Union Slices for Program Maintenance

Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó and Tibor Gyimóthy
University of Szeged, Research Group on Artificial Intelligence
Aradi vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544126
{beszedes,farago,szabozs,gyimi}@cc.u-szeged.hu

Abstract

Owing to its relative simplicity and wide range of applications, static slices are specifically proposed for software maintenance and program understanding. Unfortunately, in many cases static slices are overly conservative and therefore too large to supply useful information to the software maintainer. Dynamic slicing methods can produce more precise results, but only for one test case. In this paper we introduce the concept of union slices *(the union of dynamic slices for many test cases) and suggest using a combination of static and union slices. This way the size of program parts that need to be investigated can be reduced by concentrating on the most important parts first. We performed a series of experiments with our experimental implementation on three medium size C programs. Our initial results suggest that union slices are in most cases far smaller than the static slices, and that the growth rate of union slices (by adding more test cases) significantly declines after several representative executions of the program.*

Keywords

Software maintenance, reverse engineering, program analysis, program slice, dynamic slicing

1 Introduction

Although one cannot say that they are the ultimate solution for industrial software maintenance and reengineering tasks, program slicing methods [17, 19] have been intensively studied in recent decades, and many impressive methods and applications have been proposed by researchers with various objectives in mind, including maintenance, reverse engineering, testing, debugging (see, for example, [2, 3, 16, 20]). A *backward slice* consists of all statements and predicates that might affect a set of variables at a specific program point (the slicing criterion) [19]. In contrast, *forward slices* are used to determine those parts of a program that could be affected by a change at a specific program point. A slice may be an executable program or a subset of the program code. Furthermore, a slicing algorithm can be classified according to whether it only uses statically available information (*static slicing*) or computes those statements which influence the value of a variable occurrence for a specific program input (*dynamic slicing*).

Owing to its relative simplicity and wide range of applicability, static slices have been specifically proposed for maintenance and program understanding [6, 8, 11, 16]. By computing various dependences concerning the flow of data and control, one is able to use static slices to observe only parts of the program that may be relevant from one specific point of view. Unfortunately, in many cases the static slices are overly conservative and hence too large to supply useful information. It is not uncommon for a slicing criterion which contains a variable that has a big effect on other parts of the program to include almost the whole program! Using the CodeSurfer tool [9]—which is one of the few practical implementations of static slicing methods—we experienced the same problem. This is due to the nature of static slices: in order to be safe, all possible program points need to be involved (even if they will never be used in any case). There are many theoretical problems that lead to this inaccuracy, such as handling of unconstrained pointers (e.g. [15]) and unstructured control flow.

Dynamic slicing methods (e.g. [1, 13]) can produce a precise result for one test case, which can be quite useful for some applications such as debugging. On the other hand, in many other applications one test case is not enough to investigate, more global information may be needed about the program. Dynamic slices are simply *unsafe* if one is interested in more than one specific test case. This can be aided by computing different slices for different test cases, but to cover all possibilities is difficult, if not impossible. Other problems arise of practical nature, such as the production of the necessary dynamic data and the fact that the computation may require large memory space (cf. recent methods such as [1]).

In this paper we propose a compromise solution for program maintenance tasks where the static slices are unacceptable because of their lack of precision and dynamic methods are unfeasible because of the lack of resources needed to conduct lots of test cases. We introduce the notion of *union slices* for the computation of the union of dynamic slices for many test cases. It is based on previous results where we introduced an efficient method for the computation of dynamic slices in real life situations [4]. We recommend the use of a combination of static and union slices to determine the responsible program parts with less effort. This means that the size of program parts that need to be investigated can be reduced by concentrating on the most important parts first. The basic idea for this comes from the fact that while the static slices are safe but large, union slices are smaller but, alas, unsafe (i.e. they do not contain all possible dependences).

If we consider the static slice as an upper bound of the precise slice,[1] the union slice can be seen on the other hand, as the *lower bound* for it (see Figure 2). The most apparent advantage of the combined application of the static and union slices is when they coincide, because in this case the precise slice is surely found. However, according to our experiences, this is probable to occur only in case of small programs (at most few hundred lines of code). Even with medium size programs (as those in our experiments) the union slices are significantly smaller than the static slices (10-50%). Nevertheless, in these cases the combined use of static and union slices can still be of great help for solving a software maintenance or testing problem.

Generally, the use of different slicing methods is aimed at finding a portion of program that needs to be investigated in order to solve a specific problem (e.g. to determine how a change at a certain program point affects the remaining parts of the program). Despite the fact that the static slices provide smaller set of data to be investigated by the software maintainer than the whole program, this reduction will be too small to provide real help because of the limited resources to perform a maintenance problem. In almost every practical situation the limited resources 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 dependences? Due of this, it can easily happen that the resources are wasted on the investigation of such parts of the program that realy do not carry realizable dependences. The application of the union slices tacles exactly this problem by explicitly pointing to those parts of the program that need to be investigated, because they became parts of the slice implicated by a certain test case. This way, the resources for the maintenance task can be used effectively. Of course, there can be some situations where fully safe solutions are inevitable for a certain problem. In these cases the investigation of the program parts representing the difference between the static and the union slices need to be performed as well.

To investigate how union slices are related to the corresponding static slices and to monitor the growing tendencies of the union of dynamic slices, we implemented our algorithms in a tool for the computation of union slices for real C programs. In order to compare static slices and union slices, we performed various experiments for three medium size programs. We defined several slicing criteria using a classification of the slice variables and performed a variety of executions for these programs. Our initial results suggest that union slices are in most cases far smaller than static slices, and what is even more important is that the growth rate of the union slices (by adding more test cases) significantly declines after several representative executions of the program under investigation. Hence, we are certain that the use of the combination of static and union slices can be useful in software maintenance and program understanding.

The paper is organized as follows. In the next section we introduce the basic conceptual algorithm for the computation of dynamic slices and we formulate the method for computing the union slices. Our implementation is also described in this section. In Section 3 we present and comment on the actual results of our experiments. Section 4 discusses connections with other work, then in Section 5, we draw conclusions from our results and outline directions for future research.

2 Forward computation of union slices

When using the different slicing methods one always seeks to reduce the portion of the program that needs to be investigated. As far as backward slices are concerned, they are particularly useful in determining those parts of the program that may contain program faults, and which are responsible for incorrect behavior at a certain program point (the slicing criterion). These slices of the program can be made significantly more precise (i.e. smaller) assuming that the failure has been revealed for a given input. In fact, this was the motivation in the first place for the introduction of the dynamic slices. The difference between static and dynamic slices is well illustrated by the example in Figure 1. It can readily be seen that while the static slice covers the whole program, the dynamic slice does not take into account those instructions which even have not been executed

[1]The notion of the precise slice is used for a slice that contains only those parts of a program that are really responsible in *some of all possible executions*. This slice exists only theoretically because producing all possible executions is generally unfeasible.

```
    #include <stdio.h>
    int n, a, i, s;
    void main()
    {
1.    scanf("%d", &n);
2.    scanf("%d", &a);
3.    i = 1;
4.    s = 1;
5.    if (a > 0)
6.      s = 0;
7.    while (i <= n) {
8.      if (a > 0)
9.        s += 2;
        else
10.       s *= 2;
11.     i++;
      }
12.   printf("%d", s);
    }
```

(a)

```
    #include <stdio.h>
    int n, a, i, s;
    void main()
    {
1.    scanf("%d", &n);
2.    scanf("%d", &a);
3.    i = 1;
4.    s = 1;
5.    if (a > 0)
6.      s = 0;
7.    while (i <= n) {
8.      if (a > 0)
9.        s += 2;
        else
10.       s *= 2;
11.     i++;
      }
12.   printf("%d", s);
    }
```

(b)

Figure 1. (a) The static slice and (b) the dynamic slice for $(\langle \mathtt{a} = 0, \mathtt{n} = 2\rangle, 12, \{\mathtt{s}\})$

(e.g. 6 and 9). Therefore, in this respect the dynamic slice is more precise.

Before we describe the algorithm some basic definitions need to be reviewed. We rely on [4] and [13] with some minor modifications.

A feasible path that has actually been executed will be referred to as an *execution history* and denoted by EH. The execution history contains the instructions in the same order as they have been executed. The execution history is basically a list of *actions*, which are denoted by i^j, where i is the serial number of the instruction executed at the jth step (the *execution position*). The *trace of the execution* contains the execution history supplemented with some other information about the runtime behavior of the program, such as the usage of memory locations. The trace will be needed by our algorithm for slicing real C programs.

Next, we can define the *dynamic slicing criterion* as a triple $(\mathbf{x}, i^j, V)$ where $\mathbf{x}$ denotes the input, i^j is an action in the execution history, and V is the set of the variables for which the dynamic dependences should be computed. The notation $(\mathbf{x}, i, V)$ will be used where the execution position has no relevance, e.g. for static slices.

Since the slicing criterion for a dynamic slice is based on a given input and an action, the resulting slice is as precise as possible for that specific execution of the program. However, if we take into account more executions of the program, we can approximate that kind of a program slice which can influence the criterion point regardless of the input. This slice is often referred to as the *precise slice*, because for all possible executions it contains precisely the statements responsible. The precise slice is at most as large as the static one, but in most cases it is far more precise. Unfortunately, the computation of the precise slice is very difficult, if not impossible. Of course, the goodness of this approximation primarily depends on how large portion of all possible inputs was successfully covered.

Our approach to the approximation of the precise slice is to have these different dynamic slices merged together, that is to compute their *union*. By doing this, we can give a lower bound for the precise slice, while—as illustrated with Figure 2—the static slice can still serve as an upper bound for it.

More formally, we define the union slices as follows.

2.1 Dynamic slices

We begin with the definition of a dynamic slice. A dynamic slice ($DynSlice$) of the program with respect to the slicing criterion $(\mathbf{x}, i^j, V)$ contains those statements that influenced the values of the variables in V at the specific action i^j (which might possibly be the last occurrence before the program exited, in which case we use the notation $(\mathbf{x}, i, V)$). Note, that to get the dynamic slice for a specific

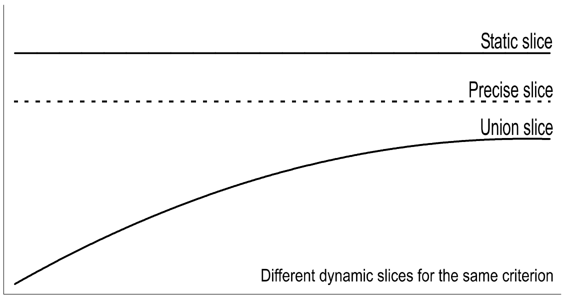


Figure 2. Approximation of the precise slice

execution all the previous dynamic slices for $i^1 \dots i^{j-1}$ are unnecessary. The most obvious application of the backward dynamic slice is in debugging.

Our algorithm for the forward computation of dynamic slices was first introduced in [10]. In [4] we showed that, unlike previous solutions, our method can be applied to real-size C programs because it is very efficient in terms of its memory requirements. In the current paper we present this algorithm in a simplified form. For clarity, we give only the basic conceptual algorithm for simple statements. However, the presented algorithm needs to be extended in various ways in order to be applicable to real C programs. The details of this can be found in [4, 5, 7].

The computation of the dynamic slices is done in three steps: (1) analyze the local static dependences (caused by one statement) in the program and instrument the source code, (2) execute the instrumented code to get the execution trace and (3) apply the forward global algorithm to compute the dynamic slices.

To compute the static dependences among program elements, we apply a program representation which considers only the *definition* and the *use* of variables and, in addition, it considers direct control dependences. We refer to this program representation as the D/U *program representation*. The D/U representation is the only statical data that is needed by the algorithm. An instruction of a program has a D/U expression as follows:

$$i.\, d : U,$$

where i is the serial number of the instruction and d is the variable that gets a new value at the instruction in the case of assignment statements. For an output statement or a predicate d denotes a newly generated "output variable"– or "predicate variable"–name of this output or predicate, respectively. Let $U = \{u_1, u_2, ..., u_n\}$ such that any $u_k \in U$ is either a variable that is used at i or a predicate-variable from which the instruction i is (directly) control dependent. Note that there is at most one predicate-variable in each U. (If an *entry* statement is defined, there is exactly one predicate-variable in each U.) Using the predicate variables a very simple algorithm can be constructed, because the control dependences can be handled the same way as the data dependences (see Figure 3).

Using the D/U representation and the execution history, the dynamic slices are computed as follows. We sequentially process each instruction in the execution history starting from the first one. Processing an instruction $i.\, d : U$, we derive a set $DynDep(d)$ that contains all the statements which affect d when instruction i has been executed. This set is computed based on the used variables' current dependences. Applying the D/U program representation the effect of data and control dependences can be treated in the *same way*. After an instruction has been executed and the related $DynDep$ set has been derived we determine the *last definition* (serial number of the instruction) for the newly assigned variable d denoted by $LS(d)$. Put simply, the last definition of variable d is the serial number of the instruction where d was defined last (considering the instruction $i^j.\, d : U$, $LS(d) = i$). Obviously, after processing the instruction $i^j.\, d : U$ (at the execution position j) $LS(d)$ will be i for each subsequent execution until d is defined next time. We also use $LS(p)$ for predicates, which means the last definition (evaluation) of predicate p. This computation order is strict, since when we determine $DynDep(d)$ we have to remember $LS(d)$ occurred at a former execution position rather than at the jth step.

Assuming that we are running a program P on input $\mathbf{x}$, after the execution of $i^j.\, d : U$, $DynDep(d)$ will contain exactly the statements involved in the dynamic slice for the slicing criterion $C = (\mathbf{x}, i^j, U)$, i.e. $DynSlice(C) = DynDep(d)$. The formal algorithm for the computation of all dynamic slices for a program P with input $\mathbf{x}$ is shown in Figure 3. Note that the algorithm computes the dynamic slices *globally* for all occurrences of every used variable at all of the instructions, but for the union slices only the last one is needed. Therefore, we also define $DynSlice(\mathbf{x}, i, V)$ (without the execution position index j) as the dynamic slice for the last occurrence of instruction i in the execution trace with input $\mathbf{x}$.

Examples of the computation can be found in e.g. [4].

2.2 The Union slice

Now we can formally define the union slices as the union of the dynamic slices for a variety of inputs. It is the following for a particular program P with different executions using the inputs $X = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$:

$$UnionSlice(X, i, V) = \bigcup_{\mathbf{x}_k \in X} DynSlice(\mathbf{x}_k, i, V)$$

Here, that specific $DynSlice$ set is used which holds the dynamic slice for the last occurrence of instruction i in the execution trace.

program DynamicSlicer($P, \mathbf{x}$)
inputs: EH for program P with input $\mathbf{x}$
D/U representation for P
outputs: $DynSlice(\mathbf{x}, i^j, V)$ sets for all i^j actions with the corresponding V sets as the used variables at i
begin
Initialize LS and $DynDep$ sets for all variables
for $j = 1$ **to** *number of elements in* EH
the current D/U element is $i^j.\ d : U$
$DynDep(d) = \bigcup_{u_k \in U} \big(DynDep(u_k) \cup \{LS(u_k)\} \big)$
$LS(d) = i$
Output $DynSlice(\mathbf{x}, i^j, U) = DynDep(d)$
endfor
end

Figure 3. Our global forward algorithm

In Figure 4 a union slice is shown with the same criterion as in Figure 1, but with one more added dynamic slice for $(\langle \mathtt{a} = 1, \mathtt{n} = 0 \rangle, 12, \{\mathtt{s}\})$. This dynamic slice contains the instructions 2, 5, 6 and 12, so the union includes the instructions 5 and 6. However, the static slice is still not attained.

2.3 Implementation

We implemented the above described algorithm for the computation of union slices for the C language. Note, that this description of the algorithm reflects only the basic principle; there are several details that needed to be solved for a real programming language like C. We will not elaborate on these details here as they can be found elsewhere (e.g. [4, 7]).

The implementation consists of four tools for the computation, visualization and comparison of the various types of slices, namely: a static analyzer, a dynamic slicer, a unionizer and a visualizer (see Figure 5, where the overall setup for our experiments is shown with the components of our tool in the bold boxes).

The dynamic slices are generated for the subject program with one execution for a set of slicing criteria simultaneously, because our algorithm is global. Additional dynamic slices are generated for all of the executions with different inputs. To get these slices two tools are needed: (1) the simple static analyzer, which produces the static D/U dependences and instruments the source code, and (2) the implementation of the dynamic slice algorithm, which computes the slices based on the execution trace and the D/U dependences computed previously. The execution trace is gained by instrumenting the (preprocessed) source code with write-out instructions (these are implemented in C++).

```
     #include <stdio.h>
     int n, a, i, s;
     void main()
     {
 1.    scanf("%d", &n);
 2.    scanf("%d", &a);
 3.    i = 1;
 4.    s = 1;
 5.    if (a > 0)
 6.      s = 0;
 7.    while (i <= n) {
 8.      if (a > 0)
 9.        s += 2;
         else
10.        s *= 2;
11.      i++;
       }
12.    printf("%d", s);
     }
```

Figure 4. A union slice

In the next step the resultant slices are processed by the unionizer tool to produce the corresponding union slices, and some statistics as well. Afterwards, the visualizer tool can be used to display all kinds of slices in a synchronized way (i.e. the same source can be displayed in several windows, the corresponding lines being indicated with different colors). The visualizer is also capable of displaying other kinds of program slices such as the static slice[2] and the execution trace itself. A snapshot of the visualizer can be seen in Figure 6. It can be seen that the trace, the static slice and the union slices are all displayed in parallel, thus allowing their simultaneous examination and analysis.

3 Comparison of static and dynamic slices

As mentioned earlier, a major goal of this work was to obtain empirical data on the relationships between the conventional static slices and the union slices proposed by this article using the same slicing criteria. Experimental demonstration was aimed of the fact that the static slices are in many cases overly conservative, and that suitably obtained union slices are more precise so a clever combination of the two is preferable in many cases.

[2]Static slices were computed using the external tool CodeSurfer [9].

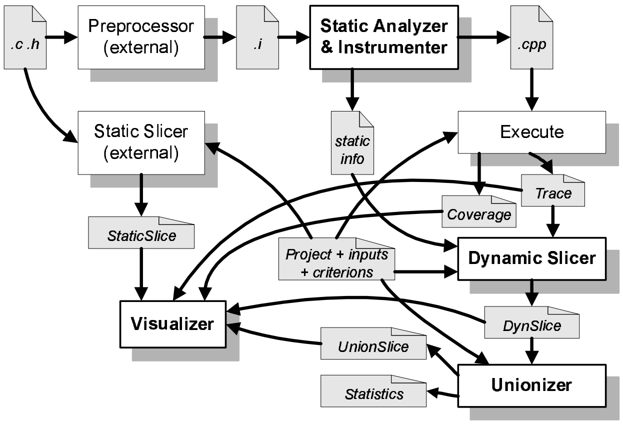


Figure 5. Experimental tool setup

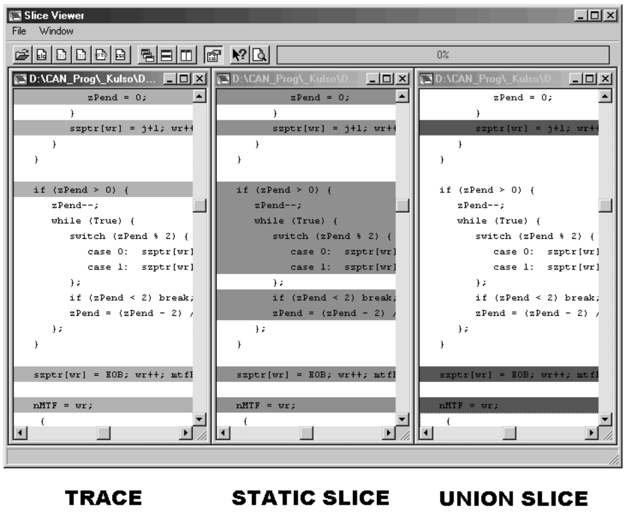


Figure 6. The slice visualizer

3.1 The test bed

For our experiments we used three medium size C programs. In the table below some details can be found about these programs ("*bzip*" is a compression utility, "*bc*" is a scientific calculator and "*less*" is a powerful text viewer):

prog	lines	extble	files	bytes	func
bzip	4495	1595	1	130 458	73
bc	11555	3220	20	312 722	138
less	21489	5400	43	639 036	363

The first column shows the total number of program lines, while the second one gives the number of nonempty executable code lines (i.e. comments, declarations, etc. are not counted). These programs are large enough to demonstrate non-trivial slices, but are also modest in size to process by the experimental tools in reasonable time. As already known, the dynamic slices are produced from the execution traces that are gained by executing the instrumented source code. Note, that the instrumented code runs significantly slower, but this could be aided by instrumenting the object code instead of the source code. Our assumptions regarding the space and time requirements of the basic dynamic slice algorithm (Section 2.1) turned out to be generally true, with some deviations because of the special computations for the C language. These are, namely, that the algorithm time is proportional to the size of the execution trace (number of instructions executed) and that the memory requirements are proportional to the number of different memory locations used by the program under investigation.

In the test programs we selected several program points that could serve as interesting slicing criteria. We used 154 criteria for *bzip*, 57 for *bc* and 50 criteria for *less*. These program points were chosen in such a way that as much portion of the program could be covered by all our test slices altogether. Another consideration was to have the variables present in these criteria correspond to different categories in order to be able to investigate how does the kind of the criterion affect the slice size. Therefore, we applied certain labels to each of the variables (similarly to those proposed by [18]): *Local*, *Global*, *Pointer dereference*, *Function parameter*, *Array index*, *Loop index*, *Return* and *Output*.

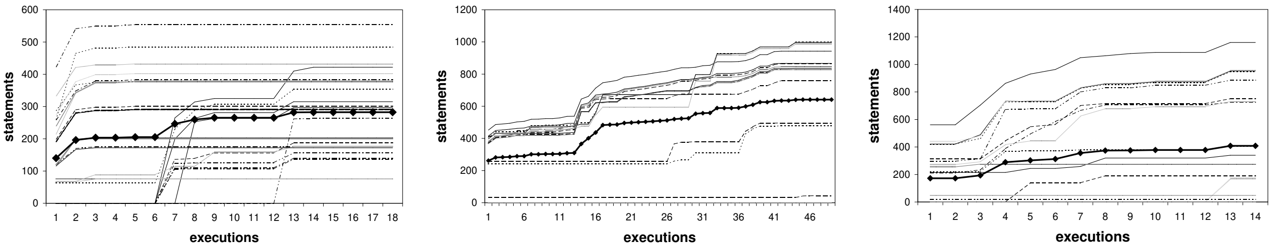
In order to compute the union slices, we need several different executions of the same program. Fortunately, since our algorithm is global, the slices for all of the criteria can be computed simultaneously and there is no need to execute the algorithm for each execution and criterion separately (see Section 2.1). We aimed at producing different inputs for the programs in order to have large program coverage[3] (*less* is an interactive program, so it needed to be handled differently). In the following table the statistics about the inputs and the coverage (with respect to the number of nonempty executable lines) can be observed:

program	criteria	executions	coverage
bzip	154	18	68%
bc	57	49	63%
less	50	14	45%

3.2 The results

For the experiments we computed the union slices for the three programs with the different inputs as described in Section 2. The data for all slicing criteria was computed simultaneously and stored in files. Recall, that our algorithm is global, meaning that the dynamic slices can be computed using only one pass through the execution history (see Figure 3).

[3]On program coverage we mean those instructions which have been executed in at least one of the test cases. Unfortunately, even full coverage cannot guarantee that the precise slice has been well approximated.



a) *bzip* b) *bc* c) *less*

Figure 7. Growth of the union slices

In our first experiment we recorded the union slices and their growth characteristics by combining the dynamic slices. In Figure 7 these characteristics can be observed for the three programs (the slice size is displayed in the number of instructions). On each diagram the growth of the union slices can be seen for several typical criteria (not all criteria are displayed for clarity) as the new dynamic slices are unioned for new test cases (executions of the program). The thick lines correspond to the average of all curves (not only for the displayed ones). It can be clearly seen that the growth slows down after only a few executions. Of course, it cannot be forseen how would additional executions change the union slices, but the general tendency of the approach towards the precise slice can be recognized as elaborated in Section 2.

In Figure 8 the average growth tendency for the three programs can be seen using the same input as above. In order to obtain this diagram we computed the simple average of growth curves of all of the slicing criteria separately for the three programs (the same as for the diagrams in the previous figure). These curves are displayed in a normalized form relative to the maximum attained slice size. The curves were also stretched horizontally, the full width depicting the total number of the executions. Interestingly, the overall characteristics for the three different programs are quite similar.

We also investigated the sizes of the final union slices compared to the program size and the static slices.[4] A general picture of the results is presented in Figure 9, where the total average values are shown. This figure shows the comparison of the program sizes (executable lines), the coverages (as in the tables above), the average static slice sizes and the average union slice sizes for the three programs displayed in lines of code. Actually, the average of the ratios of the coverage, the static slice and the union slice with respect to the program size were 59%, 72% and 15%, respectively.

More detailed data regarding the slice sizes is shown in Figures 10 and 11, where the distribution of the slice

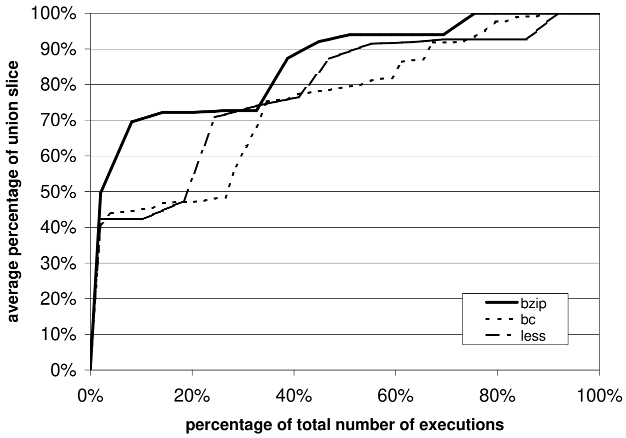
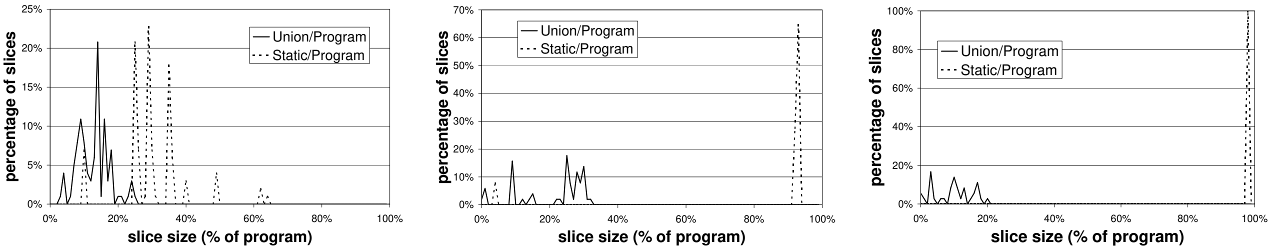


Figure 8. Average growth of the union slices

sizes is represented in the form of a histogram, counting the occurrences of specific percentage values. The sizes have been computed based on the number of program lines. The three diagrams of Figure 10 show the distributions of the static slice vs. program size (executable lines) and the union slice vs. program size (executable lines), while the ratios of union slices over static slices is shown in Figure 11. From Figure 10 we can see that for the three programs the static and union slice sizes show quite different characteristics, but generally, the union slices are much smaller. The exact difference can be seen in Figure 11: for *bzip* the union slices are about half of the static slices, while for the other two programs the difference is much greater. This is due to the fact that for *bc* and *less* the static slice is almost always nearly the whole program. On the other hand, the union slices are in all of the three cases around 20-30% of the program size.

Finally, we made some experiments to find out whether the kind of the variable present in the slicing criterion has any effect on the slice size (as investigated by [18] as well). In Figure 12 the distribution of union slice sizes is shown for two of the variable categories as defined at the beginning of this section (*Local* and *Global*). The data was generated based on the same results as for the previous diagrams for

[4] In these comparison experiments we used lines of code instead of instructions in order to be able to compare the results of our tool with the external static slicing tool.



a) *bzip* b) *bc* c) *less*

Figure 10. Distribution of slice sizes

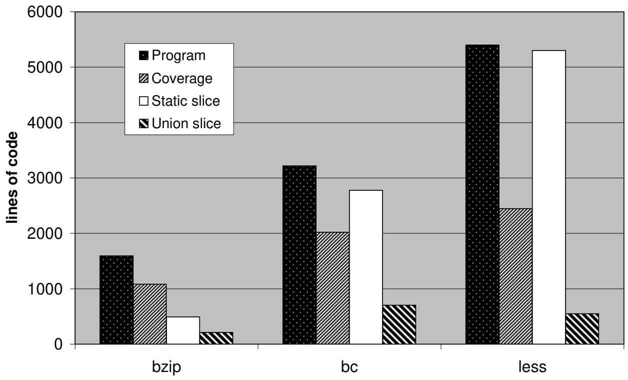


Figure 9. The average slice sizes

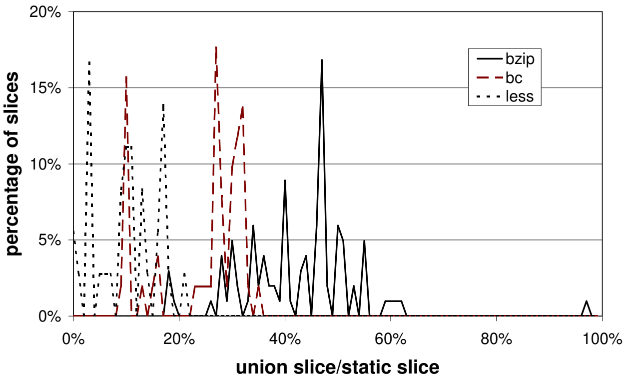


Figure 11. Distribution of union vs. static slice ratios

the program *bzip*. The other categories are not shown because there is not much difference regarding the distribution characteristics. Even in this case, the only observation is that in some cases the global variables produce slightly larger union slices.

3.3 Conclusions from the results

Generally, the results from the previous figures support our initial assumptions that we made in the beginning of this paper. The most important conclusion is probably that the union slices are much smaller than the static ones, although the test cases were more-or-less representative and covering pretty large parts of the programs (see Figures 9, 10 and 11). Moreover, this difference between the static and the union slices presumably will not be much smaller either by adding much more new test cases. We can come to this conclusion by observing the growth tendencies of the union slices in Figures 7 and 8.

As the main drawbacks of using the union slices we can mention the problem of managing to obtain many different test cases and the practical problems with the instrumentation and the slowdown as its side effect.

4 Related work

Our method for the computation of the dynamic slices significantly differs from previous approaches. Agrawal and Horgan's algorithm [1] uses a large internal representation called the Dynamic Dependence Graph (DDG), whose size may be unbounded. We showed in [4] that our algorithm is much more efficient in terms of memory requirements and therefore it can be applied to real size programs. In [14] Korel and Yalamanchili introduced a forward method for determining (executable) dynamic program slices. In many cases these slices are less accurate than those computed by our forward dynamic slicing algorithm. (Executable dynamic slices may produce inaccurate results in the presence of loops [17].) An excellent comparison of various dynamic slicing methods can be found, for example in [12] and [17].

To our knowledge, there is no previous work that deals with the comparison of static and dynamic slices, in a similar way as we do in this paper. In fact, the dynamic slices are undoubtedly much smaller than the static slices, but our union slices can be more comparable to the static ones.

As far as the practical implementations of the different slicing methods are concerned, the situation is not so good.

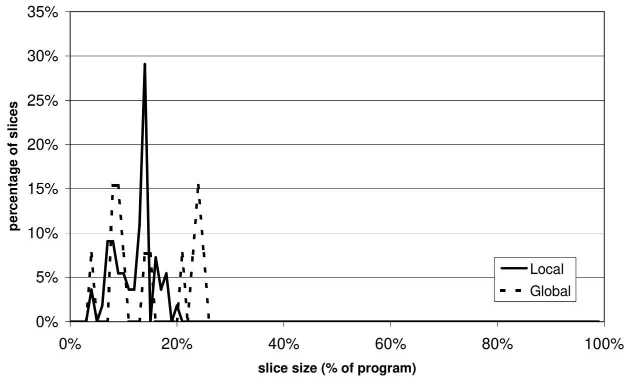


Figure 12. Union slices of two variable categories

First, many papers exist on static slices but only one commercial product is available (CodeSurfer [9]), although several prototype tools have been introduced. Further, the dynamic slicing methods are only elaborated in few papers and we are not aware of a really used dynamic slicing tool.

Venkatesh's paper [18] is the only publication that deals with the evaluation of dynamic slices, however no comparison was made to static slices in this article either. 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. An experimental prototype tool was used to perform the measurements. One of the basic observations in this work was that the slices generally show a bimodal distribution clustering at the two ends of the range of slice sizes. We used a similar approach to display the distribution of the union slice sizes (the difference is that we compared the slice size to the executable code lines while Venkatesh made this comparison to the number of executed instructions). We also observed the same characteristic for the union slice distribution of program *bc* (Figure 10b).

5 Conclusion and future work

Static slicing methods have been proposed for maintenance and program understanding because this way certain parts of the program can be "sliced away" that are of no interest with respect to the slicing criterion. However, because of their inherent static nature, the static slices are in many cases overly conservative, which means that often too large portions of the program under investigation become parts of the slice, so no really useful information can be given. While remaining safe, only the *precise slices* could produce smaller slices, but unfortunately the computation of the precise slice is in most cases unfeasible because it would require to investigate all possible executions of the program under investigation.

In this article we follow a different approach: the precise slice is approximated "from below" (see Figure 2) by computing the dynamic slices for many executions and combining the result. A *union slice* for a particular slicing criterion is computed by unioning the dynamic slices for different executions of the program. As further explained in the introduction of this paper, we also suggest using a *combination* of the static and union slices for software maintenance. The software maintainer could use her resources for the maintenance task more effectively, if the (large) static slice is not investigated completely, but only those parts of the program should be examined that became parts of the slice implicated by a certain set of test cases (i.e. only the realizable dependences are considered). The union slices provide exactly these parts. In the introduction of this paper we elaborate more on this.

As a concrete application we could imagine the approximation of decomposition slices [8] with the union of union slices for all occurrences of a variable under investigation. These decomposition slices are useful in making a change to a piece of software without unwanted side effects. These are based on static slices and therefore, the above mentioned impreciseness could be aided by the use of union slices. In this case also, the *union decomposition slices* could be used first to determine the crucial part of the program for critical test cases.

Another application can be for *regression testing* [2] to reduce the cost of testing all those parts of the program that the static slices provided. Instead, the union slices could be used to determine those parts of the program that need to be tested *based on a certain set of critical test cases* first.

Program understanding based on slicing techniques [6] can also benefit from the use of union slices because the location of the safety critical code can be determined with less effort using the union slices by computing them for the most important test cases first.

We already implemented our approach for the computation of dynamic and union slices using our efficient algorithm. The implementation is capable of handling real size C programs. For the experiments we used three medium size C programs and computed the union slices for different slicing criteria and several representative executions of the programs. Our initial results show that the growth of the union slices by adding new test cases significantly slows down after only a few representative executions and this suggest that the union slices will not be too far from the precise slice, while still being far smaller than the static ones.

In the future we plan to make a more robust implementation of the algorithms and to perform some measurements on real, industrial applications with real case studies. For this purpose we need to make the implementation more stabile and improve the performance, especially regarding the slowdown of the instrumented code. Another ongoing work is to implement the algorithm for the C++ language. However, this involves a number of technical issues such as the more complicated handling of expressions. Fortunately, the handling of *polymorphic calls* will not be a problem, as it generally is for static slicing methods for object oriented languages. This is because the static dependences can be computed in a pessimistic way, and the concrete binding of the called method will be known anyway based on the execution trace.

Acknowledgements

The authors wish to acknowledge their gratitude to GrammaTech, Inc. for supplying an academic license for the CodeSurfer static slicing tool. We also wish to thank the work of Hunor Pető for his help during performing the experiments.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *SIGPLAN Notices No. 6*, pages 246–256, 1990.

[2] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London. Incremental regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, Montreal, Canada, 1993.

[3] J. Beck. Program and interface slicing for reverse engineering. In *Proceeding of the Fifteenth International Conference on Software Engineering*, 1993. Also in *Proceedings of the Working Conference on Reverse Engineering*.

[4] Á. Beszédes, T. Gergely, Z. 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] Á. Beszédes, T. Gergely, Z. M. Szabó, C. Faragó, and T. Gyimóthy. Forward computation of dynamic slices of C programs. Technical Report TR-2000-001, RGAI, 2000.

[6] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43, 1996. Marvin Zelkowitz, Editor, Academic Press San Diego, CA.

[7] Cs. Faragó and T. Gergely. Handling the unstructured statements in the forward dynamic slice algorithm. In *Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 71–83. University of Szeged, June 2001.

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

[9] Homepage of GrammaTech's CodeSurfer. `http://www.grammatech.com/products/codesurfer`

[10] T. Gyimóthy, A. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proceedings of ESEC/FSE'99*, number 1687 in Lecture Notes in Computer Science, pages 303–321. Springer-Verlag, Sept. 1999.

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

[12] M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Systems Software*, 31:197–214, 1995.

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

[14] 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.

[15] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation*, number 27(7) in SIGPLAN Notices, pages 235–248, San Francisco, 1992.

[16] G. Rothermer and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of ISSTA'94*, pages 169–183, Seattle, Washington, Aug. 1994.

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

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

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

[20] J. Zhao. A slicing-based approach to extracting reusable software architectures. In *Proc. 4th European Conference on Software Maintenance and Reengineering (CSMR'2000)*, pages 215–223, Feb. 2000.