Verifying the Concept of Union Slices on Java Programs

Attila Szegedi, Tamás Gergely, Árpád Beszédes, Tibor Gyimóthy and Gabriella Tóth
University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 544145
attila@szegedi.org
{gertom,beszedes,gyimi}@inf.u-szeged.hu
Toth.Gabriella.2@stud.u-szeged.hu

Abstract

Static program slicing is often proposed for software maintenance-related tasks. Due to different causes static slices are in many cases overly conservative and hence too large to reduce the program-part of interest meaningfully. In this paper we further investigate the concept of union slices*, which are defined as the unions of dynamic slices computed for the same (static) slicing criteria, but for different executions of the program. We verify on real-world Java programs their usefulness as a replacement to static slices. For this we investigate the sizes of a number of backward and forward dynamic and union slices, also by comparing them to the corresponding static slices. Our results show that the union slices are precise enough (backward slices are 5–20% of the program and forward slices are 5–10%, the corresponding static slices being 25–45%), and that with the saturation of the overall coverage given many different executions, union slices also reach a steady level and typically do not grow further by adding new test cases.*

Keywords

Program analysis, program comprehension, software maintenance, dynamic program slicing, union slice, Java.

1 Introduction

The notion of *program slicing* and *the slices* [17, 24, 29] have been defined in many different ways in the literature. The basic idea is, however, always the same: to attempt to reduce the size of the problem by achieving to investigate only those parts of the program to be analyzed that are relevant from a specific point of view. Therefore, in some typical applications such as maintenance in general, or in understanding or debugging the reduction rate is crucial. In other words, the smaller the slice, the better.

In this paper we present our results from experiments aimed at investigating the sizes of different slices, and thus elaborating on the reduction rate mentioned above. The issue is important since the two fundamental categories of slicing methods—static and dynamic slicing—behave very differently from this perspective. Inherently, *static slices* are execution independent and larger, while *dynamic slices* correspond to one concrete execution and hence smaller.

We also further investigate the concept of the so-called *union slices* [2], which may be a compromise solution in the cases when static slices are too large, while dynamic methods do not include but only a small portion of relevant program points. Very simply, union slices are computed as the union of dynamic slices for many test cases for the same program and (static) slicing criteria. This way they represent both concepts: they deal with concrete test cases and, at the same time, multiple possible executions are captured.

The concept of union slices is fairly obvious as the union of dynamic slices for a (finite) set of test cases. However, if we computed the union of dynamic slices for *all possible* executions, we would obtain a theoretical slice of the program that contains all realizable dependencies. We will refer to this slice as the *realizable slice*. The realizable slice can be approximated by a sequence of union slices by adding more and more dynamic slices to it (see Figure 1). As the realizable slice is generally uncomputable, we cannot be sure whether a union slice computed for a set of test cases is close enough to the realizable slice or not, and hence we cannot estimate the distance to the static slice neither (which is the upper bound for it). Our approach is therefore to examine the relation between the union slice and the *coverage*—the number of program instructions that are executed at least once during different executions of the program—and estimate the gap between these two kinds of slices using this information.

Our previous investigation of union slices, their relation

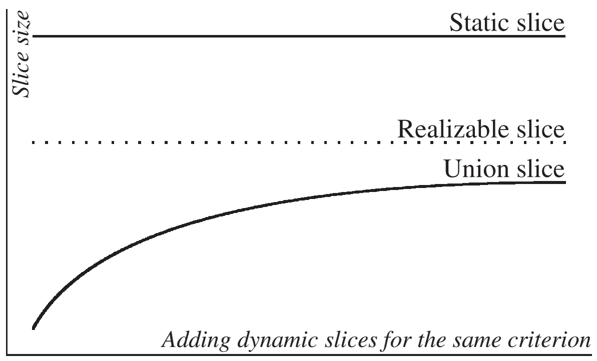


Figure 1. Approximation of the realizable slice

to the corresponding static slices and the growth tendencies of union slices was performed on medium size C programs [2]. We defined several slicing criteria using a classification of the slice variables and performed a variety of executions for these programs. We found that union slices are in most cases far smaller than static slices, and that the growth rate of the union slices (by adding more test cases) significantly declines after several representative executions of the program. In the present article we continue this work by aiming to answer the question whether the concept of union slices shows similar behavior on a different platform and different kind of applications. To this end, we applied the same approach to experiment with union slices of Java programs. We used our dynamic slicing method for *Java bytecode* programs [23], which deploys a novel approach in Java slicing. Namely, it uses an instrumented Java virtual machine for producing the execution trace, after which our forward global dynamic slicing method [3, 4] is applied. This implementation of the basic method is capable of computing both backward and forward slices, and since it is able to produce many dynamic slices globally with one run of the slicer, it is very suitable to perform the desired experiments. Furthermore, our experiments demonstrated that our tool is capable of handling real-world programs and executions.

The contributions of this paper can be summarized as follows:

- We verified the validity of the approach in Java environment on five open source Java programs, finding similar results to previous experiments for C.
- The dynamic and union slice sizes of forward slices are investigated and compared to backward slice sizes.
- We compared the resulting union slice sizes to static slices and this way recorded the benefits of the former.
- The relation between coverage and union slice sizes is also investigated.

The remainder of the paper is organized as follows. In the next section we give some background information on the concept. In Section 3 we overview our methodology for computing Java union slices, while Section 4 presents our experimental results. Section 5 deals with related work, and finally we close our paper with conclusions and some ideas for future work in Section 6.

2 Overview of the concept

One possible definition of a slice of a program is its subset that consists of all statements and predicates that might affect a set of variables at a specific program point, called the slicing criterion. This definition is sometimes more precisely referred to as the *backward slice*, since it associates a slicing criterion with a set of program locations whose earlier execution affected the value computed at the criterion. Similarly, a *forward slice* is a set of program locations whose later execution depends on the values computed at the slicing criterion.[1] Among many other applications, typical ones of backward slices are debugging and program understanding, while forward slices can be used for, say, impact analysis.

From another point of view, slices can be divided into two other categories: static slices and dynamic slices. *Static slices* represent portions of a program regardless of any concrete execution (i. e. all possible relations between program elements need to be taken into account), thus they must include a larger subset of it. On the other hand, *dynamic slices* correspond to one concrete execution, meaning that only one aspect of the program's behavior is taken into account, and naturally, resulting in a smaller subset.

The problems with these two extremities are the following. Traditionally, static slices have been specifically proposed for maintenance and program comprehension [5, 12, 15], and it is probably safe to speculate that the literature on static slicing is much broader than that of dynamic slicing. Unfortunately, in many cases the static slices are overly conservative (especially due to the dynamic aspects of modern programming languages like reflection and polymorphism) and hence too large to supply useful information. On the other hand, dynamic slicing methods (e. g. [1, 19]) can produce a narrow result for one test case, which can be quite useful for some applications such as debugging. However, in many other applications one test case is not enough to investigate, but more global information may be needed about the program.

Union slices [2] may be a compromise solution in the cases when static slices are unacceptable because of their

[1]Both slicing directions are addressed in this paper, however if not stated otherwise we will implicitly refer to backward slicing when slicing in general is mentioned.

lack of precision and dynamic methods are unfeasible because of the lack of resources needed to involve lots of test cases. Very simply, union slices are computed as the union of dynamic slices for many test cases for the same program and (static) slicing criterion. This way they represent both concepts: they deal with concrete test cases and, at the same time, multiple possible executions are captured. Naturally, while static slices are safe, union slices are smaller but, alas, unsafe (i. e. they do not contain all possibly realizable dependencies). However, this is not a problem in many applications in maintenance and comprehension, as we are able to determine the most important parts to be concentrated on, based on a chosen set of test cases. Note that it has been shown [11, 13] that a union of two dynamic slices is not necessarily a valid dynamic slice in terms of preserving the original semantics of the program, however this is not a problem either with our usage scenarios, since we do not want to preserve the executability of slices.

The verification of the concept of union slices relies on measuring *program coverage*, which is the number of those instructions of the program that were executed at least once during the different program runs. Probably the best property of union slices is that as soon as the desired degree of coverage is reached the union slices also tend to saturate at a certain level, which are, generally far smaller than the corresponding static slices. In other words, when by adding new test cases the coverage cannot be significantly increased union slices can also be treated as 'ready.' Indeed, one of our most significant findings, both in our previous work on union slices and in the present article as well, is that several representative executions of the program yield a quite usable union slice (meaning that it can be used as a replacement to static slice).

If we computed the union of dynamic slices for all possible executions, we would obtain the theoretical *realizable slice* that contains all realizable dependencies. If we consider the static slice as an upper bound of the realizable slice (we can do so because every realized dependence in some dynamic slice must be captured by the static slice as well), the union slice can be seen on the other hand, as the *lower bound* for it (see Figure 1). The most apparent advantage of the combined application of static and union slices is when they coincide, because in this case the realizable slice is surely found. However, according to our experiences, this is probable to occur only in the case of trivial programs.

The observation that the union slices are usually much smaller than static slices opens a number of potential applications of this concept, since union slices can be treated as the replacement of static ones. For instance, despite the fact that the static slices provide smaller sets of data to be investigated by the software maintainer than the whole program, this reduction will be too small to provide real help. In almost every practical situation the limited resources to conduct a maintenance task will mean a real problem because the static slices will be too large to be able to cope with. Further, the static slicing methods do not provide any kind of information about those parts of the slices that may be the most important with respect to the initial problem, i. e. what parts of the program need to be definitely investigated because they represent the real dependencies? However, with the application of the union slices, the resources for the maintenance task can be used more effectively, since the maintainer may concentrate on the most important parts of the problem first.

Therefore, the primary goal of this paper is to verify whether the concept of union slices can indeed be used as a replacement to static slices thanks to the possibility of verifying it using coverage information.

3 Computation of Java union slices

For performing the experiments we utilized our dynamic slicing tool for Java programs called *Jadys* [23] that instantiates our global method for computing dynamic slices [4]. In this section we overview the most important aspects regarding the tool's functionality and internal architecture. An important enhancement to the tool was the addition of the capability of computing forward dynamic slices, which enabled us to extend the measurements to this kind of slices, too.

3.1 The slicing toolbox

Code instrumentation is essential to obtain an execution trace that is further used to calculate the slices. Our approach was to create an instrumented Java Virtual Machine based on the code of a publicly available Open Source VM called JamVM (http://jamvm.sourceforge.net). Instrumenting at the lowest level (VM) has some obvious advantages over higher level (source code or bytecode) instrumentations. It can work without source code and it can track dependencies that arise in third-party code, in code of standard Java library classes, inside the VM itself, and to a limited extent even in third-party native machine code. It is also an advantage that instrumentation is confined to a single, finite body of code – the C source code of the VM itself. Finally, it works with programs in any language that can be compiled to Java bytecode.

Figure 2 depicts the architecture of the toolbox. One element of it is an instrumented JVM that produces the execution trace of a program as it runs. The other part is a slicer that reads the execution trace and implements the forward global method for computing backward and forward dynamic slices. The main components of the slicer are the thread multiplexer, the static analyzer, and one or more slice

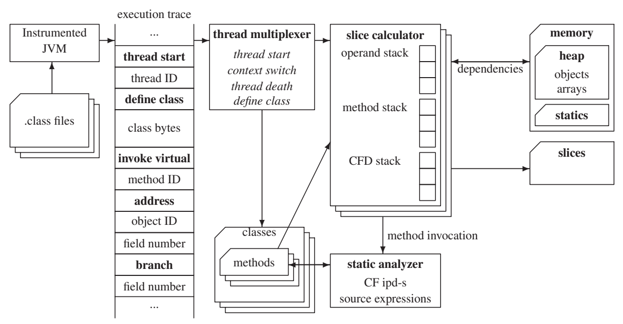


Figure 2. Experimental architecture

calculator instances. The thread multiplexer reads the execution trace, and handles all thread related events, so it creates, destroys or activates slice calculators accordingly. The static analyzer is invoked whenever a method is entered for the first time, and performs a necessary one-time analysis of the method, which is a prerequisite for the tracking of dependencies inside it.

3.2 Static analyzer

The static analysis of a method is comprised of three key operations: that of translating bytecode to source expressions, narrowing the scope of interest, and calculating control-flow information.

An important feature of the slicer is the ability to query the generated slices using source-level symbolic expressions in the slicing criterion. The slicer does not rely on the source code of the classes, rather the bytecode of the classes are passed to the static analyzer through the trace. It allows us to process on-the-fly generated code like proxy classes. The value of this approach lies in the fact that dynamic on-demand code generation has become a widespread technique in Java programming.

When the slicer reads the class bytecode from the trace, it will store it and later perform limited static analysis on it method-by-method. Part of this static analysis is the assignment of a symbolic expression to each bytecode instruction that accurately describes the result of its operation. It is trivial matter to assign a source code line number to each bytecode offset using the line number information present in the compiled class.

With the above process every non-jumping bytecode instruction became a static slicing criterion. The resulting set of slicing criteria is huge for any non-trivial program, and since our method is a forward method that, by default, calculates slices for all of them, we usually narrow the calculation to a certain subset of slicing criteria. Regular expressions for class names can be used to include or exclude them from slice calculation in their code. The scope can be further narrowed by eliminating some of the criteria in the included classes. By default, we only consider left-hand sides of assignments as well as branch predicates. We can also narrow the set of code locations that can serve as elements of dependence sets and slices. We only use locations of assignment instructions, conditional branch instructions, method return instructions, and instructions that push arguments for method calls onto the stack for this purpose.

In order to track control-flow dependencies we have to perform a static analysis of the control flow inside every executed Java method. It is sufficient to perform only procedure level slicing, since we get interprocedural dependencies from the execution trace. We partition the method code into basic blocks assuming that exception throwing and handling instructions are branch and branch target instructions. Then, we build two control flow graphs using the basic blocks as vertices: one contains only edges representing normal control flow, and another with extra edges for control transfers occurring because of exception throws.

3.3 Dynamic slice calculation

In the following, we will overview the method for computing the backward dynamic slices. The way of computing forward slices is very similar, only the dependence directions need to be appropriately exchanged, so we will not elaborate on this in detail.

The calculation of slices is driven by the events in the trace file. The component indicated in Figure 2 as a multiplexer reads the trace file. It forwards the majority of events to the currently active slice calculator.

Each data manipulating instruction overwrites a dependence set at its target or calculates the union of several (usually two, in some cases three) dependence sets and stores it in its target. If the current instruction is at a program location of interest, the program location will be added to the resulting dependence set. The elements of the currently effective control-flow dependence set are also added to the resulting set. For tracking the intraprocedural control dependencies, each slice calculator updates the active control flow dependencies whenever a branch instruction is encountered or a postdominator of the proper block is reached. When interprocedural control flow dependencies are calculated, the active dependencies remain in effect when a method is entered or exits (normally or abruptly). As each method invocation instruction is capable of throwing an exception in Java, these instructions always end a basic block. Dependencies arised in a called method are dropped only when a postdominator of such a basic block is reached. This happens immediately on normal returns. Otherwise the control flow dependencies remain in effect until the end of exception handling blocks of the caller. If an exception handler itself exits abruptly, the control flow dependencies will propagate further to the next caller, and so on.

Finally, if the expression associated with the currently executed bytecode instruction and its code location are a slicing criterion of interest, the dependence set qualifies as a slice for that criterion and is appended to the list of slices for that criterion. The static slicing criterion (V, l) is thus effectively extended to a list of dynamic slicing criteria (V, l, i), where i is an index in the slice list assigned to the static criterion (V, l).

3.4 Calculating union slices and coverage information

After we obtained slices from several executions of the same program, the slice calculator engine can merge those slice sets to form union slices. Assuming all executions used the same code-base, we can identify slicing criterions by matching class names, method signatures, and bytecode offsets, and then simply calculate the union of slices for each criterion separately.

The extent to which our slice calculator emulates the execution of the sliced program is sufficient to ensure precise instruction-level coverage measurement on it. By preserving the coverage information from separate runs we are able to maintain precise coverage information for the unions too.

4 Experimental results

In this section we present the environment we used for experimentation and the results of our measurements. Creating the measurements consisted of several steps. First, we choose five open source command-line Java programs to be measured. Static slices were computed by *Indus* [16], a Java slicer and static analysis tool. This process required the source code only. However, to compute dynamic and union slices the programs must be executed on some input data, so test cases were generated for all programs. Then the programs were executed on the test inputs using the instrumented java virtual machine *jamvm* that generated execution traces. These traces were processed by *Jadys* that produced coverage information and the dynamic slices. Finally, the dynamic slices were unioned step by step to achieve data on union slice growth.

4.1 Test programs

We selected five open source medium size Java programs for use in our experiments and we defined a number of different test cases for each of them. The first program, *RayTracer* is part of the Java Grande Forum Benchmark Suite (http://www.epcc.ed.ac.uk/javagrande/index_1.html). We slightly modified this program to accept some parameters instead of only computing one of the two fixed scenes. *JSubtitles* (http://sourceforge.net/projects/jsubtitles) is a subtitle converter, while the next program, *NanoXML* (http://nanoxml.cyberelf.be/) is a small XML parser library, whose package contains an example program called *DumpXML* that drives the library classes. We used this program in our experiments. The program *java2html* (http://www.java2html.de/) converts java source code into syntax-highlighted html, tex, rtf and other formats, and finally *dynjava* (http://koala.ilog.fr/djava/index.html) is a dynamic java source code interpreter that executes programs written in dynamic java language.

Some parameters of these test programs and the number of test cases we defined are presented in Table 1. The number of program lines is the number of those lines for which bytecode instructions were generated, summarized for all classes in the program.

Test program	Classes	Lines	Test cases
RayTracer	12	340	20
JSubtitles	15	460	100
NanoXML	27	1156	96
java2html	55	2290	95
dynjava	302	17447	25

Table 1. Program sizes

4.2 Test cases and coverage

The test cases we defined for the programs can be divided into two sets. The first half of the executions were aimed at covering only small but different parts of the program (e. g. wrong parameter handling) if it was possible, thus we used prepared inputs. We examined the programs in order to identify specific parts of them that can be executed by a well suited input data. The rest of the executions were general meaning that real (not prepared) inputs were given to the programs. In some cases we used the test data that came along with the program, and in other cases real inputs were collected from the internet. Then we created two lists for each program: the first list contained the prepared inputs in random order and the second contained the general inputs also in random order. The two lists were then concatenated. This order of the test cases in these lists determined the order in which the dynamic slices were unioned.

The programs were executed for all inputs in their own test case lists. Table 2 shows the number of executed bytecode instructions the different test cases produced. The execution traces were generated by the instrumented *jamvm* version 1.4.1 (with *classpath* version 0.19).

The slices were computed using our global slicing algorithm for all reasonable slicing criteria within the test programs. Technically this means that we computed slices for all possible slicing criteria in the executed code except for

Program	Exec. min.	Exec. max.
RayTracer	2 598 546	21 525 307 460
JSubtitles	516 213	55 459 126
NanoXML	910 806	94 754 237
java2html	1 541 531	20 370 505
dynjava	4 019 365	6 369 636

Table 2. Executed instructions

those that belong to `java.*`, `javax.*`, or `gnu.*` packages. Similarly, in the output no source code lines from the mentioned packages were included in the slices.

The output of the slicer tool is a set of source code line numbers, thus we counted slice sizes in source code lines. The relative slice sizes (given in percentages in the following) are the slice sizes divided by the total number of source code lines of the given program (not just the loaded ones).

Finally, we computed the coverage information based on the number of bytecode instructions. The relative coverage sizes are comparable to slice sizes, so in the following we use percentage values here as well.

4.3 Static slices

To compare our results with static slice results, we computed static slices using *Indus* [16], a Java slicer and static analysis tool. The slicer has many configuration options, but eventually we have chosen a non-termination sensitive backward slice configuration with both intra- and interprocedural divergence analysis, context sensitive deadlock criteria selection strategy, and symbol based ready- and interference dependencies. This configuration produces non-executable slices.

As Indus works on a low level intermediate code representation called *jimple*, and the source code mapping is not part of it, our static slice size computation is based on the number of jimple statements. Like in the dynamic size computations, we considered only the classes of the test programs, and excluded library classes. The first three columns of Table 3 show results on static slice sizes (average, standard deviation and maximal values shown). Because of the memory problems of Indus, we could not manage to compute static slices for *dynjava*.

4.4 Dynamic slices

First we computed the dynamic slices for all available criteria in one step for each execution, and investigated the individual dynamic slice sizes. Taken into account that we have some very long executions, we would get a huge number of different dynamic slices if we considered each occurrence of every statement as a distinct criterion. So we actually calculated dynamic slices for all *static* slicing criteria, which means that the dependence sets for different occurrences of the same instruction were combined internally in the slicer (we use static criteria for union slices anyway). Both backward and forward slices, and in addition, coverage information were calculated.

The average backward dynamic slice sizes with standard deviation and maximal values, the number of static slicing criteria in the program and the total number of computed slices (coming from different test cases) for the test programs are presented in the middle part of Table 3.

Almost all programs produced quite different dynamic slice sets for their inputs, except RayTracer, which, being part of the Java Grande benchmark, is aimed at performance measurement, meaning that it is computationally intensive but not too complex in terms of control flow. That is the reason why its executions included so many instructions, and why it produced only a very small number of different dynamic slice sets for its 20 different inputs.

4.5 Union slices

Our next experiment was the investigation of union slices. We computed both forward and backward union slices, and also the *union coverage* – the number of bytecode instructions executed during any test run (a simple union of individual coverages). The last three columns of Table 3 show the average values of union slice sizes and the final coverage (at the last test case) for all test programs. The deviation values were between 3 and 9%, except for RayTracer whose deviation was much higher (25%). The maximal slice sizes were between 9 and 60%. We have to remark that no forward slices were computed for dynjava due to technical reasons.

From these numbers it can be easily seen that union slices are significantly smaller than static slices. More details can be observed in Figure 3, which shows the distribution of union slice sizes per test program. Binkley and Harman presented [7] that forward static slices are smaller than backward slices. We found that (in general) forward union slices are also smaller than backward union slices.

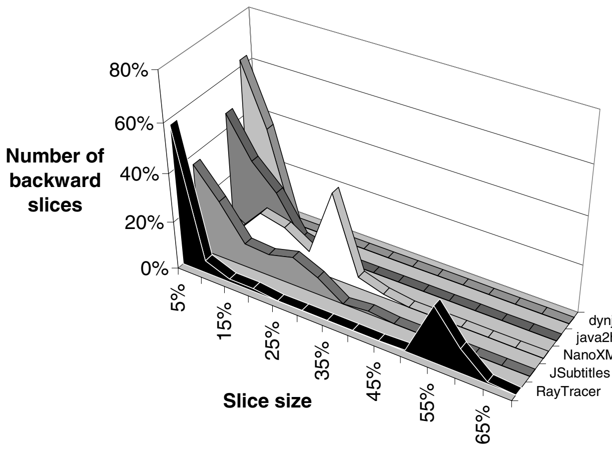
In Figure 4 we present the Monotone Slice-size Graphs [8] for backward and forward union slices. As can be observed, a lot more of the smaller slices can be found among the forward slices, but the maximum sizes are about the same as with the backward slices.

4.6 Union slice and coverage growth

We cumulatively summarized the individual dynamic slices with each execution and investigated how the resulting union slices grow test case by test case towards the static slice. In addition, we also monitored the combined coverage. By examining the results we realized that both the

Program	Static slice size avg.	dev.	max.	Dynamic slice size avg.	dev.	max.	Number of dynamic criteria	slices	Average union slice backward	forward	Union coverage
RayTracer	45%	37%	92%	9%	16%	58%	248	4167	20%	5%	87%
JSubtitles	86%	24%	93%	6%	6%	36%	260	25168	11%	8%	72%
NanoXML	25%	29%	59%	8%	5%	23%	652	58428	19%	10%	63%
java2html	82%	21%	89%	3%	3%	16%	1191	106300	6%	3%	61%
dynjava	N/A	N/A	N/A	2%	2%	7%	4823	120575	4%	–	25%

Table 3. Program and execution data, static and dynamic slice sizes. Percentage values shown with respect to program size.



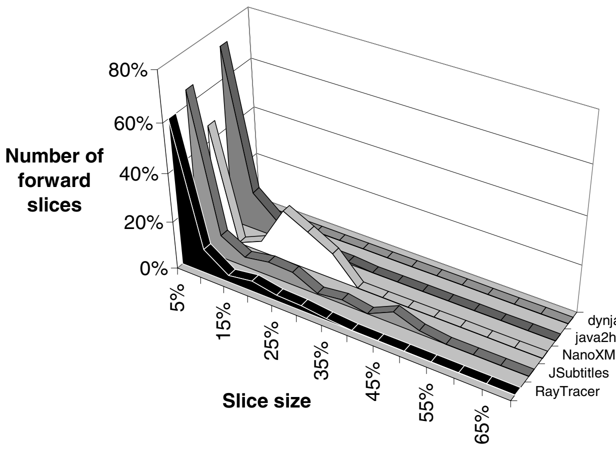


Figure 3. Distribution of backward and forward union slice sizes. The x axis shows the slice sizes measured in percentage of the program size, while the y axis is the portion of the slices of the given size out from all slices computed.

coverage and union slice sizes reached more than 95% of their maximal value (which was well below the half of the static slice size) after executing the programs with half of the prepared input set. Figure 5 shows the coverage and average slice size growth for two test programs and backward slices, whose results were typical and represent the rest of the results well.

What can be instantly observed is that the union slice sizes tend to follow the growth of the coverage. In fact, we computed the correlation between slice sizes and coverage for this data set, and found that it was very high (0.89–0.96). In other words, it is a good property that union slices do not grow further by adding more test cases. This means that if one can reach a desired coverage level (by having a suitable test suite), the union of the dynamic slices for these test cases will be a good replacement to the static slice. Furthermore, if a very high coverage is reached it is probable that the realizable slice is well approximated.

When we compared union slice results to static slice sizes we found it surprising that for *JSubtitles* and *java2html* the average static slices are high, while the average union slices are comparably small. For the other two programs, the static and union slices are much closer to each other. This phenomenon remains to be investigated, however our initial guess is that the test suites for these two programs were less well prepared to involve many different slices.

By investigating the growth of both kinds of dynamic slices, it can be clearly observed here as well that the forward slices are smaller than the backward ones. Figure 6 shows the average union slice growth for three programs and both slice directions. The shapes of the two curves for the same program look very similar, only their 'heights' are different.

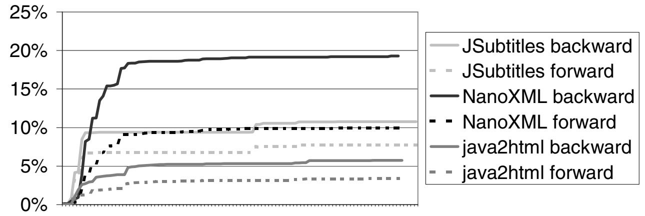


Figure 6. Backward and forward slice growth

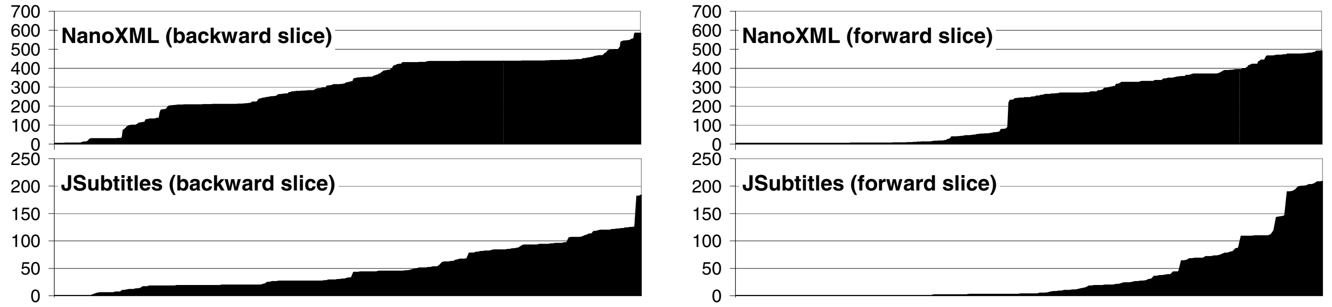


Figure 4. Monotone Slice-size Graphs for backward and forward union slices. The slices are stacked in increasing order with respect to their sizes, which are shown in the number of instructions.

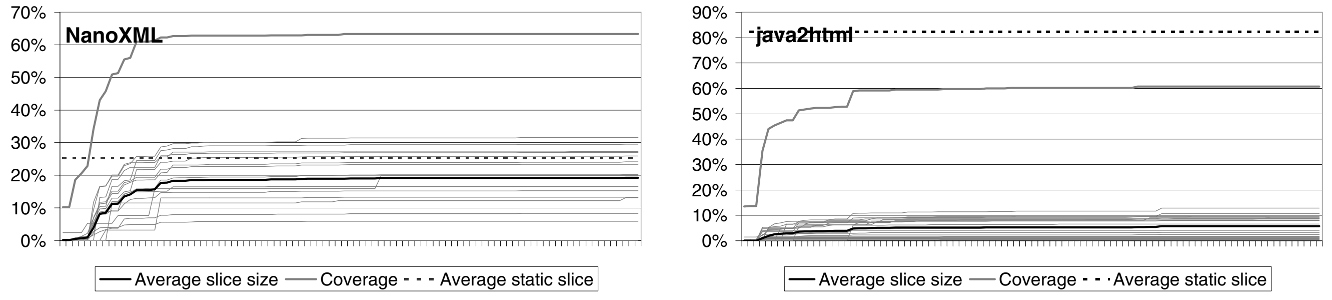


Figure 5. Coverage and average slice size growth. Sizes shown as percentage relative to the program size.

5 Related work

The problem of reducing the portion of the program that needs to be investigated (i. e. reducing the size of slices), while retaining some of the advantages of static slicing (in terms of slice generality) has been investigated by other researchers as well. *Conditioned slicing* [9, 14] computes a subset of the program which preserves the behavior of the original program with respect to a slicing criterion for a given set of execution paths. The main difference between conditioned slicing and our approach is that the former is primarily a static approach while trying to involve some dynamic information (but without actually performing the executions).

Combined use of static and dynamic slicing methods can be found in several papers, but they all have some different objectives than ours. Venkatesh introduced a hybrid slicing method [26] to compute the *quasi static slices* where the values of some input variables were fixed while other variables varied. Another example of a hybrid slicing method is the work of Rilling *et al.* [22]. They introduced a framework for the computation of both static and dynamic slices based on the notion of removable blocks [18, 20]. The objective of this work is again not to reduce the size of the parts of the program to be investigated, but to ease the computation of the dynamic slices by removing certain parts of the program first using static slicing techniques.

Significant resemblance between our approach and the work of Hall [13] can be identified. He introduced the notion of *simultaneous dynamic program slicing* to extract executable program subsets (for program subsetting and redesign). Hall was motivated by the fact that simple unioning of the dynamic slices (what exactly our approach is) cannot produce correct slices in terms of executability on all the test cases. This result was extended by De Lucia *et al.* [11] for other forms of slicing including static slicing. However, our motivation was not to create executable slices but only program parts that can be utilized in a number of applications. It is important to note that Danicic *et al.* presented an algorithm for computing executable union slices using conditioned slicing [10], which opens further applications of this concept.

Empirical investigation of the sizes of slices was rarely the primary research objective of researchers. Static slice sizes were intensively investigated by, for example, Binkley and Harman [6, 7]. On the other hand, Venkatesh's paper [27] is among the very few publications that deal with the evaluation of the sizes of dynamic slices. The author performed a large number of experiments to determine the typical size distribution of the dynamic slices and to investigate different kinds of slicing criteria with different kinds of variables.

One of our findings in the present article is that forward slices are typically smaller than the backward ones, which is in accordance with the results Binkley and Harman demonstrated for static slices of C programs [7].

Our basic method for the computation of the dynamic slices, namely the forward global approach, significantly differs from previous approaches, as elaborated in some of our previous work in the field, e. g. [3, 4].

Dynamic slicing Java programs has a modest literature. Umemori *et al.* [25] reported an implementation of a bytecode-based Java slicing system. Their technique, however, is a hybrid static and dynamic approach that generally yields less precise slices compared to a fully dynamic system like ours. They also state that their approach requires a custom Java compiler and is therefore essentially confined to analyzing programs written in Java source language. Wang and Roychoudhury [28] also reported an implementation of a bytecode-based Java slicing system. In contrast with our system, they use a demand driven slicing method where each slice calculation requires one traversal of the execution trace, so their main focus is that of minimizing the execution trace size and they do indeed present a novel approach for compressing the trace. Wang's implementation uses manual dependence specifications for library and third party code, but we are able to track dependencies in third party and library code as well. Zhang *et al.* [30] reported a technique for improving the efficiency of the forward slicing method for C programs that could be adapted to our Java slicing method as well. Zhao [31] published a fully static technique for slicing multithreaded Java programs. Finally, Masri [21] presents a dynamic slicing method for Java bytecode, which is similar to ours in its calculation of data and control flow dependencies. However, it does not deal with exception handling, multithreading, and dependence tracking in native code.

6 Conclusion and future work

We compute the union slice as the simple union of dynamic slices for many different executions of a given program and static slicing criterion. Static criterion is used since different executions imply a different set of dynamic slicing criteria, so the dynamic slices for a specific execution are produced also by unioning the dynamic slices for criteria corresponding to different occurrences of multiply occurring statements during the execution.

We demonstrated that union slices can be used as a replacement to static slices because of the observation that with the saturation of the overall coverage given many different executions, union slices also reach a steady level and typically do not grow further by adding new test cases. This supports our initial assumptions that the realizable slice—as defined at the beginning of this article—can be well approximated. A useful scenario for using union slices could be that given a selected set of test cases, the resulting union slices for this set may be used instead of the static slices. The advantage of this approach lies in the fact that union slices are generally much more precise than the static slices; according to our experiments, they are usually more than twice smaller.

With this paper we contributed to the topic with empirical measurements of real-world Java programs with versatile test inputs. The experiments were performed for all significant slicing criteria, and by examining both backward and forward slices. We were able to do this kind of detailed measurements thanks to the global nature of our dynamic slicing tool. We also demonstrated that our tool is capable of handling real-world programs and executions.

The backward and forward slice sizes showed similar tendencies in our experiments, with an interesting observation that forward slices are typically smaller than the backward ones (which is in accordance with the findings by other researchers [7]). This phenomenon is another interesting topic for future research, especially as one of the most important applications of forward slicing is with impact analysis for regression testing, and testing in general, which is a very important field in software engineering and software maintenance.

The graphs in Figure 4 show resemblance to what Binkley and Harman presented in their work about dependence clusters for C programs [8]. The 'plateaus' that can be observed mean that there are groups of slices with the same size, which are potential dependence clusters. In the future we plan to investigate this phenomenon with union slices.

Although our current implementation of the Java dynamic slicer was able to analyze really long execution traces as can be read in Section 4, we are planning to extend it in such a way that potentially infinite traces could be processed by the slicer; acting as a kind of slice server that stores all occurring slices on disk and waits for trace events endlessly. This kind of operation could ease the computation of union slices as well.

Acknowledgements

Our thanks goes to Dr. Hatcliff and his team for providing us help with using the static Java slicer *Indus*. This work was supported, in part, by national grant GVOP-3.1.1.-2004-05-0345/3.0 (OpenOffice++) and grant no. RET-07/2005 of the Péter Pázmány Program of the Hungarian National Office of Research and Technology.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Pro-*

gramming Language Design and Implementation, number 6 in SIGPLAN Notices, pages 246–256, White Plains, New York, June 1990.

[2] Á. Beszédes, Cs. Faragó, Zs. M. Szabó, J. Csirik, and T. Gyimóthy. Union slices for program maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 12–21. IEEE Computer Society, Oct. 2002.

[3] Á. Beszédes, T. Gergely, and T. Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, pages 21–30, Sept. 2006.

[4] Á. Beszédes, T. Gergely, Zs. M. Szabó, J. Csirik, and T. Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, Mar. 2001.

[5] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996. Marvin Zelkowitz, Editor, Academic Press San Diego, CA.

[6] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pages 44–53. IEEE Computer Society, Sept. 2003.

[7] D. Binkley and M. Harman. Forward slices are smaller than backward slices. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 15–24. IEEE Computer Society, Sept. 2005.

[8] D. Binkley and M. Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE Computer Society, Sept. 2005.

[9] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–607, 1998.

[10] S. Danicic, A. De Lucia, and M. Harman. Building executable union slices using conditioned slicing. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, pages 89–97, Bari, Italy, June 2004.

[11] A. De Lucia, M. Harman, R. Hierons, and J. Krinke. Unions of slices are not slices. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 363–367. IEEE Computer Society, Mar. 2003.

[12] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[13] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, Mar. 1995.

[14] M. Harman, R. M. Hierons, C. Fox, S. Danicic, and J. Howroyd. Pre/post conditioned slicing. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 138–147, Florence, Italy, Nov. 2001.

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

[16] `http://indus.projects.cis.ksu.edu/` Indus project: Java program slicer and static analyses tools.

[17] M. Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, Dec. 1995.

[18] B. Korel. Computation of dynamic program slices for unstructured programs. *IEEE Transactions on Software Engineering*, 23(1):17–34, Jan. 1997.

[19] B. Korel and J. W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[20] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, Washington, Aug. 1994.

[21] W. A. Masri. *Dynamic information flow analysis, slicing and profiling*. PhD thesis, Case Western Reserve University, 2005. Adviser-Andy Podgurski.

[22] J. Rilling and B. Karanth. A hybrid program slicing framework. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 12–23, Nov. 2001.

[23] A. Szegedi and T. Gyimóthy. Dynamic slicing of Java bytecode programs. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 35–44. IEEE Computer Society, Sept. 2005.

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[25] F. Umemori, K. Konda, R. Yokomori, and K. Inoue. Design and implementation of bytecode-based java slicing system. In *SCAM 2003: Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 108–117, Amsterdam, The Netherlands, September 2003.

[26] G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Notices*, 26(6):107–119, 1991.

[27] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, Mar. 1995.

[28] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 512–521, Edinburgh, United Kingdom, May 2004.

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

[30] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 502–511, Edinburgh, United Kingdom, May 2004.

[31] J. Zhao. Slicing concurrent Java programs. In *Proceedings of the Seventh IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 126–133, May 1999.