate that minimal slices do not need to be unique

```
(1)       x = 2;
(2)       x = x + 1;
(3)       y = x;
(4)       x = 2;
(5)       x = x + 1;
(6)       y = x;
```

There exist algorithms which are able to calculate practically small slices. These algorithms are in general not able to calculate statement-minimal slices, but instead perform approximations. These algorithms attempt to calculate slices which are as close to statement-minimal slice as possible, but when it is not possible to decide if a statement must be in the slice or not they will include the statement (conservative approach). Algorithms calculating such approximations are introduced in Chapter 4.

Chapter 3

Challenges and current implementations of Java slicing

As mentioned in Chapter 1, the goal of this project is to build a slicing platform for Java that supports Java 6 and is able to slice complete and incomplete programs. The output should be given in the form of syntactically correct Java programs. In Chapter 2 the concept of slicing was introduced and demonstrated both intuitively and formally. Static, dynamic and quasi-static slicing were introduced. Since the slicer should be able to work with incomplete programs, the program can not be executed and thus only static slicing can be used.

3.1 Challenges in static slicing Java programs

Static slicers are limited to information that can be derived at compile time from the program text. When slicing a program like `SumProd`, this does not cause any problems. However, Java programs in general consist of multiple procedures and use more Java features than `SumProd`, like arrays and objects. This chapter shows which challenges occur when slicing programs that use more advanced Java features. This chapter also shows the conservative decisions a slicer may need to make.

3.1.1 Procedures

The `SumProd` program only consisted of one procedure. However, Java programs consist of classes which contain members such as fields and methods. Dependence of statements on other statements needs to be tracked through different methods, which means that the values of formal parameters of the called method depend on the actual parameters at the call site. Also, if the return value of a method call is used, this usage depends on the expression stated in the return-statement of the method called.

Consider program `Procedures` (Program 3.1) with slicing criterion $\langle 14, \{\texttt{b}\}\rangle$. The value of variable `b` depends on the return value of `func(y)` in statement `(12)`. Therefore, statement `(4)` is in the slice as well. Statement `(4)` uses the value of parameter `p` which has the value of variable `y` of method `main` adding statement `(9)` to the slice as well.

Program 3.1 `Procedures`

```
(1)     public class Procedures {
(2)         public static int func(int p)
(3)         {
(4)             return p + 1;
(5)         }
(6)
(7)         public static void main(String[] args) {
(8)             int x = 1;
(9)             int y = 2;
(10)
(11)            int a = func(x);
(12)            int b = func(y);
(13)
(14)            System.out.println(b);
(15)        }
(16)    }
```

3.1.2 Dynamic dispatch

Consider the program `DynamicDispatch` (Program 3.2) and slicing criterion $\langle 30, \{x\}\rangle$. This program in used to show in which way the dynamic dispatch of Java makes slicing more complicated. The value of `x` depends on the `setx` method. However, since `obj` is a variable of type `A` and class `B` is a subclass of `A`, the object that variable `obj` refers to can be either of type `A` or `B`. This means that the call to `setx` on statement (28) can be a call to either the `setx` defined in class `A` or `B`. This influences the slice, since the value of `x` in the slicing criterion can be either from statement (7) or statement (20). When the actual type of `obj` is not known, we again need to use a conservative approach and include both statements (7) and (20) in the slice.

3.1.3 Arrays

Consider the `ArrayAssign` program (Program 3.3), with slicing criterion $\langle 6, \{\texttt{x}\}\rangle$. When we only look at the source code, the value of variable `i` is not known, thus variable `x` can contain either `0` or `1`. When attempting to slice, we naturally need to include statement (5). After that it becomes tricky, since the value of `a[1]` will come from the assignment of statement (4) if `i` is equal to `1` or from statement (3) if `i` is not equal to `1`. This issue can not be decided without knowing the value of variable `i`, thus in order to get a slice we need to take a conservative approach and simply include both statements. Completing the slice, will lead to a slice which consists of the entire program (no statement could be deleted).

3.1.4 Object aliasing

Next, consider the `Alias` program (Program 3.4), and the slicing criterion $\langle 14, \{x\}\rangle$. When calculating the slice, statement (13) is naturally included. After that one might be tempted to say that statement (10) should be included since `a.x` is assigned a value there, and statement

Program 3.2 `DynamicDispatch`

```
class A
{
    protected int x;

    void setx()
    {
        x = 37;
    }

    int getx()
    {
        return x;
    }
}

class B extends A
{
    void setx()
    {
        x = 42;
    }
}

class X
{
    static void method(A obj)
    {
        obj.setx();
        int x = obj.getx();
        System.out.println(x);
    }
}
```

Program 3.3 `ArrayAssign` [41]

```
int x,i = readInt();
int[] a = new int[20];
a[1] = 0;
a[i] = 1;
x = a[1];
writeInt(x);
```

(11) does not need to be included. However, variables a and b do not contain objects but rather references to objects. This is called *aliasing*. Thus a and b may refer to the same object. In that case, statement (11) must be included but statement (10) can be left out, since the value assigned there to a.x is replaced in statement (11) before being used. If we can know for sure that a and b are either aliased or not aliased, then we can leave out either statement (10) or (11). However, if this can not be determined, it is necessary to use a conservative (safe) approach and leave both statements (10) and (11) in the slice.

Program 3.4 Alias

```
(1)     class T {
(2)         int x;
(3)     }
(4)
(5)     class A {
(6)         static int method(T a,T b)
(7)         {
(8)             int x;
(9)
(10)            a.x = 10;
(11)            b.x = 20;
(12)
(13)            x = a.x;
(14)            return x;
(15)        }
(16)    }
```

3.1.5 Threads

Things become more complicated when threads are used, as can be seen in the Threads program (Program 3.5). This program creates one object t of type T which is passed to both threads that the program creates during execution. This means that the field a.t and b.t both refer to the same object of type T. When slicing with respect to slicing criterion $\langle 17, \{x\} \rangle$, it is apparent that statement (16) and (15) are included as well. Statement (15) assigns but also makes use of the previous value of t.x. Since it has two threads which are active at the same time, this value can come from either statement (14) or from the other thread, from statement (32). In which way the statements of the first thread (statements (14) through (17)) and the second thread (statement (32)) are interleaved, is determined by the Java runtime system which gives us no guarantee about which order will be chosen. Therefore both statement (14) and statement (32) must be included in the slice.

3.1.6 Reflection and dynamic class loading

Reflection is a feature of Java which allows a Java program to inspect its own classes and their members. Program Reflection (Program 3.6) shows a program which takes the name of a class from the command line, uses reflection to find a constructor in that class which has two integers as parameters and creates an object of that type. Using only statically available

Program 3.5 Threads

```
class T { int x; }

class A implements Runnable
{
    private T t;

    public A(T t)
    {
        this.t = t;
    }

    public void run()
    {
        t.x = 2;
        t.x = t.x + 3;
        int x = t.x;
        System.out.println(x);
    }
}

class B implements Runnable
{
    private T t;

    public B(T t)
    {
        this.t = t;
    }

    public void run()
    {
        t.x = 10;
    }
}

class Main
{
    public static void main(String[] s)
    {
        T t = new T();
        A a = new A(t);
        B b = new B(t);
        new Thread(a).start();
        new Thread(b).start();
    }
}
```

information, a slicer has no information on which class is accessed on statement (7), thus there is no way to know which constructor is invoked on statement (15). A static slicer could be very conservative and include every constructor, but this is very likely to lead to very large slices containing lots of irrelevant nodes which degrades the usefulness of the slice.

Dynamic class loading makes slicing with only static information even more complicated. Dynamic class loading allows loading classes during runtime where the name of the class is specified in a string. This allows a Java program to introduce classes that were not known to exist to the static slicer during the slicing process. Since slicing incomplete programs is possible, the dynamically loaded classes can be considered to be missing.

As mentioned above, programs that make use of reflection and dynamic class loading can not (reasonable) be sliced by a static slicer, so the best thing a static slicer can do is simply report the usage of reflection and/or dynamic class loading. Even though the slicer might report such problems, it can continue to calculate a slice where the statement that use reflection and/or dynamic class loading are simply ignored.

Program 3.6 `Reflection`, a program creating a class of a type given by a string

```
(1)     import java.lang.reflect.*;
(2)
(3)     public class Main {
(4)         public static void main(String args[])
(5)         {
(6)         try {
(7)             Class cls = Class.forName(args[0]);
(8)             Class partypes[] = new Class[2];
(9)             partypes[0] = Integer.TYPE;
(10)            partypes[1] = Integer.TYPE;
(11)            Constructor ct = cls.getConstructor(partypes);
(12)            Object arglist[] = new Object[2];
(13)            arglist[0] = new Integer(37);
(14)            arglist[1] = new Integer(47);
(15)            Object retobj = ct.newInstance(arglist);
(16)        } catch (Throwable e) { System.err.println(e); }
(17)        }
(18)    }
```

3.2 New Java language features

In Java 5 several new language features were introduced [27,47]. In Java 6, no new language features were introduced [22,62].

In the subsections below, the new features will be discussed. Most of the language features new in Java 5 can be seen as 'syntactic sugar' and can therefore be expressed in Java 1.4 ('de-sugaring'). However, just de-sugaring may not always be a good idea since this might throw away information and thus make the slice less precise.

3.2.1 Generics

With the addition of generics in Java 5 it is now possible to make a class more generic since when you declare a variable or parameter with a certain class you can add a class as parameter. This prevents a lot of type casting when for example using collections. To show in what way generics are de-sugared, example Program 3.7 is used.

Program 3.7 Example using Java generics

```
class GenericClass<T>
{
    T value;

    GenericClass(T value)
    {
        this.value = value;
    }

    public String toString()
    {
        return value.toString();
    }

    T get()
    {
        return value;
    }
};

class GenericUser
{
    static public void main(String[] args)
    {
        GenericClass<Integer> b;

        b = new GenericClass<Integer>(15);
        System.out.println(b);

        Integer c = b.get();
        System.out.println(c);
    }
};
```

When Program 3.7 is compiled, the class `GenericClass<T>` will result in a normal class called `GenericClass`. In this class, every occurence of `T` will be replaced by `Object`. When we look at the `GenericUser` class, a class that creates an object of the type `GenericClass<Integer>`, it will actually simply create an object of the type `GenericClass`, with a parameter `15`. This `15` will be converted in an Integer object (this uses the autoboxing feature) which is of course of the type `Object`. Thus for methods that take something of type `T` as a parameter, no modifcations are necessarry to make this work. This is different when a method returns something of the type `T` like with method `get`. In this case, the compiler adds a typecast to `Integer`, in order to make it work. The same holds when getting the value of a field from `GenericClass`.

In this way, generics are completely replaced by constructs that are also in Java 1.4, so programs that work on class files do not have to know anything about generics. Arguably, however, the precision of slices might be compromised by the fact that we throw away information about the type `T` in `GenericClass`. When `GenericClass` makes use of mehods of `T`, after de-sugaring, the slicer will know no better than `T` is of the type `Object` (or a class deriving from `Object`, which includes every class). In such a generic class, it is not possible to use methods that Object does not have, but for a slicer it might be interesting to know more about `T`. In the example above, the slicer can assume that `T` is of the type Integer since only `GenericClass<Integer>` is used, but no other type for `T`.

3.2.2 Enhanced for loop

This new language feature makes it easier to loop over all elements of a collection by adding some syntactic sugar which takes care of creating an iterator for the collection class and advancing this iterator to the next element. In Java 5 you can use the construct in Program 3.8. When de-sugaring this, it will expand to the code shown in Program 3.9.

Program 3.8 Example with an enhanced for loop

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask t : c)
        t.cancel();
}
```

Program 3.9 Enhanced for loop, desugared

```
void cancelAll(Collection<TimerTask> c) {
    for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); ) {
        TimerTask t = i.next();
        t.cancel();
    }
}
```

Note that in Java 5, collection classes were modified to implement the Iterable interface, an interface that was introduced in the Java 5 class library [27]. Only classes that implement this interface can be used in an enhanced `for` loop.

When de-sugaring enhanced ,for, loops, an extra statement and a few extra expressions are introduced. These could all be linked to the original `for` statement. However, if `t` in the body is not used because none of the statements in the body of the enhanced for loop is in the slice, the entire `for` could be sliced away. In that case, the slicer should be able to see that creating the iterator is not be needed. This might be easier, if the enhanced for loop is not de-sugared. However, it should be taken into account that the slicer can also be used to slice pre-Java 5 code which does not use enhanced for loops or generics, but rather does it in the old-fashioned way shown in Program 3.9.

Code using the old-fashioned way of iterating should also be sliced correctly, so it might be a good idea to de-sugar enhanced for loops anyway and make sure that the slicer is able to drop the entire ,for, if it is unused.

3.2.3 Autoboxing

When using collections, it is impossible to include primitive types like `int` into a collection since only objects are allowed. Thus, before placing an int in a collection it has to be converted to the object `Integer` and when reading from the collection it has to be converted back before you can use the value as a normal `int`. Autoboxing (introduced in Java 5) makes sure that these cumbersome operations are done automatically, rather than by hand. Each of the primitive data types has a wrapper object type (like `Integer` for `int`). Autoboxing means that conversion between these corresponding pairs of primitive types and wrapper object types will be done automatically when needed. The boxing and unboxing will be translated into method calls to methods which are known to be in the standard Java library. Therefore, de-sugaring autoboxing does not give problems when slicing.

3.2.4 Typesafe enumerations

Before Java 5 there was no true enumeration type, which lead to the usage of the int Enum pattern which consisted of the enumeration values being declared as constant integers. Since they were not true enumerations but rather integers this lead to problems, since the type checker in the Java compiler does not distinguish between different types of booleans. Since 5 there is a dedicated enumeration type.

The typesafe enums can also be seen as syntactic sugar. Defining a typesafe enum results in the definition of a complete new class, which is a subclass of java.lang.Enum. An example of an typesafe enum definition in a class is shown as Program 3.10.

Program 3.10 Example showing typesafe enumerations

```
class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
                                    NINE, TEN, JACK, QUEEN, KING, ACE }
}
```

It will lead to an inner class of `Card` (`Card$Rank` which will inherit from `java.lang.Enum`, and contains static objects of type `Card$Rank` for each of the options in the enum (e.g. `Card$Rank.DEUCE`, `Card$Rank.THREE`, ...). Since enums will actually have the type `Card$Rank` rather than just being an `int`, they will be typesafe. However, the amount of overhead is increased since these `enum`s are classes rather than `int`s. It is also possible to add methods to an `enum`, where these methods will also end up in the inner class.

3.2.5 Varargs

This new feature allows you to create methods which take a variable number of parameters. Before 5, when a method took a variable amount of parameters this could only be done by placing all arguments into an array and passing that as a parameter. The new varargs feature in Java allows methods to have a variable amount of parameters, where placing the arguments in an array is automatically done by the compiler rather than requiring the programmer to do that manually. Whether or not this can be safely de-sugared depends on how well the slicer can work with arrays. If the slicer treats an array as one object, then it could be a good idea not to de-sugar since then the slicer would not be able to determine that some of the parameters from the variable list are not used. If the slicer can handle arrays properly (treat each element of the array as a seperate piece of data) then de-sugaring can be done without losing slice precision.

3.2.6 Static import

This allows you to use the static import directive to make it easier to use static members from classes. For example: before you used the `sqrt` method from the Math class, you had to write `Math.sqrt` each time. With a static import this can be reduced to just writing `sqrt`. The lack of static import lead to the constant interface antipattern, where static members where placed into interfaces which essentially abuses interfaces.

Static import is easy to translate to Java 1.4, since it only involves adding the class name of the statically imported class. This is a simple name resolution process, and will not affect the precision of the slice.

3.2.7 Annotations

Annotations are constructs in Java source code that start with an @-sign. They are used for different purposes, like adding metadata which is required for *Enterprise JavaBeans (EJB)* [3]. Annotations allow adding the metadata in the Java source code itself, rather than requiring seperate files which refer to the source code, as in older versions of EJB. A slicer that supports EJB should take EJB related annotations into account, otherwise it should just ignore them. Annotations unrelated to EJB can be ignored by the slicer.

3.2.8 'Syntactic sugar'

Except for annotations, these language features can be seen as 'syntactic sugar', which means that these features can be represented in Java 1.4 language features [27]. Different tools exist, which enable usage of Java 5 language features on Java 1.4 systems:

- The standard java compiler (javac) has a mode, the so called JSR 14 target mode, which will compile source code which uses Java 5 language features into class files which are Java 1.4 compatible. However, not all new features are supported. The generics, varargs and autoboxing features are supported. The enhanced for-loop can only be used on classes that derive from the Collection class in the Java standard library. Enumerations are not supported.

- Retroweaver and Retrotranslator are two tools which are each capable of converting class files compiled for source that uses Java 5 language features to class files which are Java 1.4 compatible. Retrotranslator is the most capable of the two.

Unfortunately, slicing source code which uses new language features can not simply be done by converting Java source or class files to Java 1.4. The JSR 14 target mode is not usable, since this mode does not fully support Java 5. Retroweaver and Retrotranslator can not be used, since they work on class files that do not contain enough information to produce syntactically correct programs.

3.3 Existing slicers for Java

In the Section 3.1 we have seen which challenges are encountered in static slicing in Java. To my knowledge, there exist three slicers capable of slicing Java programs. These are JSlice, VALSOFT/Joana and Indus Kaveri.

3.3.1 JSlice

JSlice [7] is a dynamic Java slicer, which is based on an existing alternative Java virtual machine (Kaffe [8]), so the slicer actually is a modified Java virtual machine. It determines the slice by actually running the Java program. JSlice is a dynamic slicer and makes use of a method described in [58] which represents traces through the bytecode of a Java program in a very compact, efficient way. Since this slicer is dynamic, its approach of the slicing problem is completely different from a static Java slicer. This makes JSlice unsuitable to base the slicing platform on. It has not been updated since 2008.

3.3.2 VALSOFT/Joana

VALSOFT/Joana [11] is a framework which is able to perform software analysis on Java programs. It contains a slicer which is able to perform backward static slicing on program given to VALSOFT/Joana in the form of Java class files. The version tested here (the version from August 2, 2010) is a preliminary and incomplete version which comes with a disclaimer stating that it is a work in progress and that it is likely to contain errors.

In order to use the slicer, perform the following steps:

1. Compile the Java source as you normally would, which results in class files
2. Use VALSOFT/Joana to transform the class files into sdg files (files that contain system dependency graphs)
3. Perform slicing on the the sdg file. The slicing criterion consists of the name of the source file and a line number. Note that no program variables are specified, the set of variables associated with the slicing criterion is the set of variables of which the value is used at that statement.

Calculating multiple slices on the same program is possible by repeating step 3 for each slicing criterion. the system dependence graph generation of step 2 only has to be performed once, which is an advantage since step 2 is the most time consuming.

The output of the VALSOFT/Joana slicer is a list of line numbers, accompanied with file names (in the form `Filename:Line number`). These line numbers are the line numbers of the statements that are part of the slice. When multiple statements reside on one line, they can not be distinguished. Also, the output of the slice is limited to statements which is insufficient to derive a syntactically correct program from the output of Joana. This limitation is probably caused by the fact that Joana works on class files which do contain information which links statements to line numbers, but lacks information about positions on lines and also lacks information about other constructs required to make a syntactically correct program like field declarations.

I tested Joana on the `SumProd` program (Program 2.1). This required writing implementations for the `readInt` and `writeInt` methods. The contents of these methods will be ignored here, since these methods have no side effects other than reading from the standard input or writing to the standard output. The slice produced by Joana when slicing `SumProd` with the slicing criterion set to the line corresponding to statement `(10)` is shown as Program 3.11. Note that the slice also included statements from the `readInt` and `writeInt` methods and statements from the Java Runtime Environment. These methods are both omittted from the slice shown in program.

Program 3.11 `SumProd`, sliced by VALSOFT/Joana with respect to slicing criterion `(10)`

```
(1)   int n = readInt();
(5)   while (i <= n) {
(6)       sum += i;
(7)       product *= i;
(8)       i++; }
(9)   writeInt(sum);
(10)  writeInt(product);
```

The result differs significantly from my expectations (the slice calculated intuitively in Section 2.1). The statements assigning `i`, `sum` and `product` their initial value are absent from the slice while statements `(6)` and `(9)` are still present. It is clear that the program does not reproduce the behaviour of the original program.

3.3.3 Indus Kaveri

Indus [5] is a framework for static analysis of Java source code and Kaveri [9] is a plugin for Eclipse [2] which adds a graphical user interface to Eclipse from which the Java slicer provided by Indus can be used. Running Indus Kaveri, however, is a bit difficult since it requires you to install an old version of Eclipse (3.1 or 3.2). In order to run Indus Kaveri, you must have Java 5 or higher installed since Indus Kaveri itself makes use of Java 5 language features (even though it can not slice programs that use them). It also requires Java 1.4 and requires that the project to be sliced is configurated to use Java Runtime Environment (JRE) 1.4. This can be omitted if you only want to slice programs that make no use of the Java standard libraries. Of course, few real-world programs fall into that category. Indus Kaveri only supports slicing Java up to version 4, and does not support dynamic class loading, reflection and native methods. These two problems mean that Indus Kaveri is now somewhat difficult to install and can not be used to slice Java programs that use Java features introduced after Java 1.4. Of course, since it is open source it is still interesting for code and idea reuse.

Most slicing related work is based on system dependence graphs, but Indus does not actually generate a complete system dependence graph, but rather generates it on the fly [38, 51]. Indus is not strictly a slicer, but a framework for static Java code analysis. The advantage of not generating the entire system dependence graph, is the fact that the amount of data that must be maintained in memory for slicing is much less than when the entire system dependence graph must be kept in memory.

Indus consists of three parts:

StaticAnalyses Provides low-level static analysis such as object flow analysis, which can be used by more high-level static analysis.

Indus Provides more high-level static analysis such as call graph analysis, and also program transformations.

Slicer Makes use of the analysis methods provided by the above libraries to calculate backward and forward slices.

Indus (the entire Indus project, thus also including StaticAnalyses and Slicer) is based on the Soot framework [10], which is a Java optimization framework. The Soot framework is capable of reading Java classes in the form of either Java source or Java class files. Soot converts the source files into an intermediate form named Jimple.

The instructions in Jimple are very simple, and they resemble Java bytecode quite closely. The difference, however, is that Jimple is not stack-based (like Java bytecode), but Jimple instructions work on variables. For example a Jimple instruction to add two integers has three parameters, two input and one output. Adding three or more variables requires multiple Jimple instructions, since one add instruction can only add two values. One Java statement in general consists of more than one Jimple statement. Slicing is performed on the level of these Jimple instructions rather than program statements. This means that the slicing criterion can consist of Java statements, but it is also possible to pick one or more specific Jimple instructions allowing a fine-grained slicing criterion selection.

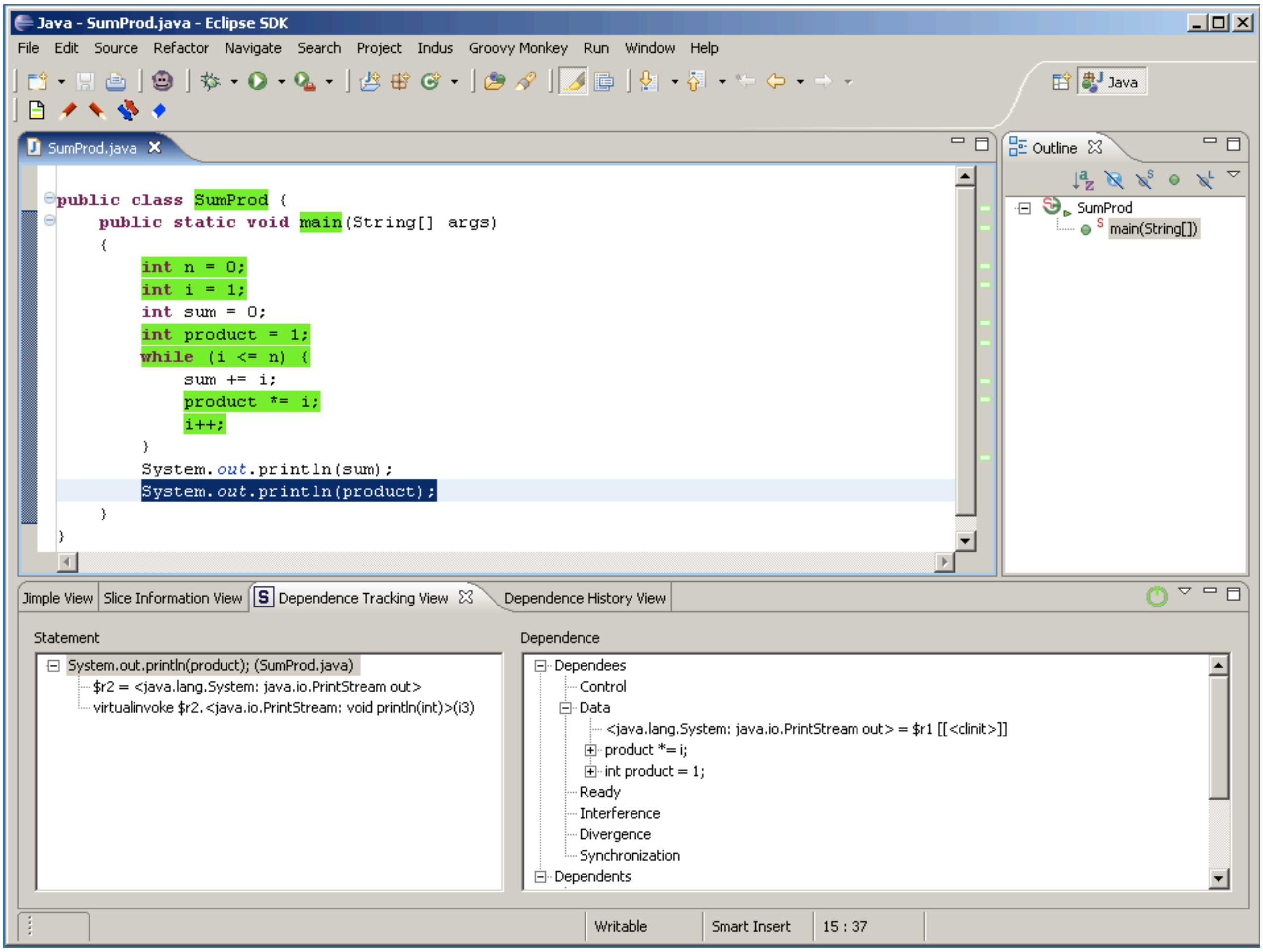
Kaveri is an Eclipse plugin which adds a graphical front-end to the slicer (Figure 3.1) provided by Indus, in such a way that it can be used from Eclipse. Kaveri allows the user to select which classes should be sliced, so you can choose to limit slicing to a subset of classes.

The Soot framework keeps a mapping of Jimple statements to line numbers. A single line in the source code can have zero, one or multiple statements associated to it. If all Jimple statements associated with a line are in the slice (and the line has at least one Jimple statement associated), the line is highlighted in green. If only some of the Jimple statements are in the slice, then the line is highlighted in yellow. Otherwise, the line is not highlighted.

The Indus framework is used by the Bandera framework [1], which is a framework that is capable of extracting models for model checking [35] from Java programs. The slicer of Indus reduces the program by slicing prior to generating the models for model checking, in order to reduce the size of the state space.

3.4 Software requirements

The software requirements of the slicing platform, based on the user requirements given in the introduction (Chapter 1).



```
public class SumProd {
    public static void main(String[] args)
    {
        int n = 0;
        int i = 1;
        int sum = 0;
        int product = 1;
        while (i <= n) {
            sum += i;
            product *= i;
            i++;
        }
        System.out.println(sum);
        System.out.println(product);
    }
}
```

```
System.out.println(product); (SumProd.java)
    $r2 = <java.lang.System: java.io.PrintStream out>
    virtualinvoke $r2.<java.io.PrintStream: void println(int)>(i3)
```

```
Dependees
    Control
    Data
        <java.lang.System: java.io.PrintStream out> = $r1 [[<clinit>]]
        product *= i;
        int product = 1;
    Ready
    Interference
    Divergence
    Synchronization
Dependents
```

Figure 3.1: Screenshot of Indus Kaveri slicing `SumProd` with respect to slicing criterion `System.out.println(product)`

ID	Requirement	Priority
SR1	Calculate backward static slices of sequential Java programs	1
SR2	Calculate forward static slices of sequential Java programs	2
SR3	Support for Java 6 language features (Section 3.2)	1
SR4	Output slices in the form of syntactically correct Java source	1
SR5	Calculate slices of concurrent Java programs	3
SR6	Visualize slices by marking the statements in the text	3

Each software requirement has a priority in the range 1-3. The priorities have the following interpretations:

1: The requirement must be implemented

2: It is desirable that the requirement is implemented

3: The requirent does not have to be implemented, but must be taken into account in the design

In terms of the challenges from Section 3.1, software requirement SR1 includes support for correct slicing of programs making use of multiple procedures, dynamic dispatch, arrays, objects that are potentially aliased. It does not include threads since this is part of SR5. Reflection and dynamic class loading also are not part of SR1 (or any of the other software requirements), since proper slicing with only static information is infeasible.

The following table shows which of the software requirements are met by the existing slicers, discussed in Section 3.3. Note that JSlice is omitted, since it is a dynamic slicer rather than a static slicer.

Slicer	SR1	SR2	SR3	SR4	SR5	SR6
VALSOFT/Joana	Yes [1]	No	Yes	No	Yes	No
Indus Kaveri	Yes	Yes	No	No	Yes	Yes

None of the two slicers fulfill all software requirements. Since VALSOFT/Joana does not fulfill three software requirements and is a work in progress, it does not seem like a good plan to base the slicing platform on VALSOFT/Joana.

Indus Kaveri only misses two software requirements, namely Indus Kaveri does not support Java 6 (only Java 1.4) and is not able to output slices in the form of syntactically correct Java source.

In order to output syntactically correct Java source, the mapping of Jimple statements to source code needs to be more precise than it is now (more detailed information of this can be found in Section 5.1). Since these mappings are created by Soot, this requires modifications to Soot.

The lack of support for Java 6 is caused by the fact that Indus Kaveri uses version 2.1.0 of the Soot framework to parse the Java source code and Soot 2.1.0 only supports Java 1.4. In order to let Indus Kaveri support Java 6, Indus Kaveri must be recompiled with Soot 2.4.0 which does support Java 6. From the forums of Indus Kaveri it becomes clear that multiple developers already tried to compile Indus Kaveri with Soot 2.4.0, but were not succesful [6]. My own attempts to compile Indus Kaveri with Soot 2.4.0 did not lead to a working slicer (it

[1] Note that the slice given by the tested version VALSOFT/Joana for `SumProd` is incorrect, which is likely caused by the fact that this version is a preliminary version rather than a release.

always gave an exception during slicing) and my question about this on the forums remained unanswered.

Since the attempts to get Indus Kaveri to work with Soot 2.4.0 (thus supporting Java 6) were unsuccesful, another solution had to be found. Instead of basing the slicing platform on an existing slicer, the slicing platform is directly based on Soot 2.4.0. Since Soot provides an intermediate representation but does not provide a slicer, a slicing platform with multiple slicing algorithms is implemented as part of this project. Chapter 4 provides an overview of existing slicing algorithms and is followed by chapter 5 that goes into depth about the slicing implementation based on the Soot framework.

Chapter 4

Static slicing algorithms

Chapter 2 gave the definition of static slicing and shows that it is very easy to get a *valid* slice by not deleting any statements. However calculating a *statement-minimal* slice is an undecideable problem. Fortunately, there exist algorithms which potentially allow to calculate *practically small* slices, even though these slices may not be statement minimal. This chapter provides a comparison between the slicing algorithms found in the literature, in order to determine which of these algorithms is the most suitable for Java programs. After that, we will go into more depth on the chosen algorithm, revealing in which way this algorithm applies to slicing Java programs.

4.1 Comparison of static slicing algorithms

The survey papers about slicing [54, 63] compare three slicing algorithms. All of these algorithms are capable of calculating slices on 'simple' programs where a 'simple' program is a program that consists of only one procedure, has no unstructured control flow statements (jumps like `break`, `continue` and exceptions), only has scalar variables and has no concurrency. Some of these slicing algorithms are capable of slicing programs that employ features that are excluded from 'simple' programs.

The *dataflow equations* are the equations originally defined by Weiser [60] in order to calculate backward slices. Weisers algorithm consists of calculating the set $R_C(n)$, which is defined in these dataflow equations in an iterative way. Here $C = \langle p, V \rangle$ is the slicing criterion and n a statement of the program. The sets $R_C(n)$ for different statements n represent the sets of variables that are relevant to the slicing criterion. If variable $v \in R_C(n)$ then the value of v during execution of statement n may influence one of the variables in V during execution of statement p. The slice follows from these sets, because if a statement n defines a variable `v` that is relevant to the slicing criterion (i.e. $\mathtt{v} \in R_C(n)$), then statement n must be in the slice.

The *information-flow relations* were defined by Bergeretti and Carré [15], and are capable of calculating backward and forward slices. The information-flow relations are derived from a program in a syntax-directed fashion, and are defined for empty statements, assignment statements, conditional statements, loop statements and sequential composition of statements. The relations specify for each program variable, which statements can have an influence on the variable. The algorithm calculates the information-flow relations, and derives a backward or forward slice from these relations.

	Dataflow equations	**Information-flow relations**	**Dependence graph**
Unstructured control flow	Yes	No	Yes
Procedures	Yes	Yes	Yes
Composite data types and aliasing	Yes	No	Yes
Concurrency	No	No	Yes
Preserving non-termination	No	No	Yes

Table 4.1: Comparison of the three slicing algorithms

Karl Ottenstein and Linda Ottenstein [49] were the first to define slicing as a reachability problem in a *dependence graph*, a graph that is derived from the input program. Each statement of the original program has a node in the dependence graph. The arcs in the graph denote where a statement depends on values calculated in another statement. The algorithm first constructs the dependence graph, then calculates the slice by solving a reachability problem in the graph.

Table 4.1 presents a comparison of the capabilities of the slicing algorithms, which is based on the survey papers of Tip [54] and Xu et al. [63]. Java programs may have unstructured control flow (in the form of `goto`, `break`, `continue` and exceptions), in general consist of multiple procedures (methods), have compositie data types and aliasing (object instances) and have concurrency. Thus a slicing algorithm capable of slicing Java 6 programs is required to support each of these features. It is also an advantage if the slicing algorithm supports non-termination. From the comparison in table 4.1 follows that the dependence graph based algorithm is the only algorithm suitable for slicing Java 6 programs. It also support non-termination which is also an advantage. Therefore, this algorithm is the only algorithm that we discuss from this point on.

4.2 Program dependence graph

This section introduces the *program dependence graph (PDG)*, which is a directed graph that has a node for each statement in the program and has edges between nodes to indicate depence between nodes. Each statement in the program corresponds to a node in the PDG.

A PDG corresponds to a single procedure of a program (or to the entire program, in case of a non-procedural program). The *system dependence graph (SDG)*, which we introduce in Section 4.4, combines multiple PDGs into one graph.

Deriving the edges in the PDG that represent dependence requires knowledge of the order in which the statements in the program can be executed. The control flow graph captures this order and has the following definition:

Definition 4.1 (Control flow graph (CFG) [54]). *The* control flow graph *is a directed graph which has a node for each program statement and control predicate, as well as the special nodes Start and Stop. An edge from i to j means that control can directly flow from i to j. The Start node is the initial node and the Stop node is the node where the program terminates. An additional edge from Start to Stop is present.*

The edge from Start to Stop is not an edge that represents actual control flow, but has a special purpose which will be made clear at the time the SDG is defined.

Note that the start node will have no incoming edges and the stop node has no outgoing edges. In the CFG as presented in this chapter, each node corresponds to a statement except for the special Start and Stop nodes. When there is an edge from node i to j, then j is a *successor* of node i in the CFG and i is a *predecessor* of j. A node i that has more than one successor is called a *control predicate*, since the statement executed after i depends on the value of the predicate in the statement corresponding to i.

Figure 4.1 shows the CFG of `EGCD` (Program 4.1). Program `EGCD` is an implementation of the Extended Euclidian algorithm for finding the greatest common divisor. The nodes correspond to statements, but only the number of the statements are drawn in the nodes in order to improve the readability of the graph. The Extended Euclidian algorithm takes two integers `m` and `n`. The algorithm calculates the values of integer variables `x`, `y` and `z` in such a way that `x` is the greatest common divisor of `m` and `n` and $\texttt{my} + \texttt{nz} = \texttt{x}$.

In figure 4.1, node 7 is the only control predicate since it is the only node that has more than one successor, namely 8 and 20. After node 7 is executed, control proceeds to node 8 if `d` is not equal to zero or to node 20 if `d` is equal to zero. Node 19, which is the last node in the body of the `while` statement has an edge pointing back to node 7, since after each iteration of the loop the predicate is re-evaluated.

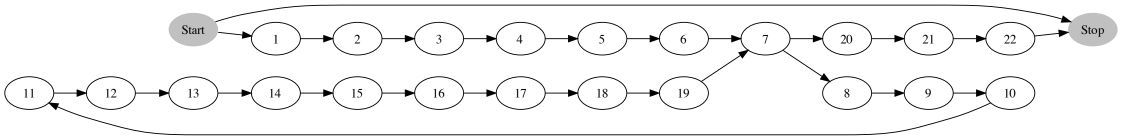


Figure 4.1: Control flow graph of `EGCD`

Now that we have defined the CFG, it is possible to define the *program dependence graph (PDG)*. At this point we limit ourself to programs that do not have concurrency, which means that there are only three types of dependence that must be taken into account, namely *data dependence*, *control dependence* and *divergence dependence*. Section 4.7 introduces three more types of dependence for concurrent programs. The definitions of the all six of these dependence types are borrowed from [33].

A node j is data dependent on node i if and only if node i stores a value in a variable x and node j uses variable x and the value of x is potentially not redefined between execution of i and j. Of course it is possible that x may or may not be redefined between execution of

Program 4.1 `EGCD` [15]

```
a1 = 0;
a2 = 1;
b1 = 1;
b2 = 0;
c = m;
d = n;
while (d != 0) {
    q = c / d;
    r = c % d;
    a2 = a2 - q * a1;
    b2 = b2 - q * b1;
    c = d;
    d = r;
    r = a1;
    a1 = a2;
    a2 = r;
    r = b1;
    b1 = b2;
    b2 = r; }
x = c;
y = a2;
z = b2;
```

i and j, depending on some control predicates executed between i and j. In this case j will be data dependent on node i, since node j can potentially use a value that i defines. The definition of data dependence is as follows:

Definition 4.2 (Data dependence)**.** *Node j is data dependent on node i if and only if there exists a variable x for which holds that $x \in DEF(i)$, $x \in USE(j)$ and there exists a path p from i to j such that for $x \notin DEF(n)$ for each CFG node n on path p excluding i and j. In this definition $DEF(i)$ is the set of variables defined at node i and USE(i) is the set of variables of which the value is used at node i.*

A node j is control dependent on node i if and only if the fact whether or not node j is executed depends on node i. This means that node i must be a node with at least two successors, thus it must be a control predicate. The definition of control dependence in terms of post-domination is as follows:

Definition 4.3 (Post-dominance)**.** *Node i is post-dominated by node j if $i \neq j$ and each path from i to Stop in the CFG contains j.*

Definition 4.4 (Control dependence)**.** *Node j is control dependent on node i if and only if $i \neq j$ and there exists a path p from i to j where each node n on path p excluding i and j is post-dominated by j, and i is not post-dominated by j.*

The edge in the control flow graph from Start to Stop makes sure that all statements in the body that are not nested within other statements (statements `(1)` through `(7)` and `(20)` through `(22)` in `EGCD`) are control dependent on the Start node, since the edge from Start to Stop essentially makes the Start node into a control predicate.

The last kind of dependence, divergence dependence, takes preservation of non-termination into account. A node is j is divergence dependent on node i if node i is a control predicate which is able to prevent node j from executing. The definition of divergence dependence, in terms of pre-divergence points, is as follows:

Definition 4.5 (Pre-divergence point)**.** *A node i is a pre-divergence point, if and only if i has multiple successors in the CFG, where one successor of i exits a loop (e.g. control predicate produced as part of a `while` or `for` statement) and another successor of i does not.*

The only pre-divergence point in `EGCD` is statement 7 (the `while` statement).

Definition 4.6 (Divergence dependence)**.** *Node j is divergence dependent on node i if and only if $i \neq j$ and i is a pre-divergence point and there exists a path p from i to j in the CFG such that no node on path p, excluding i and j, is a pre-divergence point.*

Figure 4.2 contains the PDG of program `EGCD`. All nodes that are control dependent on another node, are also divergence dependent on that node. For that reason, if a node is both control dependent and divergence dependent on that node, only the control dependence is shown in order to reduce clutter in the graph.

We can leave divergence dependence out, if we can be sure that the program terminates for every input given to it, or if we are not interested in non-terminating runs of the program. It is impossible to determine whether or not an arbitrary program halts. For simplicity, we assume here that each loop may be a non-terminating loop, even though it is possible to show for certain program that they either always terminate or never terminate.

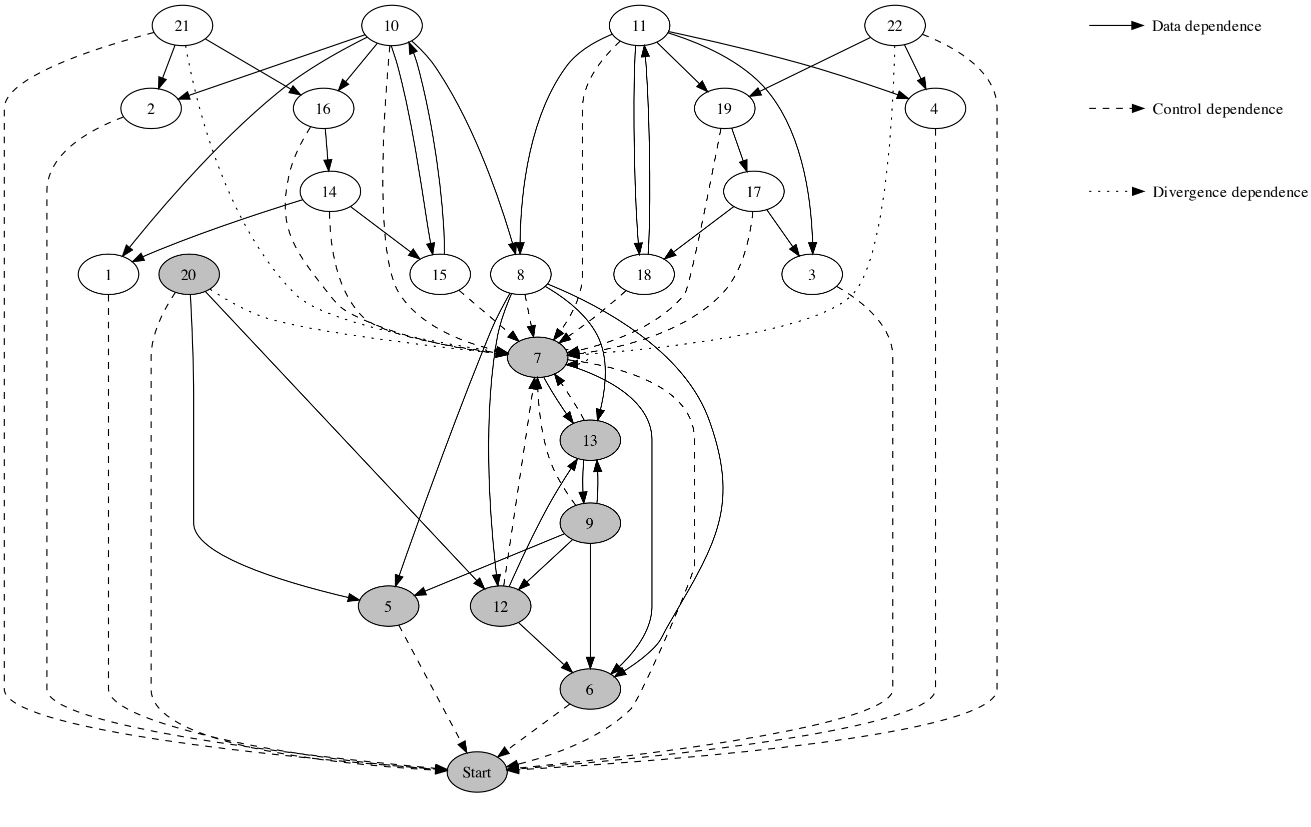


Figure 4.2: Program Dependence Graph of EGCD

These three types of dependence allow us to construct the PDG, where a backward slice with respect to slicing criterion $\langle p, \mathrm{USE}(p)\rangle$ is determined by starting at the node corresponding to statement p, and adding all nodes that are reachable from this node by following the edges, to the slice. The dependence graph based method of slicing only allows slicing criteria of the form $\langle p, \mathrm{USE}(p)\rangle$, which may seem like a limitation but in practice most slicing criteria are of this form.

The PDG also allows calculating forward slices for slicing criteria of the form $\langle p, \mathrm{DEF}(p)\rangle$. The forward slice consists of node p and all nodes that are reachable from p by following the edges *in reverse order.*

Since dependence graph based slicing only allows slicing criteria of the form $\langle p, \mathrm{USE}(p)\rangle$ for backward slicing and $\langle p, \mathrm{DEF}(p)\rangle$ for forward slicing, it is sufficient to specify only the statement p instead of the whole slicing criterion. Therefore, from now on only the statement of the slicing criterion is given instead of a slicing criterion of the form $\langle p, V\rangle$.

Example 4.7. *Program* `EGCD` *calculates three values* `x`*,* `y` *and* `z`*. Variable* `x` *contains the greatest common divisor of* `m` *and* `n` *after execution of* `EGCD`*. Now say we are only interested in calculating the greatest common divisor of* `m` *and* `n`*. The PDG of* `EGCD` *allows us to calculate a slice that only calculates* `x` *by backward slicing with respect to slicing criterion* $\langle 20, \{\mathtt{c}\}\rangle$*. Note that the slicing criterion contains variable* `c` *rather than* `x`*, since we calculate a backward slice and USE*$(20) = \{\mathtt{c}\}$*. In figure 4.2, the shaded are the nodes that are reachable from node 20, thus the nodes that are in this slice. Program 4.2 is the subprogram of 4.1 consisting of only the nodes in the slice. This program can be recognised as the Standard Euclidian algorithm for calculating the greatest common divisor of two integers.*

Program 4.2 The `EGCD` program sliced with respect to the slicing criterion 20 also known as the Standard Euclidian algorithm

```
(5)     c = m;
(6)     d = n;
(7)     while (d != 0) {
(9)         r = c % d;
(12)        c = d;
(13)        d = r; }
(20)    x = c;
```

4.3 Unstructured control flow

Section 4.2 demonstrates the construction of the PDG and the process of calculating slices using these PDGs. Unfortunately, using the PDG in this form leads to incorrect slices when a program contains unstructured control flow, in the form of jump statements like `break` and `continue` (Java has no `goto`). A `return` statement can also be a form of unstructured control flow. Another form of unstructured control flow exists in Java in the form of exceptions.

The problem with jump statements is as follows: a jump statement does not assign a value to any variable, which means no statement can be data dependent on it. A jump statement is not a control predicate, since it has only one successor, thus no statement can be control dependent on the jump statement. The consequence of this is that a jump statement will never be added to the slice, unless of course the jump statement is the slicing criterion.

Note that `return` statements in procedures that return a value (have a non-`void` return type) and statements that throw exceptions do count as unstructured control flow, but since they pass data, other statements can be data dependent on them, so they do not cause incorrect slices, even though they introduce unstructured control flow.

A solution to the problem of unstructured control flow (both the solution and problem were discovered by Ball and Horwitz [13] and Choi and Ferrante [21], independently of each other) is to treat a jump as if it were a control predicate where the predicate always evaluates to `true`. Treating the jump as such a control predicate results in an *augmented CFG (ACFG)* where the jump does not only have an edge to its target which it already has in the normal CFG, but it also has an edge to its *fall-through* node. The *fall-through* node is the node that would have been the jumps successor it were an empty statement. Calculation of control dependence using the ACFG rather than the normal CFG, causes the jump to be the dependee of other nodes in the PDG. Note that dependence types other than control dependence make use of the normal CFG rather than the ACFG.

4.4 Interprocedural slicing

In order to extend slicing to programs consisting of multiple procedures, the *system dependence graph (SDG)* has been introduced in [36] as an inter-procedural extension of the PDG. The system dependence graph consists of the PDGs of all procedures of the program, with extra edges to model the transfer of parameters and extra nodes to model all actual and formal parameters. The PDGs with these extra nodes and edges are *intra-procedural dependence graphs (IDG)*. In order to make clear how this works, we look at the SDG of program `Procedures` (repeated as Program 4.3 for ease of reference). Figure 4.3 contains two frames where each frame contains the statements of the IDG of one of the two procedures `func` and `main` of the program. The transfer of parameters is modeled by temporary variables. Variable `p_in` is the temporary variable associated to parameter `p` of procedure `func` and `func_ret` is the temporary variable associated to the return value of procedure `func`. When a call to `func` is made, the actual parameter is copied into `p_in` (node `p_in = x` and `p_in = y` in the SDG). In `func` the value of `p_in` is copied into the formal parameter `p`. Transferring the return value back to the calling procedure works in a similar way, via temporary variable `func_ret`. The edges that connect actual parameters and formal parameters are labeled `par-in` and `par-out` edges (abbreviations of `parameter-in` and `parameter-out`). A return value is an output parameter. Call edges connect the Start node of each method to the nodes that call them. All statements that are directly in the body of the method (thus not nested into another construct like `if` or `while`) are control dependent on the Start node.

However, calculating a backward slice with respect to slicing criterion `(14)` leads to a slice that includes statements `(8)` and `(11)` since `p_in` is has a `parameter-in` edge to both `p_in = x` and `p_in = y` (the shaded nodes in 4.3 are in the slice). The slice is correct, but contains statements that clearly do not need to be in the slice.

A possible solution is to make a copy of the IDG of `func` and connect the parameter-in, parameter-out and call edges of the two calls to `func` with different copies of the IDG of `func`. This solution, however, is not applicable when the program contains recursion, since then an infinite number of copies of the recurring procedure would be required.

Program 4.3 `Procedures`, repeated

```
public class Procedures {
    public static int func(int p)
    {
        return p + 1;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;

        int a = func(x);
        int b = func(y);

        System.out.println(b);
    }
}
```

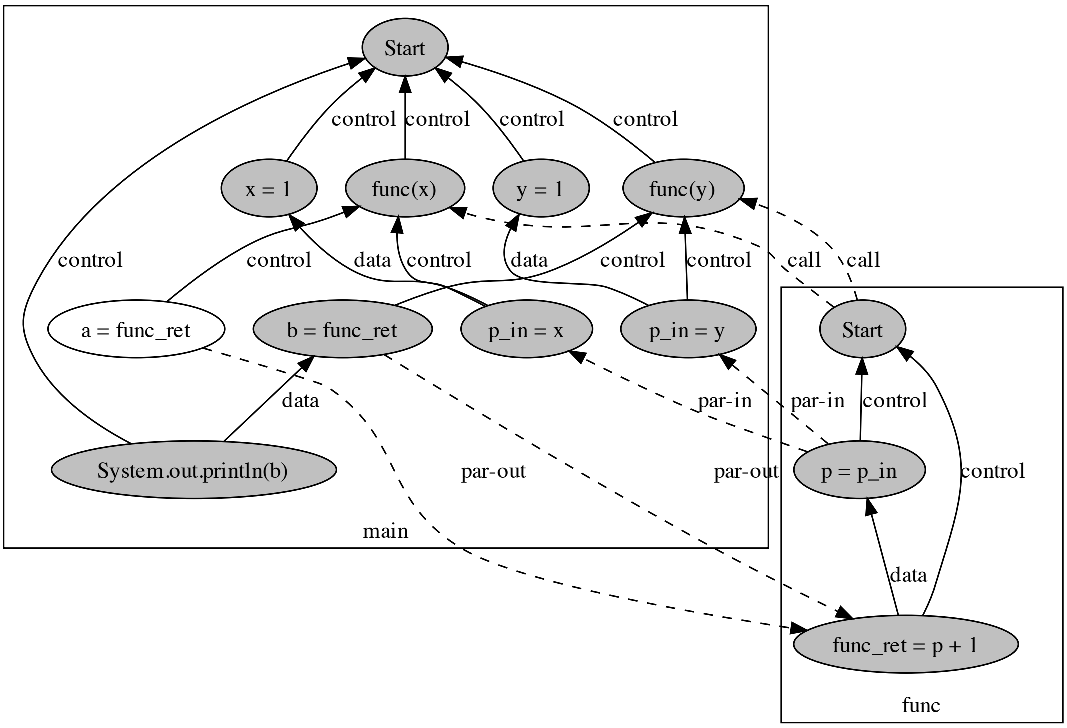


Figure 4.3: System dependence graph of `Procedures`

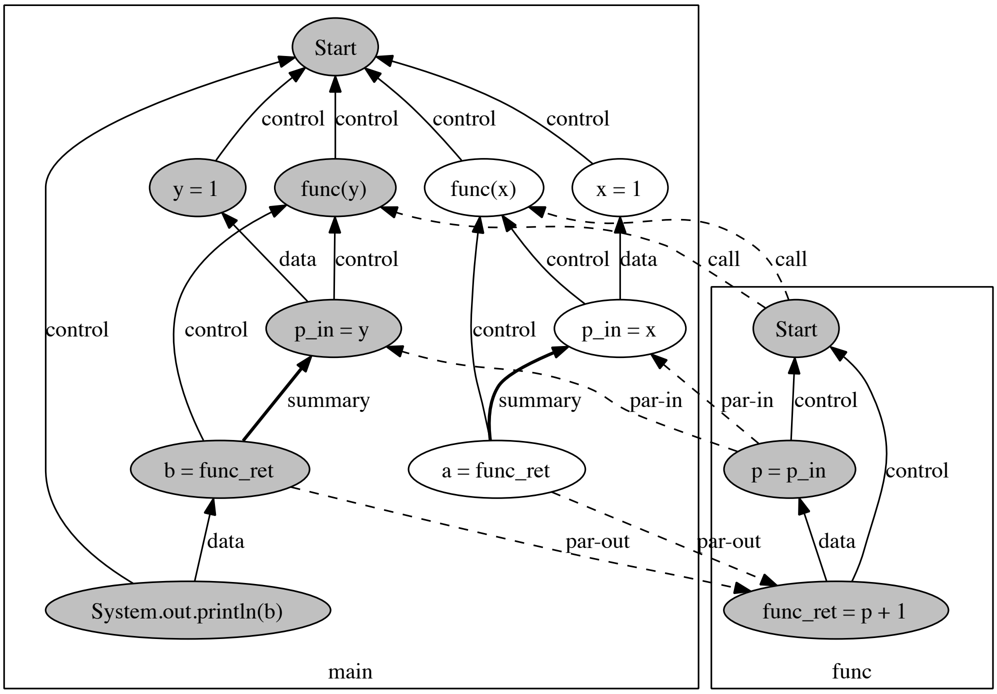


Figure 4.4: System dependence graph with summary edges of `Procedures`

Another algorithm which is capable of handling recursion is the summary two-phase slicing algorithm and is based on *summary edges* [36, 53]. Summary edges are edges that connect the nodes representing actual input parameters and actual output parameters when the value of the actual output parameter uses the actual input parameter. The SDG of program `Procedures` has a summary edge from the return value node to the actual parameter node for both of the calls (Figure 4.4). Slicing with summary edges consists of two passes. The first phase calculates a descending slice, which does not follow all `parameter-out` edges. The second phase calculates an ascending slice from the nodes that are reachable from `parameter-out` edges from the nodes reached during the first phase. The slice consists of all nodes that were reached in any of the phases. The slice with respect to slicing criterion `(14)` appears shaded in Figure 4.4 and does not contain the statements related to variable `x`, which should not be in the slice as explained above.

Summary edges are only capable of giving information on which output parameters depend on which input parameters. In the presence of dependence between nodes of different procedures (*inter-procedural dependence*), the summary two-phase slicing algorithm gives incorrect results. An example of this is given in Section 5.5.1, which provides a correctness validation of the slicing platform. At this point, none of the features we have discussed in this chapter create inter-procedural dependence, but Section 4.6 and Section 4.7 both do create such dependence. However, it is possible to get correct slices by making use of the iterated two-phase slicing algorithm [26] which is a slightly modified version of the summary two-phase slicing algorithm. Each time the iterated two-phase slicing algorithm follows a dependence edge that is inter-procedural in either the first or the second phase, the newly reached node is considered a new slicing criterion. This means that the iterated two-phase slicing algorithm keeps repeating the first and second phase until no new slicing criterion are added.

An extra challenge in slicing Java programs is *dynamic dispatch*. For program `Procedures` it is clear which method is called from `main`. However, when performing a method call on an object, the actual type of the object may have an influence on the method called. Note that the actual type of the object can not simply be obtained from the type of the variable in which the object resides, since the actual type may be a subclass of the type of the variable. *Points-to analysis* is capable of determining these types.

Points-to analysis is a static code analysis technique that establishes which pointers may point to which actual storage locations. In Java, the pointers are variables that have an object type, and the storage locations are object instances. Points-to analysis can be used to determine if two variables (potentially) refer to the same object during runtime, thus giving an approximation of which variables can be aliased during runtime. Another application of points-to analysis is providing information about which `new`-operator can be the creating operator of an object. An efficient implementation of points-to analysis for Java exists in the form of SPARK [42] which is used in combination with the Java program optimization framework Soot [10].

If the called method is a non-static method, the method has access to the object it is called on in the form of the `this` pointer. Therefore, non-static methods have another input parameter which contains the `this` pointer.

4.5 Exceptions

As mentioned in section 4.3, exceptions introduce unstructured control flow. A `throw` statement can cause control to flow to the Stop node of a procedure, or to a `catch` clause. Fortunately, exceptions involve data in the form of an object of type `Exception` or an object derived from it. This allows extending data dependence for exceptions.

When a `throw` statement occurs, and the exception is caught by a `catch` clause, the statements in the `catch` clause are data dependent on the `throw` statement. If the exception is not caught, then control proceeds to the the end of the procedure. The `throw` is treated as a special `return` statement which returns the thrown exception. The returned exception is treated as an output parameter, just like a return value. In this way, the exception is propagated to the procedure calling the procedure that can throw an uncaught exception. A method call to a procedure that is capable of throwing exceptions it does not catch, is treated as a `throw` statement itself.

4.6 Composite data types and aliasing

In Java, two composite data types are present: arrays and objects. A simple approach for arrays is to treat the whole array as a single scalar. It should be noted that for arrays, a statement that defines the value of an array element depends on earlier statements that also define the value of an array element, since such a statement only modifies one element and leaves the other elements unchanged.

A method to handle objects properly was introduced in [30]. Their method introduces data dependencies between nodes that define and use the value of a certain field (instance variable). The definition is as follows:

Definition 4.8 (Data dependence for object fields:). *SDG node n_2 has a data dependency on SDG node n_1 if and only if n_1 defines field* `f` *of variable x, n_2 uses field* `f` *of variable y, x and y are potential aliases (determined by points-to analysis) and control flow can reach n_2 after n_1 without intervening definition of field* `f`*. Field* `f` *occurs multiple times in the definition. In all occurences, this must be the same field in terms of the fields name and defining class.*

Since these types of data dependence are capable of producing dependence between nodes of different procedures, the summary two-phase slicing algorithm can not be used in combination with this method, but the iterated two phase slicing algorithm must be used.

4.7 Concurrency

Slicing programs with concurrency adds three extra dependence types [33] to the already existing three that are part of the PDG and IDG. The first concurrency dependency is *interference dependence*. Interference dependence essentially models data dependencies between threads and has the following definition:

Definition 4.9 (Interference dependence). *A node is n interference dependent on another node m, if both nodes are run from different threads and m modifies a variable of which the value can be used by node n.*

Since interference dependence produces dependence between nodes of different procedures, the slicing algorithm with summary edges introduced in 4.4 can not be used in combination with interference dependence. However the iterated two-phase slicing algorithm can be used, even though it produces imprecise slices since interference dependence is not *transitive* [26]. A type of dependence is *transitive* if it is the case that if node a depends of node b and node b depends on node c then node a depends on node c. Interference dependence is the only type of dependence discussed in this chapter that is not transitive. Figure 4.5 gives an example of a case where interference dependence is obviously not transitive. For clarity, the figure only includes data and interferences dependence edges. The frames consist of nodes that are in the same thread and both threads execute concurrently, where the execution order within both threads is from top to bottom. The slicing criterion is the statement `b = a - 4`. Slicing with the iterated two-phase slicing algorithm leads to a slice consisting of all six nodes. However, `x = b / 2` can not have an influence on the slicing criterion `b = a - 4` since it is executed after the statement of the slicing criterion, therefore the slice in imprecise (Krinke calls this effect *time travel*). Krinke [41] and Nanda [46] both introduced two different slicing algorithms that are capable of calculating precise slices (slices that are capable of pruning time travel) of concurrent programs. These two algorithms were compared and improved by Giffhorn and Hammer in [26]. Their conclusion was that both algorithms produce correct and precise slices, where Nanda's algorithm performs best speedwise.

The second concurrency dependency is *synchronization dependence*, which makes sure that `synchronized` statements are included when needed to preserve correct behaviour of the slice.

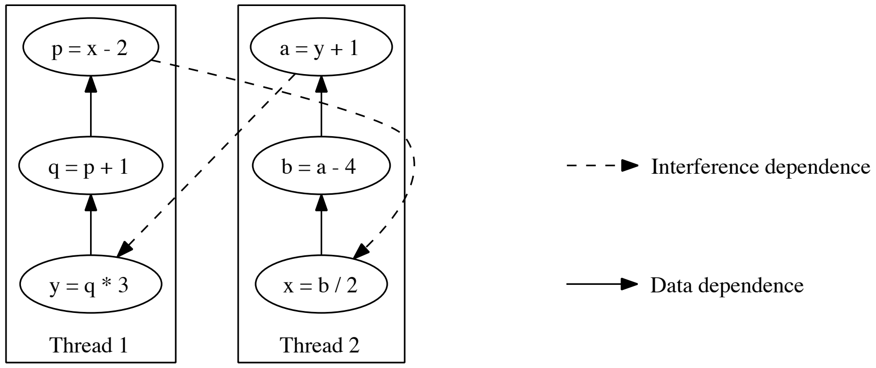


Figure 4.5: System dependence graph demonstrating non-transitivity of interference dependence

Definition 4.10 (Synchronization dependence)**.** *If some node n which resides inside of a critical region (a synchronized region in Java) modifies a variable which can be shared with other threads, then the synchronization construct which lies around n should appear in the slice.*

The third and last concurrency dependency is *ready dependence.* Ready dependence models the dependence of a `wait` statement on a `notify`.

Definition 4.11 (Ready dependence)**.** *A node n is ready dependent on node m if m's failure to complete (either because m is never reached, or there is a* `wait` *for which a* `notify` *never occurs) would make the thread of n block before n is reached.*

Chapter 5

Syntactically correct slices of Java 6 programs

In Chapter 4, static slicing algorithms are discussed which are able to determine which statements can be deleted from the program being sliced. However, the goal is to produce a slice in the form of source code which is syntactically valid and succesfully compiles. Only including statements from the original program is insufficient since essential parts would be missing, like declarations of local variables, fields, methods and classes. However, the slicing algorithms only consider statements.

The slicing platform makes use of Soot, a Java optimization framework [10], that translates Java source code into an intermediate representation. This chapter demonstrates in which way system dependence graphs (SDGs) are built from the intermediate representation provided by Soot in such a way that a syntactically valid slice can be derived.

Another issue is that simply building an SDG of a whole Java program, including the Java Runtime Environment classes it uses, may result in a very large graph, even for small programs since small program can import lots of functionality from the Java Runtime Environment. This chapter introduces some techniques to overcome this problem.

5.1 Intermediate representation

The slicing platform makes use of Soot, a Java optimization framework [10]. As the name indicates, Soot is a framework built for optimizing Java programs. Soot works by reading Java source code or Java class files and convert them to one of the four intermediate representation forms [24] that are part of Soot. The code in intermediate form can now be optimized by Soot, and after that written out as Java class files.

In Soot and also in the slicing platform, a distinction is made between two types of classes: *application classes* and *library classes*. The *application classes* are classes that make up the program that is being sliced. All application classes must be available in the form of Java source files. The *library classes* are classes that are used by the program, however the calculated slice does not include information about which statements of the library classes are in the slice. Library classes are read in the form of Java class files.

The slicing platform makes use of the capability of Soot to read Java source code and Java classes and convert them to Jimple [55], one of the four intermediate representation forms provided by Soot. Note that this conversion includes desigaring as described in Section

3.2. The Jimple representation of method `main` of program `C` (Program 5.1) is shown as Program 5.2. The lines that start with the letter `J` followed by a number are the actual Jimple statements.

Program 5.1 Source code of `C`

```
(1)     public class C {
(2)         public static int func(int p)
(3)         {
(4)             return p + 1;
(5)         }
(6)
(7)         public static void main(String[] args) {
(8)             int x = 1;
(9)             int y = 2;
(10)
(11)            int a = func(x + y + 3);
(12)            int b = func(y);
(13)
(14)            if (b > 2) {
(15)                b = 0;
(16)            } else {
(17)                b = 4;
(18)            }
(19)
(20)            int c = b;
(21)        }
(22)    }
```

Jimple statements are very low-level statements, as Program 5.2 shows. Arithmetic instructions such as additions add two values and store them in a variable. Adding more than two values requires the addition to be split up into multiple statements. For an example of this, see line 11 of `C` and Jimple statement `J3-J5` which calculates expression `x + y + 3`. First, the value of `x + y` is stored in a temporary variable and after that the value `3` is added. Jimple has no direct support for constructs such as `if` and `while` but instead Jimple uses conditional and unconditional gotos (i.e. `J10`, `J11` and `J15`) in order to express such constructs. Jimple statement `J0` loads the value of the formal parameter of this method into local variable `args`, which is a variable named after the parameter.

5.2 Calculating syntactically correct slices

The previous section gave an introduction to the intermediate representation Jimple. This section shows in which way Java code in Jimple form is used to derive slices in the form of syntacticaly valid programs.

Program 5.2 Jimple code of method `main` of `C`

```
Text range (7:2)-(21:2)
Method signature: <C: void main(java.lang.String[])>

No text range
J0    args := @parameter0: java.lang.String[]

Text range (8:3)-(8:12)
J1    x = 1

Text range (9:3)-(9:12)
J2    y = 2

Text range (11:3)-(11:26)
J3    temp$0 = x
J4    temp$1 = temp$0 + y
J5    temp$2 = temp$1 + 3
J6    temp$3 = staticinvoke <C: int func(int)>(temp$2)
J7    a = temp$3

Text range (12:3)-(12:18)
J8    temp$4 = staticinvoke <C: int func(int)>(y)
J9    b = temp$4

Text range (14:3)-(18:3)
J10   if b > 2 goto J13
J11   goto J17
J12   nop

Text range (15:4)-(15:9)
J13   temp$5 = 0
J14   b = temp$5
J15   goto J19
J16   nop

Text range (17:4)-(17:9)
J17   temp$6 = 4
J18   b = temp$6
J19   nop

Text range (20:3)-(20:12)
J20   c = b

No text range
J21   return
```

5.2.1 Mapping system dependence graph nodes to source text ranges

Looking at Program 5.2, we can see that Jimple statements have source code text ranges associated with them. Note that these text ranges are produced by a modified version of Soot 2.4.0 (these modifications were performed as part of this project). This is the case, since the text ranges originally provided by Soot 2.4.0 are not precise enough. The Jimple statements corresponding to the same statement in the source code are grouped. Each of these groups is accompanied by the text range in the original source that they correspond to. For example, Jimple statement `J1` corresponds to line 8 in the original source. The text range shown in `(8:3)-(8:12)` which means that the statement ranges from line 8, column 3 through line 8, column 12. The text ranges include column positions, since a line can have multiple statements and one statement can span mutiple lines.

Figure 5.1 contains all ranges in the method in the form of a tree. The node on the top represents the entire method and its text range from line 7 through 17. The children of this top node represent all statements that are directly in the method. The `if` statement ranging from line 14 through 18 has two children, namely the two statements inside of the `if` statement.

Each Jimple statement corresponds to one node in the SDG. An exception to this rule is a statement that performs a method call (`J6` and `J8` in Program 5.2). For statement that perform a method call, in addition to a node for the call itself there is a node for each actual parameter and the output value (as shown in Section 4.4). Since each node in the dependence graph has a Jimple statement associated with it, each node in the SDG also has a text range associated with it. After calculating the slice, these text ranges are used to determine which parts of the source text are in the slice. When determining which text ranges are in the slice, it is imporant to look at the text range tree (5.1). As can be seen there, the `if` statement spans from line 14 through 18. Taking this text range from the source file would result in the text `if (b > 2) { b = 0; } else { b = 4; }`, excluding whitespace. Thus this also includes the statements within the `if`, which should not be actually part of the `if`. Therefore it is necessarry to subtract the text ranges corresponding to the children of the `if` node in the text range tree. This will result in the text `if (b > 2) { } else { }`, (again excluding whitespace) thus the `if` without the statements within it. The same needs to be done for the entire method. The text range of the entire method without the statements within it results in the text `public static void main(String[] args) { }`. The text range of the entire method is associated with the `Start` node in the SDG, so that the text declaring the method is included whenever a statement from the method is included in the slice.

The SDG of program `C` making use of Jimple statements is shown in Figure 5.2, including a backward slice with respect to a slicing criterion consisting of statement `int c = b;` on line 20. The shaded nodes are in the slice, and the darker shaded node is the slicing criterion. Extracting the text ranges corresponding to the nodes in the slice leads to Program 5.3. However, this program is not yet syntactically correct since it misses the class declaration. Another deficiency is the fact that one statement in the original program can correspond to multiple Jimple statements and thus also multiple SDG nodes. If only some of the nodes correspond to one statement in the source, but not all of them, then it is unclear whether or not the statement is included or not. A solution for these problems is given below.

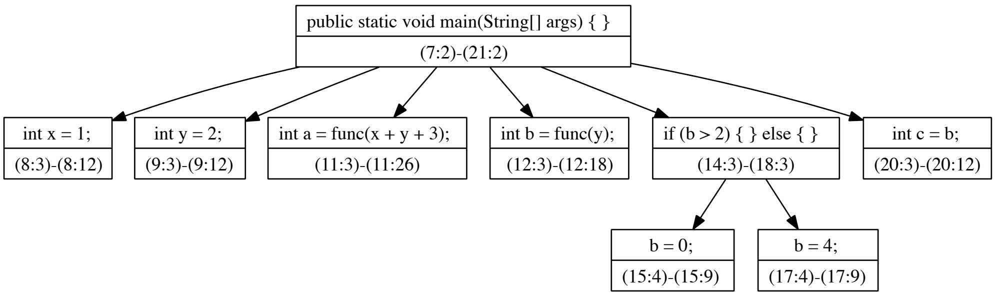


Figure 5.1: Text range tree of method `main` of `C`

Program 5.3 Backward slice of `C` with respect to slicing criterion consisting of statement `int c = b`, on line 20

```
public static int func(int p)
{
    return p + 1;
}

public static void main(String[] args) {

    int y = 2;

    int b = func(y);

    if (b > 2) {
        b = 0;
    } else {
        b = 4;
    }

    int c = b;
}
```

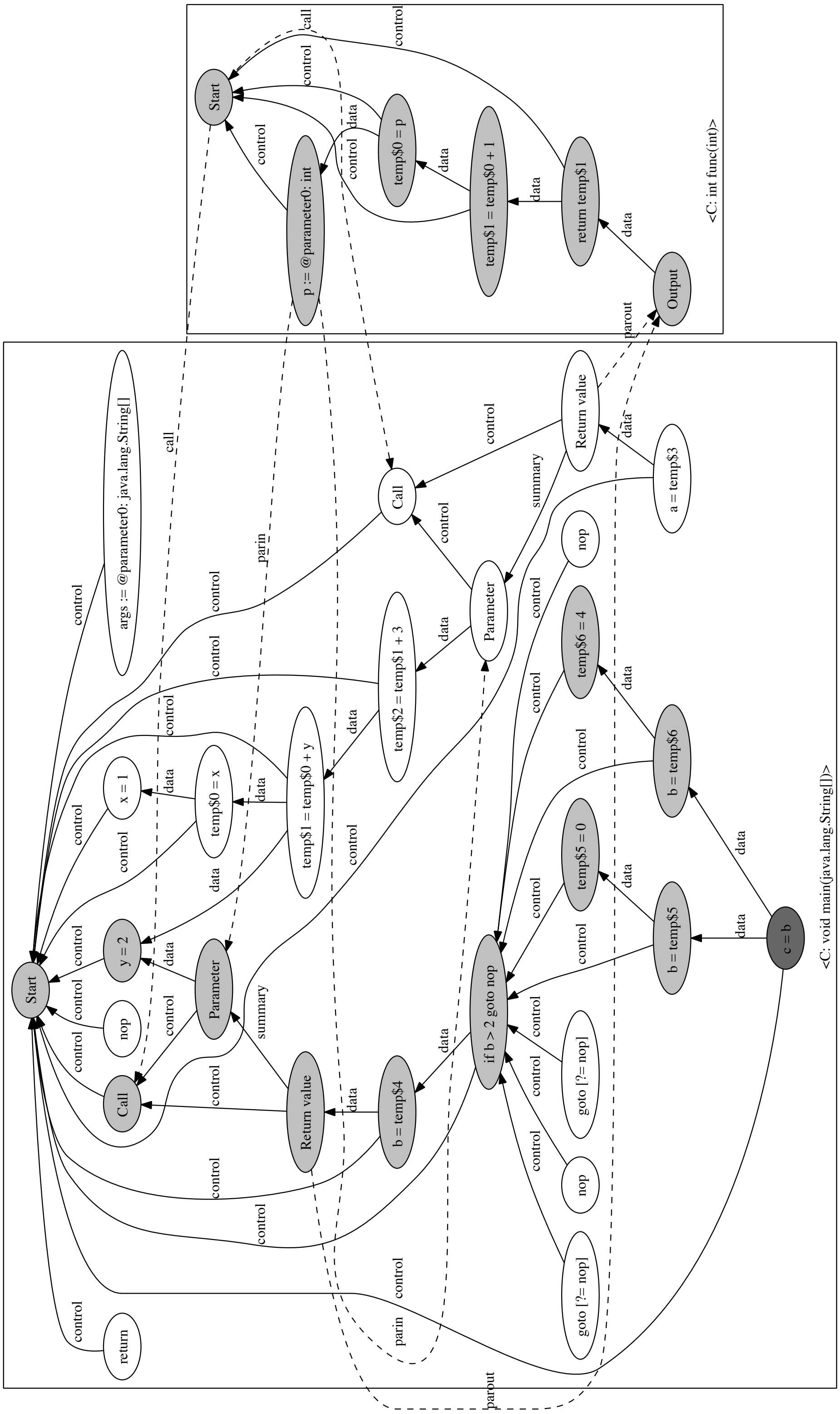


Figure 5.2: System dependence graph of C

5.2.2 Structure dependence

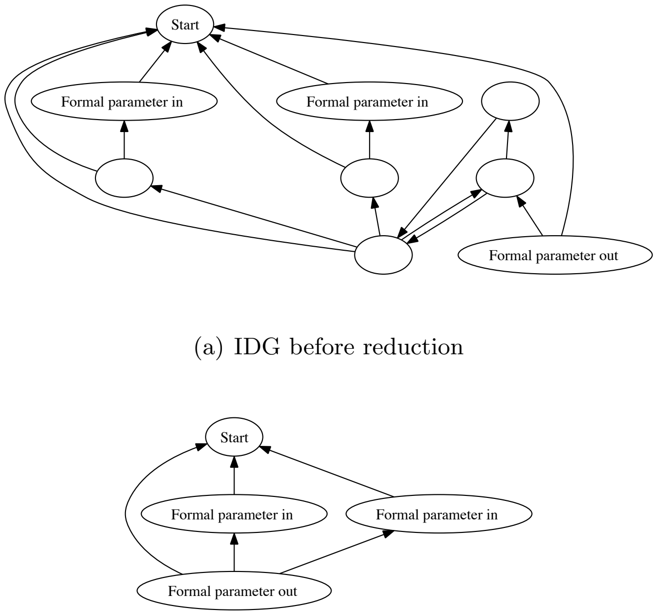
This section introduces a new type of dependence, in order to resolve the problem with missing constructs in the sliced output in source form and the problem that multiple Jimple statements may correspond to the same statement in the source text. The introduction of this type of dependence is part of the contribution of this thesis. This new type of dependence is called *structure dependence* and links SDG nodes to other SDG nodes as required to provide syntactically correct programs. This requires extra SDG nodes to be introduced for constructs other than statements and method declarations. There are extra nodes for: class declarations, interface declarations, field declarations, abstract method declarations and interface method declarations. In the following case nodes are structure dependent on other nodes:

- SDG nodes that are mapped to the same source text range, are structure dependent on each other.
- A method declaration is structure dependent on the class it is declared in.
- A statement that uses or defines the value of a local variable is structure dependent on the declaration of the local variable.
- A statement that uses or defines the value of a field is structure dependent on the field declaration.
- A class declaration is structure dependent on the class declaration of its super class.
- A class declaration is structure dependent on the interface declarations of the interfaces it implements.
- A method declaration of a method that has a non-`void` return type is structure dependent on all return statements of the method. This is required since the Java language requires that each execution path if such a method returns a value.
- A method call on an object (through dynamic dispatch) residing in a variable of type `A` is structure dependent on the method declaration of the method with the same signature in `A` or a superclass of `A` if `A` does not have the method itself.
- An abstract method declaration or a method declaration in an interface is structure dependent the corresponding method declaration in each class implementing the interface or extending the abstract class.

An important property of structure dependence is that its direction does not reverse when calculating forward slices rather than backward slices.

5.3 Dealing with large system dependence graphs

Constructing a full system dependence graph, including nodes for classes in the Java Runtime Environment may lead to a very large graph, even for small programs since programs can use a lot of functionality from the Java Runtime Environment. This section discusses two ways to deal with this problem. The first one is using techniques to reduce the size of the SDG, and the second one is not storing the entire SDG but generating it on the fly.



(a) IDG before reduction

(b) IDG after reduction

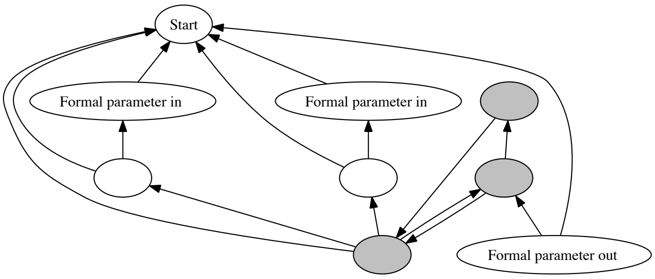
Figure 5.3: Two IDGs demonstrating reduction of methods from library classes

5.3.1 Reducing the system dependence graph

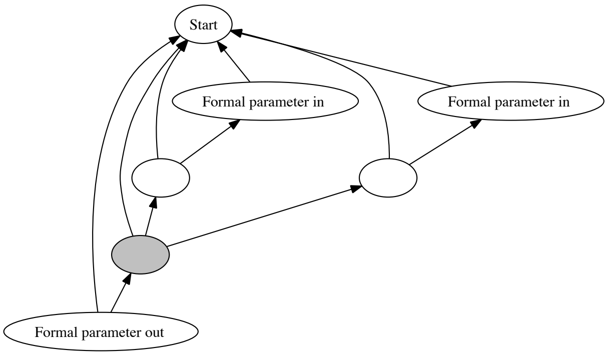
This section introduces two types of reductions that can be employed to reduce the size of the SDG. Both techniques are performed on the inter-procedural dependence graphs (IDGs), rather than the entire SDG. The advantage of this is that each IDG can be calculated individually and be reduced before adding it to the SDG.

The first reduction technique exploits the fact that in library classes, it is not required to know which nodes are exactly in the slice. Thus only the relation between the input and output parameters is relevant, making it possible to remove all nodes other than the parameter in and parameter out edges. An example of this reduction is shown in Figure 5.3. The unlabeled nodes in Figure 5.3(a) are the nodes that can be removed. The result of the reduction is shown in Figure 5.3(b), which contains dependence edges in such a way that nodes that were reachable from each other before the reduction are still reachable from each other after the reduction, assuming the two nodes are two nodes that still exist after the reduction.

The second reduction technique is based on the notion of *strongly connected components.* A *strongly connected component* is a subgraph of the IDG for which holds that for each pair of IDG nodes a and b in the subgraph, there is a path in the subgraph from a to b. This must hold for both directions, thus if there is a path in the subgraph from a to b then there must also be a path from b to a. All nodes in a strongly connected component are dependent on each other, thus if one IDG node of the strongly connected component is in the slice then each IDG node is in the slice. Therefore, all nodes of the strongly connected component can



(a) IDG with a strongly connected component (the strongly connected component is depicted shaded)



(b) IDG with the strongly connected component merged into one node (the node resulting from the merge)

Figure 5.4: Two IDGs demonstrating merging of strongly connected components

be merged into one node, as shown in Figure 5.4.

The reduction techniques were shown on IDGs, but in case a context-insensitive slicing algorithm is used, these reduction techniques can also be used on the entire SDG. It must however be taken into account that some of the slicing algorithms (the two-phase slicing algorithms and limited context slicing algorithm) treat `parameter-in`, `parameter-out` and `call` edges in a special way so nodes connected with these edges must not be merged.

5.3.2 On the fly creation of the system dependence graph

An alternative approach to creating the full SDG and using reductions, is to not create the entire SDG, but only those parts that are actually visited during slicing. Slicing with SDGs is essentially a graph reachability problem, which means that only for the nodes that are in the slice it is necessarry to know the dependences.

During slicing, the slicer keeps track of two sets of SDG nodes. The first set is called the front. Initially, the front contains only the node or nodes corresponding to the slicing criterion. The second set is the retired set which is initially empty. When running the algorithm, it will repeatedly pick a node from the front set, and process this node. This will be repeated until the front set is empty. Processing a node consists of determining the nodes on which it depends (for backward slicing) or the nodes which are dependent on it (for forward slicing). Each of these nodes that is not already in either the front or the retired set, will be added to the front set and the node being processed is moved to the retired set. The algorithm terminates since the number of nodes that can be reached, which is a subset of all the nodes that would be in a full SDG, is finite and no node can be processed more than once, since after processing a node it will be moved to the retired set which means it can not be added to the front set again.

When the algorithm terminates, the retired set contains all nodes that are in the slice. Of these nodes, only those nodes that have corresponding text ranges are kept in the dependence graph.

Calculation of the dependencies requires various code analysis techniques. Points-to analysis is global and calculated before slicing is started. Determining the IDG of a method is local, which means it only applies to a single method and is not influenced by any other methods. Since these IDGs take up memory space, and they might not be needed after some time it is beneficial to only store those IDGs into memory that are still needed. These IDGs will be stored in a cache which only stores at most n IDGs where n is a user configurable parameter. The heuristic which is used to pick a node prefers to pick a node for which the program dependency graph is still in the cache. When the IDG for a new method is needed and there cache already contains n IDGs, then another IDG is removed from the cache. This can either be done at random, or by removing the IDG from the cache that has been in the cache for the longest amount of time.

5.3.3 Conclusion on dealing with large system dependence graphs

The slicing platform implements both of the reduction techniques, but does create the full SDG rather than creating it on the fly. The disadvantage of generating the SDG on the fly is that information needs to be recalculated for each slice. Since it is expected that more than one slice is calculated on the same program, the approach of generating the full SDG is chosen in favor of calculating it on the fly.

5.4 Slicing platform design

This section shows the design of the slicing platform I made as a part of this project. It gives an overview of the slicing platform and shows in which way the slicing platform can be configured and extended.

5.4.1 Overview

Figure 5.5 provides an overview of the components of the slicing platform. The square nodes are processes and the round nodes are data. The nodes corresponding to user input data are depicted shaded. The input consists of the Java source code (the application classes) and Java class files (the library classes). The other user input data consists of the configuration (see Section 5.4.3) and the slicing criterion. The frame labelled contains the components that are part of the existing frameworks Soot and JastAddJ. The components outside of this frame are part of my slicing platform. The version of Soot used here is a modified version of Soot 2.4.0. These modifications consists of improvements of the precision of the text ranges added

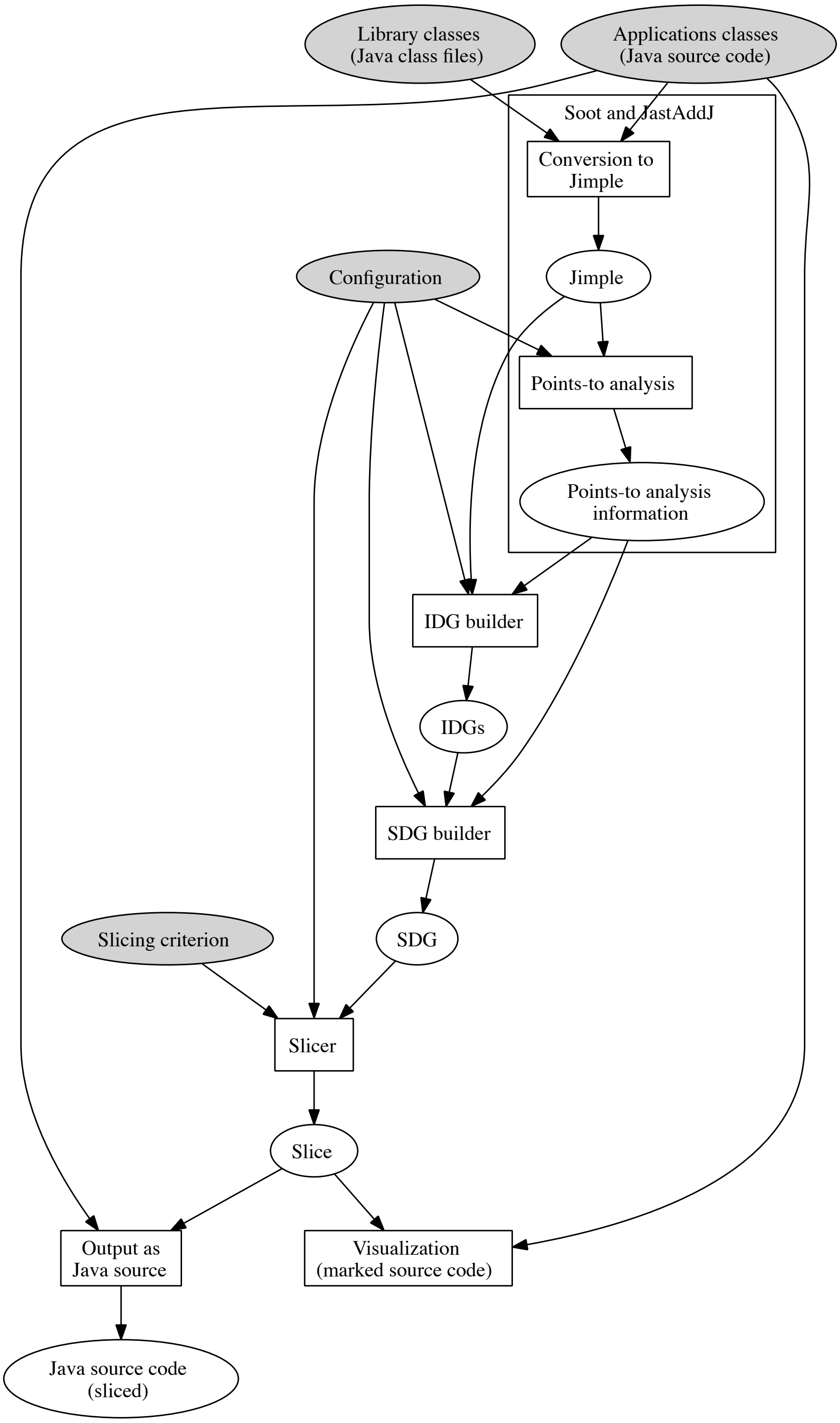


Figure 5.5: Slicer platform overview flow chart

to the intermediate representations by Soot. Soot is free software and licensed under the GNU Lesser General Public License, and JastAddJ is licensed under the BSD license, thus both are open source.

The first step of the slicing platform is the conversion of both the applications and library classes into Jimple by Soot. The application classes are in Java source code form and the library classes are in Java class file form. The Jimple code that results from conversion of the application classes is augmented with information about the corresponding text ranges, as shown earlier in this chapter.

The next step is performing the points-to analysis. The slicing platform contains a component that uses SPARK [42] (which is part of Soot) to perform points-to-analysis. The slicing platform allows plugging in other points-to-analysis algorithms. A flag in the configuration determines the points-to-analysis method that is used by the slicing platform. Section 5.4.3 provides an overview of all configuration flags.

The IDG builder uses both the Jimple code and points-to-analysis information to build the IDGs. The IDG builder only builds IDGs for methods that are reachable from the main method of the program. In order to determine which methods are reachable, the points-to-analysis information is needed. The points-to-analysis provides information about the types of actual objects in variables, giving information about which method call transfers control to which method. The slicing platform derives the set of reachable methods from this information.

The IDG builder creates one IDG for each reachable method in the program. The IDG builder creates one node for each Jimple statement and also the other nodes that IDGs require (start, formal parameters and actual parameters). After creating the nodes, the IDG builder inserts intra-procedural dependence edges between the nodes. The types of dependence for which edges are inserted can be controlled by flags in the configuration. After adding dependence edges, the slicing platform allows transformation to be executed on the IDG, such as reduction and merging of strongly connected components (Section 5.3.1). Which transformations the slicing platform performs is determined by the configuration. It is also possible to postpone performing these transformations on the SDG. However, these transformations reduce the size of the dependence graph and doing this in an earlier stage means that less nodes have to be in the memory of the computer at the same time, thus reducing the amount of memory that is required to build the SDG.

The SDG builder combines the IDGs for all the reachable methods into one dependence graph, the SDG. As with the IDG, there is a step that consists of adding dependence edges and a step that consists of performing transformations. The dependence edges represent inter-procedural dependence, such as dependence between calling methods, and field references. The transformations are again reduction and strongly connected component merging, this time on the entire graph. Again, the transformations the slicing platforms performs is determined by the configuration.

The slicer is the component that takes the SDG and the slicing criterion and produces the slice, where the slice is represented by the subset of SDG nodes that are in the slice. Multiple slicing algorithms are available, and the configuration determines which one of these is used. It is also possible to calculate multiple slices using the same SDG. There are two components that provide ways to show the slice to the user. The first one output the slice in the form of Java source and the second one shows the slice by marking the text that is in the slice.

5.4.2 Components

The slicing platform consists of components that are based on interfaces and abstract classes that allow for other types of dependence, transformations and slicing algorithms to be added. The configuration determines which actual implementations are used, see section 5.4.3 below. In section 5.4.3, the names of the implementations of the components are also given. The table below shows the interfaces and classes that components extending the slicing platform need to implement or extend:

Interface/Class	**Description**
`IPointsToAnalysis`	Interface for implemetations of points-to analysis. This interface and its implementations reside in package `javasdgslicer.pointstoanalysis`.
`IIDGDependenceGenerator`	Interface for an implementation that produces intra-procedural dependence edges for IDGs. This interface and its implementations reside in package `javasdgslicer.idg.dependence`.
`IIDGTransformation`	Interface for an implementation that provides a transformation on IDGs. This interface and its implementation reside in package `javasdgslicer.idg.transform`.
`ISDGDependenceGenerator`	Interface for an implementation that produces inter-procedural dependence edges for SDGs. This interface and its implementations reside in package `javasdgslicer.sdg.dependence`.
`ISDGTransformation`	Interface for an implementation that provides a transformation on SDGs. This interface and its implementations reside in package `javasdgslicer.sdg.transform`.
`Slicer`	Abstract class that needs to be extended in order to provide slicing algorithms. This class and its implementations reside in package `javasdgslicer.slicer`.
`Output`	Abstract class that needs to be extended in order to provide alternative methods to output the slice. The slice is calculated in the form of a set of SDG nodes, and the output can be in any format that is implemented here, as a subclass of `Output`. An example is the class `JavaSourceDirectoryOutput` which writes the slice in the form of Java source code. This class and its implementations reside in package `javasdgslicer.slicer.output`.

Note the two differences between dependence generators and transformations. The first difference is that dependence generators are only allowed to add dependence edges between nodes, while transformations are also able to remove dependence and add, remove or merge nodes. The slicing platform always runs the dependence generators before the transformations for the IDGs and the SDG.

5.4.3 Configuration

The overview (Section 5.4.1) mentions the fact that certain aspects of the SDG building process and the slicing process are influenced by the configuration. The slicing platform loads the configuration from a text file, such as the example shown in Text file 5.1. The `//` denotes that the rest of the line is a comment. Configuration entries are of the form `key = value`. The table below gives the meaning of each of the keys that correspond to components present in the implementation of the slicing platform. The value is a boolean (`true` or `false`) for most of the keys, but can also be an identifier or a number. The table below contains for each key the type of value expected for that specific key.

Components that extend one of the interfaces or the abstract classes from 5.4.2 need to define new keys, and add these to the class that keeps track of the configuration (class `Configuration` in package `javasdgslicer`).

Text file 5.1 Example configuration file

```
// Points-to analysis
points-to-analysis = spark

// IDG dependence
idg-data-dependence = true
idg-control-dependence = true
idg-divergence-dependence = false
idg-structure-dependence = true

// IDG transformations
idg-text-ranges = true
idg-reduction = true
idg-strongly-connected-components = true

// SDG dependence
sdg-call-dependence = true
sdg-data-dependence = true
sdg-structure-dependence = true
sdg-summary-dependence = false

// SDG transformations
sdg-reduction = false
sdg-strongly-connected-components = false

// Slicer
slicer=iterated-two-phase
slice-direction=backward
```

Key	**Interpretation**
`points-to-analysis`	Determines the type of points-to analysis used. The only type of points-to analysis implemented is SPARK, which is denoted by the value `spark`. The points-to analysis is provided by class `PointsToAnalysisSPARK` which calls SPARK in the Soot framework to perform the points-to analysis.
`idg-data-dependence`	The value is a boolean. If this value is set to `true`, data dependence edges are added to the IDGs for scalar variables. The implementation resides in class `IDGDataDependence`.
`idg-control-dependence`	The value is a boolean. If this value is set to `true`, control dependence edges are added to the IDGs. The implementation resides in class `IDGControlDependence`.
`idg-divergence-dependence`	The value is a boolean. If this value is set to `true`, divergence dependence edges are added to the IDGs. The implementation resides in class `IDGDivergenceDependence`.
`idg-structure-dependence`	The value is a boolean. If this value is set to `true`, structure dependence edges are added to the IDGs. This only includes structure dependence that is local to a single procedure. There are two cases of structure dependence that are intraprocedural: the case where a method declaration of a method that has a non-`void` return type is structure dependent on all return statements of the method and the case where the occurence of a local variable depends on the declaration of that local variable. The implementation resides in class `IDGStructureDependence`.
`idg-text-ranges`	The value is boolean. If set to `true`, text ranges are added to the IDG nodes of application classes. Also, structure dependence between IDG nodes correponding to the same text range are added. The implementation resides in class `IDGTextRanges`.
`idg-reduction`	The value is boolean. If set to `true`, reduction is applied to each IDG that belongs to a library class. The implementation resides in class `IDGReduction`.
`idg-stronglyconnectedcomponents`	The value is boolean. If set to `true`, nodes in the IDG that belong to the same strongly connected component are merged. The implementation resides in class `IDGStronglyConnectedComponents`.
`sdg-call-dependence`	The value is a boolean. When `true`, dependence edges between method calling each other are added to the SDG (`parameter-in`, `parameter-out` and `call` edges). The implementation resides in class `SDGCallDependence`.
`sdg-data-dependence`	The value is a boolean. When `true`, data dependence between nodes using the value of a field or array element and nodes setting the value of a field or array element are added. The implementation resides in class `SDGDataDependence`.

`sdg-summary-dependence`	The value is a boolean. When `true`, summary edges are added. The edges are required for the summary two-phase and iterated two-phase slicing algorithms. If this type of dependence is disabled for these algorithms, then statements that calculate values used as parameters will be missing from the slice. The implementation resides in class `SDGSummaryDependence`.
`sdg-structure-dependence`	The value is a boolean. When `true`, all inter-procedural structure edges are added. The implementation resides in class `SDGStructureDependence`.
`sdg-reduction`	Similar to `idg-reduction`, setting this to `true` enables reduction on the entire SDG. The only nodes that remain in the final SDG are the nodes that have one or more text ranges associated to them. This can only be used in combination with the context-insensitive slicing algorithm. The implementation resides in class `SDGReduction`.
`sdg-strongly-connected-components`	Similar to `idg-stronglyconnectedcomponents` setting this to `true` enables merging strongly connected components on the entire SDG thus strongly connected components that span multiple methods can be merged. This can also only be used in combination with the context-insensitive slicing algorithm. The implementation resides in class `SDGStronglyConnectedComponents`.
`slicer`	Chooses the slicing algorithm. The possible values are `context-insensitive`, `summary-two-phase`, `limited-context` and `iterated-two-phase-slicer`. The slicing algorithms are explained in the next table.
`slice-direction`	The value is either `backward` or `forward` and determines if a backward or forward slice is calculated. The `limited-context` slicer does not support forward slicing. When this slicing algorithm is chosen, the calculated slice is always a backward slice.

Multiple slicing algorithms are present in the slicing platform. The table below describes the slicing algorithms that can be chosen with the key `slicer`:

Algorithm	**Description**
`context-insensitive`	The context insensitive algorithm, which is basically just determines the nodes reachable from the slicing criterion. This leads to slices that are correct but contain irrelevant function calls, as we have seen in 4.4. This slicing algorithm supports both backward and forward slicing. The implementation of this slicing algorithm resides in class `ContextInsensitiveSlicer`.

`limited-context`	Use the limited context slicing algorithm [41]. It limits the number of methods in the calling context to the number set by key `context-limit`, which has a default value of 10. This slicing algorithm supports only backward slicing and is not able to deal with inter-procedural dependence. The main reason it is implemented in the slicing platform is to show that the framework can be extended with different slicing algorithms. The implementation of this slicing algorithm resides in class `LimitedContextSlicer`.
`summary-two-phase`	Uses the summary two-phase slicing algorithm as shown in [41].This slicing algorithm supports both backward and forward slicing but is not able to deal with inter-procedural dependence. The implementation of this slicing algorithm resides in class `SummaryTwoPhaseSlicer`.
`iterated-two-phase`	Use the iterated summary two-phase slicing algorithm as shown in [26], and is a slightly modified version of the summary two-phase slicing algorithm. This slicing algorithm supports both backward and forward slicing and is also capable of dealing with inter-procedural dependence. The implementation of this slicing algorithm resides in class `IterateTwoPhaseSclier`.

5.4.4 Command line interface

The slicing platform has a command line interface which can be used to invoke the slicer. The command line interface has two commands. The first command, `sdg` produces an SDG from the given program and store it in an `.sdg`-file. This command is invoked with the following command:

```
javasdgslicer sdg <input configuration> <input project> <output sdg> (<output dot>)
```

The program expects a project file while is a text file that specifies the directories that contain application classes and the directories and/or `.jar` files that contain library classes. It also specifies the class containing the `main` method. Text file 5.2 is an example of a project file. The line starting with `main` specifies the class containing the `main` method of the program, which is a method that has the signature `public void main(String[] args)`. The line starting with `src` specifies a directory that contains application classes in the form of Java source code (`.java` files). The directory is searched recursively for Java source files. The `libjre` line indicates that the Java Runtime Environment (JRE) classes of the Java installed on the computer running the slicing platform are part of the project as well. This line must be omitted in case the project includes another version of the JRE classes than the one installed. This line can also be omitted when the program to be sliced does not make use of JRE classes, such as small programs created to test the slicing platform. The line starting with `libjars` adds all `.jar` files that reside in the specified directory as library classes. The directory is searched recursively. Lines that start with `lib` add a directory with library classes in the form of Java classes (`.class`-files) to the project; this option does not occur in the example.

The output of this command is an `.sdg` file and optionally a `.dot` file, which contains the system depdendence graph in a format that can be visualized with GraphViz [4].

Text file 5.2 Example of a project file

```
main: javasdgslicer.Main
src: src
libjre
libjars: dist/lib
```

The second command, `slice` calculates a slice using an `.sdg`-file created with the previous command. Other parameters it takes are the configuration, the system dependence graph created with the first command, and the slicing criterion. This command is invoked with the following command:

```
javasdgslicer slice <input configuration> <input sdg> <slicing criterion>
    [<output specification>],
```

The slicing criterion is specified in the form `<filename>:<line number>:<column number>`. The column number can be omitted in case the line that contains the statement of interest only has one statement. The slicing criterion is then of the form `<filename>:<line number>`. After the slicing criterion, zero or more output specifications can be given. If no output specification is given on the command line, the slice is written to the standard output in the form of Java source code. The output specification `linear:<filename>` writes all Java source code that is in the slice to the specified file (classes from different input files all end up in this single file). The output specification `dir:<directory>` writes all Java source code to the specified directory where each class ends up in its own Java source file. The slicing platform will also create subdirectories for packages, so that the resulting folder forms a valid Java program which can be compiled by a Java compiler. There is a third output specification, namely `graphic:<directory>` which also places the output in a directory, but rather than writing Java source files it writes `.eps` files that contain the source code, where the parts of the source text that are in the slice are marked. This essentially provides a simple method of visualizing the slice. Section 5.5.1 has plenty of examples of this visualization, such as Figure 5.6.

5.5 Validation

In this chapter we have seen in which way a Java 6 slicer can be built that is capable of producing slices in the form of syntactically correct, successfully compiling Java 6 programs. This chapter validates the correctness of the slicing platform I built as part of this project.

5.5.1 Correctness

The correctness validation tests the slicing platform on the `SumProd` program (Program 2.1 and the challenges given in Section 3.1). The challenges for threading, reflection and dynamic class loading are omitted, since these features are not supported by my slicing platform. Support for threading could be added to the slicing platform by adding dependence generators for interference, synchronization and ready dependence (see Section 4.7). Support for reflection and dynamic class loading could not be added, because of limitations of static slicing (see Section 3.1.6). The correctness validation uses two different configurations that differ in the slicing algorithm that they use. Configuration A (Text file 5.3) uses the context insensitive slicing algorithm. Configuration B (Text file 5.4) uses the iterated two-phase slicing algorithm. Each of the configurations enables all types of dependence, except for divergence since all tested programs are known to terminate. This correctness validation only considers the context-insensitive and iterated two-phase slicing algorithm since these are the only two implemented slicing algorithms that produce correct slices in the presence of inter-procedural dependence.

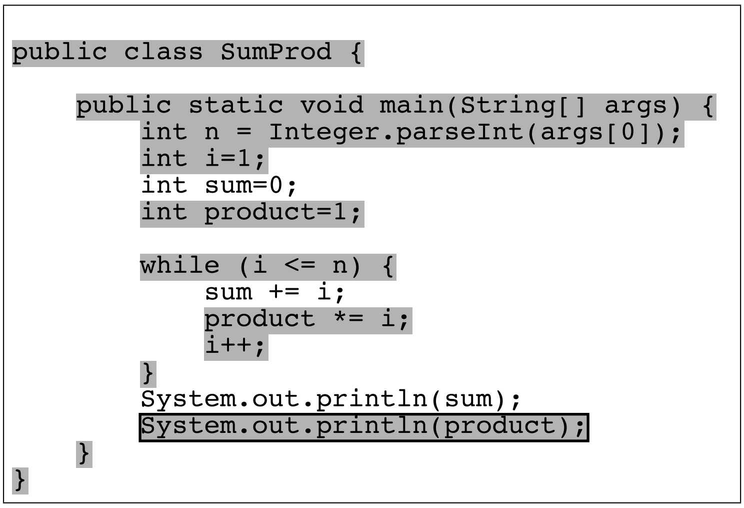
Text file 5.3 Configuration A : context insensitive slicing

```
// Points-to analysis
points-to-analysis = spark

// IDG dependence
idg-data-dependence = true
idg-control-dependence = true
idg-divergence-dependence = false
idg-structure-dependence = true

// IDG transformations
idg-text-ranges = true
idg-reduction = true
idg-strongly-connected-components = true

// SDG dependence
sdg-call-dependence = true
sdg-data-dependence = true
sdg-structure-dependence = true
sdg-summary-dependence = false

// SDG transformations
sdg-reduction = true
sdg-strongly-connected-components = true

// Slicer
slicer=context-insensitive
slice-direction=backward
```

Figure 5.6 shows the slice for the `SumProd` program. The shaded text is the text that is in the slice, and there is a box around the slicing criterion. The slice given by the slicing platform (Figure 5.6) is the same as the slice given in Program 2.2, the slice of `SumProd` that was determined in an intuitive way in Chapter 2. Thus the slice is correct. Since `SumProd` is a program that consists of only one procedure, it does not make a difference which slicing algorithm is used.

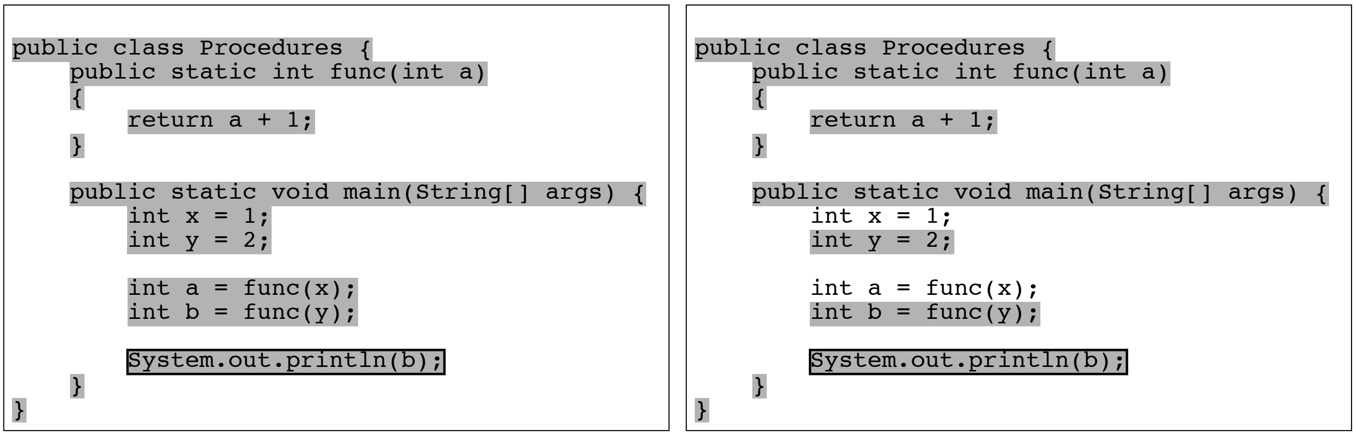
Text file 5.4 Configuration B : iterated two-phase slicing, only contains the keys that differ from configuration A

```
// SDG transformations
sdg-reduction = false
sdg-stronglyconnectedcomponents = false

// Slicer
slicer=iterated-two-phase
slice-direction=backward
```



```
public class SumProd {

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int i=1;
        int sum=0;
        int product=1;

        while (i <= n) {
            sum += i;
            product *= i;
            i++;
        }
        System.out.println(sum);
        System.out.println(product);
    }
}
```

Figure 5.6: Slice of `SumProd` (configuration A and B)



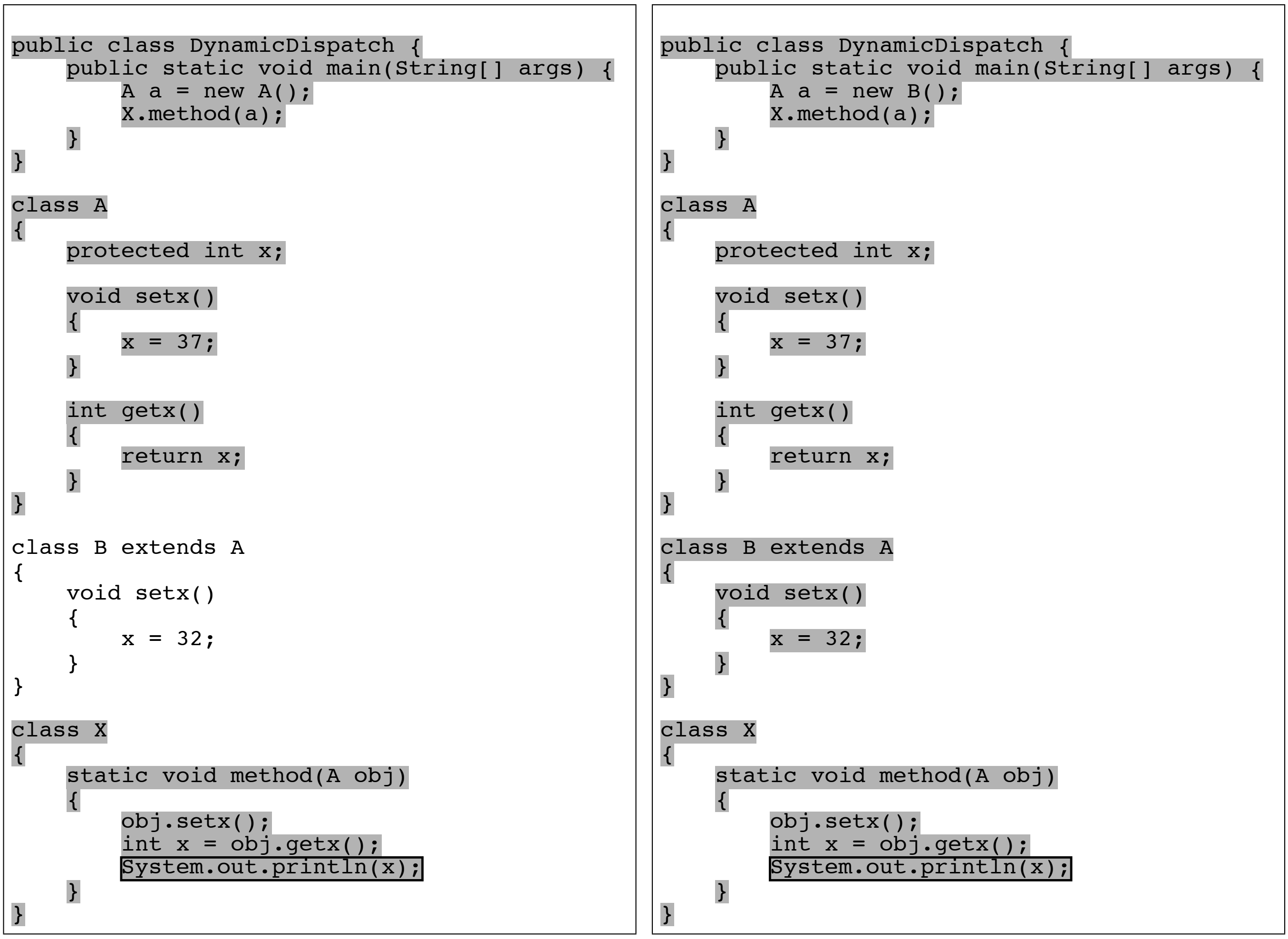
```
public class Procedures {
    public static int func(int a)
    {
        return a + 1;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;

        int a = func(x);
        int b = func(y);

        System.out.println(b);
    }
}
```

(a) Configuration A

```
public class Procedures {
    public static int func(int a)
    {
        return a + 1;
    }

    public static void main(String[] args) {
        int x = 1;
        int y = 2;

        int a = func(x);
        int b = func(y);

        System.out.println(b);
    }
}
```

(b) Configuration B

Figure 5.7: Slices of `Procedures`



```
public class DynamicDispatch {
    public static void main(String[] args) {
        A a = new A();
        X.method(a);
    }
}

class A
{
    protected int x;

    void setx()
    {
        x = 37;
    }

    int getx()
    {
        return x;
    }
}

class B extends A
{
    void setx()
    {
        x = 32;
    }
}

class X
{
    static void method(A obj)
    {
        obj.setx();
        int x = obj.getx();
        System.out.println(x);
    }
}
```

(a) Object of type `A`

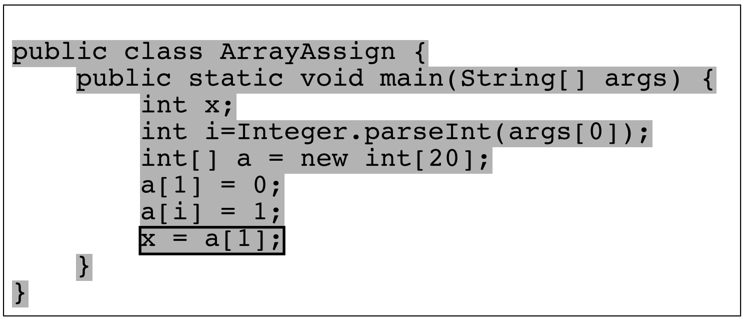
```
public class DynamicDispatch {
    public static void main(String[] args) {
        A a = new B();
        X.method(a);
    }
}

class A
{
    protected int x;

    void setx()
    {
        x = 37;
    }

    int getx()
    {
        return x;
    }
}

class B extends A
{
    void setx()
    {
        x = 32;
    }
}

class X
{
    static void method(A obj)
    {
        obj.setx();
        int x = obj.getx();
        System.out.println(x);
    }
}
```

(b) Object of type `B`

Figure 5.8: Slices of `DynamicDispatch` for objects of different types (configurations A and B)

For the `Procedures` program the chosen slicing algorithm does make a difference (Figure 5.7). The slicing criterion only uses variable `b`, so the first function call and the declaration of variable `x` can be left out. The context-insensitive slicing algorithm (Figure 5.7(a)) is not capable of seeing this but the iterated two-phase slicing algorithm is capable of this (Figure 5.7(b)). Both slices are correct, but the slice produced by the iterated two-phase slicing algorithm is more precise.

Figure 5.8 shows two slices for two variants of the `DynamicDispatch` program. Both configurations give the same slice for this program. The `DynamicDispatch` program given in Section 3.1 is not a complete program since it has no main method. Figure 5.8 shows the `DynamicDispatch` program with two different main methods added. The first one (Figure 5.8(a)) creates an object of type `A` and the second one (Figure 5.8(b)) creates an object of type `B`. When the object is of type `A`, the slice includes the entire program, except for class `B` and its method, which is indeed correct. However, when the object is of type `B`, both `setx` methods are in the slice. The slice is obviously correct since it consists of the entire program, but the statement `x = 37;` could be removed since the program calls the `setx` method of an



```
public class ArrayAssign {
    public static void main(String[] args) {
        int x;
        int i=Integer.parseInt(args[0]);
        int[] a = new int[20];
        a[1] = 0;
        a[i] = 1;
        x = a[1];
    }
}
```

Figure 5.9: Slice of the `ArrayAssign` program (configurations A and B)

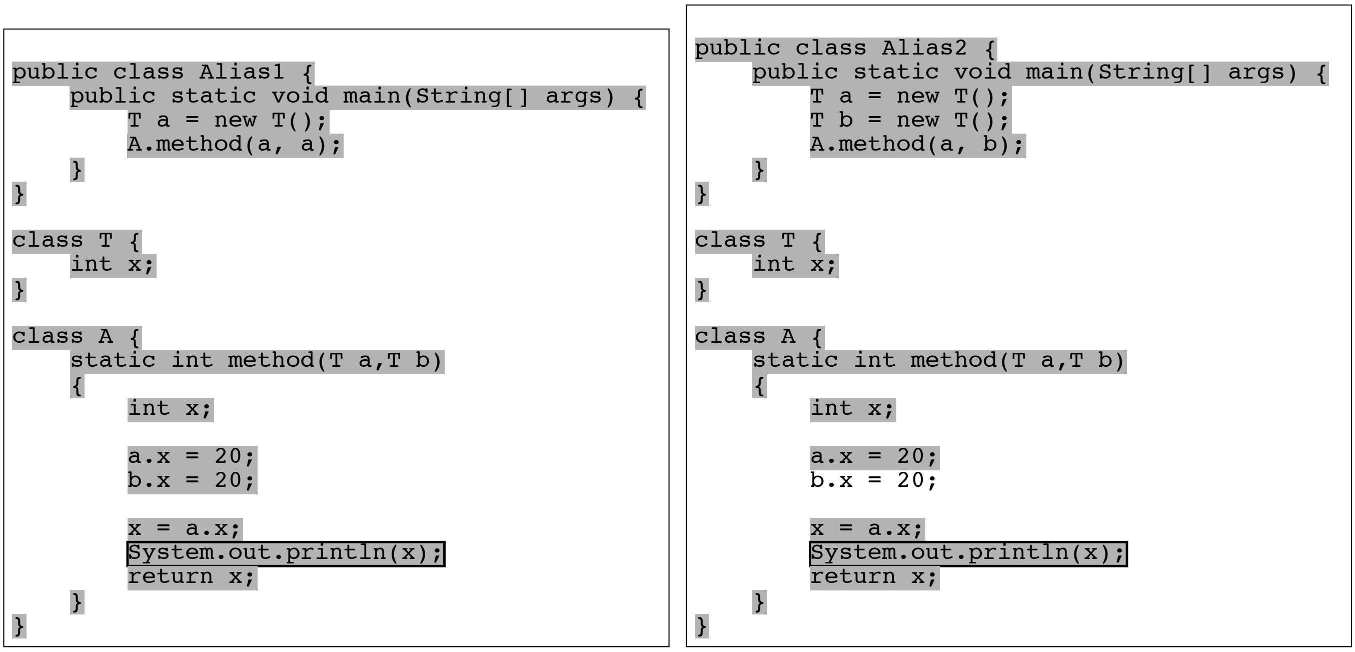
object that is known to be of type `B`. Note that the method body of method `setx` in class `A` needs to be in the slice, or else it would not be possible to call method `setx` on an object that resides in a variable of type `A` such as in this program. The fact that the program contains the `x = 37;` statement which is not needed in the slice is caused by the points-to analysis which is unable to determine that paramater `obj` in `method` can only be of type `B` but not `A`.

Slicing the `ArrayAssign` program (slice in Figure 5.9) results in a slice that consists of the entire program for both configurations. This is caused by the fact that the value of `i` is not known during run time, thus the slice needs to keep both the statements `a[1] = 0;` and `a[i] = 1;` in the slice. However, since the slicer treats every element of an array as the same object, the slice would still consist of the entire program, even if variable `i` is known to be `1` or known to have another value than `1`. Since the slice consists of the entire program, it is obviously correct. Since the program consists of only one procedure, the slice is the same for each of the configurations.

The last challenge program treated here is the `Alias` program. Figure 5.10 contains two slices for this program. Just like the `DynamicDispatch` program, this program is not a complete program. Therefore, Figure 5.10 contains two variants of the program. There is one variant where the two parameters of `method` are aliased (figure 5.10(a)) and one where they are not (figure 5.10(b)). The slices are the same for both configurations A and B. For variant 1, where parameters `a` and `b` are aliases, both assignments to field `x` are in the slice. The assignment `a.x = 20;` could be left out, since its value is overwritten by `b.x = 20;`. This is caused by the fact that the implementation of data dependencies for fields is a simplification that does not take reassigment of field values into account. In the slice of variant 2, the assignment `b.x = 20;` is left out of the slice, since it is known that `a` and `b` are not aliases. Thus the value of `a.x` does not depend on any value assigned to `b.x`. Both slices are correct, even though the slice of variant 1 of the `Alias` program contains a statement that could be left out.

The slicing platform also contains implementations of the summary two-phase slicing and the limited-context slicing algorithm. Both of these algorithms are capable of computing precise slices for programs consisting of multiple procedures. The slices produced by these two algorithms are identical to the slice shown as 5.7(b). However, neither of the algorithms is not capable of dealing with inter-procedural dependence. When slicing variant 1 of the `DynamicDispatch` program, both algorithm give the same incorrect slice shown in Figure 5.11.

The problem here is that the call to `setx` is not in the slice while the statements in the



```
public class Alias1 {
    public static void main(String[] args) {
        T a = new T();
        A.method(a, a);
    }
}

class T {
    int x;
}

class A {
    static int method(T a,T b)
    {
        int x;

        a.x = 20;
        b.x = 20;

        x = a.x;
        System.out.println(x);
        return x;
    }
}
```

(a) Variant 1

```
public class Alias2 {
    public static void main(String[] args) {
        T a = new T();
        T b = new T();
        A.method(a, b);
    }
}

class T {
    int x;
}

class A {
    static int method(T a,T b)
    {
        int x;

        a.x = 20;
        b.x = 20;

        x = a.x;
        System.out.println(x);
        return x;
    }
}
```

(b) Variant 2

Figure 5.10: Slices of the `Alias` program (Configuration A and B)

`setx` method are (only those of class `A`, since no object of type `B` is produced in this variant of the `DynamicDispatch` program). This is caused by the fact that the return statement of the `getx` method has an inter-procedural dependence on the `x = 37;` statement. However, both algorithm enter the `getx` method by following a `parameter-out` edge but do not follow the `call` edge connecting the definition of `setx` with the call to `setx`, since these `parameter-out` and `call` edges have different call sites.

5.5.2 Robustness verfication

In order to validate the robustness of the slicing platform, it was tested on a large program. The slicing platform itself is not very large, but it makes use of Soot, JastAddJ and the Java runtime environment which adds a large amount of classes. In this verification, Soot and JastAddJ are treated as library classes and the classes of the slicing platform itself are treated as application classes. The configuration used for this is configuration A (Text file 5.3) with reduction and strong connected components on the SDG level disabled (`sdg-reduce` and `sdg-strongconnectedcomponents` set to `false`).

Creating the system dependence graph lead to a graph with 574832 nodes and 2956852 edges. Producing the system dependence graph took about 10.5 minutes on a system with two Intel Xeon CPUs running a 2.66 GHz and 16,0 GB of internal memory. The Java virtual machine was ran with the argument `-Xmx12g` which sets the maximal heap size to 12 GB. This was necessarry, since the slicing platform failed with an out-of-memory exception when the maximal heap size was set to 8 GB.

```
public class DynamicDispatch1 {
    public static void main(String[] args) {
        A a = new A();
        X.method(a);
    }
}

class A
{
    protected int x;

    void setx()
    {
        x = 37;
    }

    int getx()
    {
        return x;
    }
}

class B extends A
{
    void setx()
    {
        x = 32;
    }
}

class X
{
    static void method(A obj)
    {
        obj.setx();
        int x = obj.getx();
        System.out.println(x);
    }
}
```

Figure 5.11: Slice variant 1 of the `DynamicDispatch` program, using either the summary two-phase slicing algorithm or the limited-context algorithm

5.5.3 Validation conclusion and limitations of the slicing platform

This section presented a validation study for the correctness of the slices calculated by the slicing platform. The validation study compared two different algorithms (context-insensitive slicing and iterated two-pass slicing) where both slicing algorithms provided correct slices, but the iterated two-pass slicing algorithm is capable of producing more precise slices.

From the robustness validation it can be concluded that the memory requirements of the slicing platform go up very fast for large programs, or even programs that are not large themself but make use of libraries with a large amount of classes. This is a serious limitation of the approach of building the system dependence graph into memory.

The slicer calculates slices by tracking dependence between statements. However, the slicer is not capable of tracking dependence that does not occur in the program's variables but is external to the program. An example of this is data that is stored in a file or database, and is retrieved from the file or database at a later point in the program. The value or values obtained during the retrieval depend on the data stored earlier. Another example is data sent and received via a network. The slicing platform is also not capable of handling programs that have components written in other languages than Java, such as programs that call native code from Java code using the Java Native Interface (JNI).

Also, the slicing platform can only slice Java programs that have a main method (a method that has the signature `public static void main(String[] args)`. This means that libraries, Java applets and Enterprise Java Beans (EJB) can not be sliced, unless a surrogate `main` method is added that calls the relevant methods in the library, Java applet or EJB. This provides a work-around for this problem. This limitation has two different causes. The first one resides in SPARK, the points-to analyser provided by Soot. SPARK requires a main method in order to perform the analysis. Resolving this problem requires either modifying SPARK, or adding a different implementation of points-to analysis to the slicing platform. The other cause is the fact that the slicing platform only creates intra-procedural dependence graphs for methods that are reachable from the `main` method. This problem is easier to solve, since it is possible to allow the user to specify multiple main methods, or simply assume that all methods in applications classes are main methods.

The current version of the slicing platform is limited to sequential Java programs (programs that have no concurrency), however it can be extended with extra types of dependence and extra slicing algorithms thus enabling the slicing platform to slice concurrent Java programs.

Chapter 6

Conclusion and future work

6.1 Conclusion

As mentioned in the introduction, at the start of the project there was no slicer for Java 6 that is capable of creating static slices in the form of correct Java programs. The goal of this project is to create a slicing platform that is capable of producing such slices. The result of this project is a slicing platform based on the Soot [10] framework which is capable of creating correct slices for sequential Java programs (Java programs that have no concurrency). These Java programs must be programs that are purely written in Java (program using multiple languages are not supported). The slicing platform is based on system depedence graphs, graphs that consist of a node for each statement and edges that indicate dependence, indicating which statements depend on each other. The slicing platform constructs the system dependence graphs in internal memory, thus the maximal size of the programs that can be sliced is determined by the amount of internal memory available. The slicing platform is modular, in the sense that it allows extending it with other slicing algorithms, other types of dependence, or more efficient or precise implementations of calculating dependence types that are already present in the slicing platform.

6.2 Future work

The preceding chapters already made some directions for future work clear. The precision of slices can be improved by making use of partial evaluation. This allows removing parts of the input program that do not apply for a certain case.

The slicing platform does not support Java programs that make use of concurrency. Adding support for concurrent Java programs would then require the inclusion of the calculation of the three concurrent dependence types (interference, synchronization and ready dependence). The slicing platform is written in a modular way in order to allow adding extra dependence types and slicing algorithms.

The data dependence for arrays and fields of objects is imprecise (yields larger slices than necessary) under certain circumstances, as seen in the correctness validation in Section 5.5.1. Improved implementations for these types of dependence can be provided by adding components to the slicing patform that replace the current implementations.

Java software systems such as librariers, Enterprise Java Beans and Java Applets which do not have a `main` method, even though a work-around is to create a surrogate `main` method.

A direction for future work is extending the slicer platform in such a way that it is capable of slicing programs that do not have a `main` method, so that this workaround is no longer needed.

The slicing platform is not capable of determining external dependence, such as dependence caused by data that is stored and retrieved from a file, or the use of embedded languages such as SQL. A direction for future work is enabling the slicing platform to track such external dependence, by adding modules to the slicing platform that calculate these types of dependence.

The slicing platform deals with large dependence graphs by applying methods that reduce the size of the system dependence graph. Another solution to this problem which was not implemented is the on-the-fly generation of the system dependence graph. A possible direction for future work is implementing this method so it is possible to compare the two methods. Another possible direction is storing the system depedence graph into a database rather than in memory, in order to allow handling of large system depedence graphs.

The slicing platform produces slices in the form of programs that are syntactically correct and compile successfully, however it disregards comments in the source code. If the goal of slicing is to use it in another tool such as a model generator for model checking, then it is no problem that the slice does not contain any comments. However, if the the goal of slicing is to discover in which way the source program works, then the comments need to be included. The direction for future work is thus deciding which comments should be in the slice and which should not be.

Bibliography

[1] Bandera. http://bandera.projects.cis.ksu.edu. Consulted on May 31, 2011.

[2] Eclipse — The Eclipse Foundation open source community website. http://www.eclipse.org/. Consulted on May 21, 2011.

[3] Enterprise JavaBeans Technology. http://www.oracle.com/technetwork/java/javaee/ejb/index.html. Consulted on May 31, 2011.

[4] Graphviz - graph visualization software. http://www.graphviz.org/. Consulted on June 30, 2011.

[5] Indus. http://indus.projects.cis.ksu.edu/. Consulted on May 21, 2011.

[6] Indus forums, Thread Indus project status. http://projects.cis.ksu.edu/gf/project/indus/forum/?_forum_action=ForumMessageBrowse&thread_id=676&action=ForumBrowse&forum_id=34. Consulted on June 6, 3011.

[7] JSlice. http://jslice.sourceforge.net/. Consulted on May 21, 2011.

[8] Kaffe. http://www.kaffe.org/. Consulted on May 21, 2011.

[9] Kaveri. http://indus.projects.cis.ksu.edu/projects/kaveri.shtml. Consulted on May 21, 2011.

[10] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/. Consulted on May 22, 2011.

[11] VALSOFT/Joana. http://pp.info.uni-karlsruhe.de/project.php?id=30&lang=en. Consulted on May 21, 2011.

[12] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Notes*, 25:246–256, June 1990.

[13] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In Peter Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer Berlin / Heidelberg, 1993. 10.1007/BFb0019410.

[14] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[15] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.

[16] Árpád Beszédes, Csaba Faragó, Zsolt M. Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. *18th IEEE International Conference on Software Maintenance*, pages 12–21, 2002.

[17] Dragan Bošnački. *Enhancing State Space Reduction Techniques for Model Checking.* PhD thesis, Technische Universiteit Eindhoven, 2001.

[18] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *SIGPLAN Notes*, 21:152–161, July 1986.

[19] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[20] T.Y. Chen and Y.Y. Cheung. Dynamic program dicing. In *Software Maintenance, CSM-93, Proceedings, Conference on*, pages 378–385, September 1993.

[21] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Language Systems*, 16:1097–1113, July 1994.

[22] Danny Coward. What's New in Java SE 6. `http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/beta2.html`. Consulted on May 4, 2011.

[23] Daniela Carneiro da Cruz. *Methods and techniques to analyze multi-level code to explore software components.* PhD thesis, University of Minho, 2011. First draft, available at `http://alfa.di.uminho.pt/~danieladacruz/phdDCCvf.pdf`, consulted on April 13, 2011.

[24] Árni Einarsson and Janus Dam Nielsen. A Survivor's Guide to Java Program Analysis with Soot, Version 1.1. `http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf`, July 2008. Consulted on June 5, 2011.

[25] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '95, pages 379–392, New York, NY, USA, 1995. ACM.

[26] Dennis Giffhorn and Christian Hammer. Precise slicing of concurrent programs. *Automated Software Engineering*, 16:197–234, 2009.

[27] Brian Goetz. Java theory and practice: Using Java 5 language features in earlier JDKs. `http://www.ibm.com/developerworks/java/library/j-jtp02277/index.html`. Consulted on May 4, 2011.

[28] Rajiv Gupta, Mary Lou Soffa, and John Howard. Hybrid slicing: integrating dynamic information with static analysis. *ACM Transactions on Software Engineering and Methodology*, 6:370–397, October 1997.

[29] Robert J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2:33–53, 1995.

[30] Christian Hammer and Gregor Snelting. An improved slicer for Java. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 17–22, New York, NY, USA, 2004. ACM.

[31] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[32] Mark Harman, Margaret Okulawon, Bala Sivagurunathan, and Sebastian Danicic. Slice-based measurement of function coupling. *IEEE/ACM ICSE workshop on Process Modelling and Empirical Studies of Software Evolution*, pages 28–32, May 1997.

[33] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 1999.

[34] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13:315–353, 2000.

[35] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, Boston, Massachusetts, 2003.

[36] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–60, January 1990.

[37] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. *SIGSOFT Software Engineering Notes*, 19:2–10, December 1994.

[38] Ganeshan Jayaraman, Venkatesh Ranganath, and John Hatcliff. Kaveri: Delivering the Indus Java Program Slicer to Eclipse. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering*, volume 3442 of *Lecture Notes in Computer Science*, pages 269–272. Springer Berlin / Heidelberg, 2005.

[39] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28:480–503, September 1996.

[40] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Proceedings of the Second International Workshop on Software Configuration Management*, 17(7):46–55, 1989.

[41] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, Universität Passau, April 2003.

[42] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using SPARK. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin / Heidelberg, 2003.

[43] Andrea De Lucia. Program slicing: methods and applications. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 142–149, 2001.

[44] Andrea De Lucia, Mark Harman, Robert Hierons, and Jens Krinke. Unions of slices are not slices. *European Conference on Software Maintenance and Reengineering*, pages 363–367, 2003.

[45] Markus Müller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 647–656, New York, NY, USA, 2001. ACM.

[46] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. *SIGSOFT Softw. Eng. Notes*, 25:180–190, August 2000.

[47] Oracle. J2SE(TM) 5.0 New Features. `http://download.oracle.com/javase/1.5.0/docs/relnotes/features.html`. Consulted on May 4, 2011.

[48] Linda M. Ott and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 11th international conference on Software engineering*, ICSE '89, pages 198–204, New York, NY, USA, 1989. ACM.

[49] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Notes*, 19:177–184, April 1984.

[50] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages Systems*, 22:416–430, March 2000.

[51] Venkatesh Ranganath and John Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer*, 9:489–504, 2007.

[52] Thomas Reps and Thomas Bricker. Illustrating interference in interfering versions of programs. *Proceedings of the Second International Workshop on Software Configuration Management*, 17(7):46–55, 1989.

[53] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Software Engineering Notes*, 19:11–20, December 1994.

[54] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[55] Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical Report 1998-4. Technical report, Sable Research Group, McGill University, Montreal, Quebec, Canada, 1998.

[56] G. A. Venkatesh. The semantic approach to program slicing. *SIGPLAN Notes*, 26:107–119, May 1991.

[57] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[58] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 512–521, 2004.

[59] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25:446–452, July 1982.

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

[61] Mark Weiser and James Lyle. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[62] Wikipedia. Java version history. `http://en.wikipedia.org/wiki/Java_version_history`. Consulted on May 4, 2011.

[63] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, 30:1–36, March 2005.