Static Execute After/Before as a Replacement of Traditional Software Dependencies

Judit Jász, Árpád Beszédes and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 544145
{jasy,beszedes,gyimi}@inf.u-szeged.hu

Václav Rajlich
Wayne State University, Department of Computer Science
427 State Hall, Detroit, MI 48202, (313) 577-5423
rajlich@wayne.edu

Abstract

The paper explores Static Execute After (SEA) dependencies in the program and their dual Static Execute Before (SEB) dependencies. It empirically compares the SEA/SEB dependencies with the traditional dependencies that are computed by System Dependence Graph (SDG) and program slicers. In our case study we use about 30 subject programs that were previously used by other authors in empirical studies of program analysis. We report two main results. The computation of SEA/SEB is much less expensive and much more scalable than the computation of the SDG. At the same time, the precision declines only very slightly, by some 4% on average. In other words, the precision is comparable to that of the leading traditional algorithms, while intuitively a much larger difference would be expected. The paper then discusses whether based on these results the computation of the SDG should be replaced in some applications by the computation of the SEA/SEB.

Keywords

Source code analysis, program dependencies, System Dependence Graph, program slicing, Static Execute After.

1. Introduction

In order to understand and evolve software, programmers must understand not only the components that comprise the software, but also the dependencies among them. These dependencies play a key role during software change and they guide the activities of impact analysis, change propagation, regression testing, and so forth. The importance of understanding dependencies among program components has been illustrated many times; some of the most dramatic examples are the software-based catastrophes caused by misunderstood dependencies [10], [16].

Source code analysis is a discipline that started in compiler research but found many applications in software engineering [6]. One of the central topics for source code analysis is to find dependencies among program components. A classical result of source code analysis is the work on the *System Dependence Graph (SDG)* [15]; a range of sophisticated algorithms analyses the source code and produces the SDG that, among other information, contains dependencies between the modules. The SDG dependencies and their transitive closure can be retrieved from the source code with the techniques of classical program analysis and they are currently computed by leading commercial software tools. In this work, we define these dependencies and call them *Traditional Software Dependencies (TSD)*. They are a foundation of program slices, that are subsets of a program relevant to a specific computation [6, 22, 24].

While TSD are widely used in the current software tools, there are two major problems: their computation requires very expensive algorithms, and in some cases TSD may miss some important software dependencies [25], therefore there is a strong incentive to search for new algorithms and new techniques that would be simpler and at the same time to provide larger dependency set than TSD. *Static Execute After/Before (SEA/SEB)* is such a new algorithm; it determines the dependencies among program procedures by simply considering their execution order.

However, there is an immediate concern: when moving from TSD to SEA/SEB, are we going to give programmers a huge set of dependencies, most of them irrelevant? This paper reports a surprising answer to this question: when moving from TSD to SEA/SEB, the set of dependencies increases only very slightly, by some 4% on average, or in other words, the results are comparable to that of the leading TSD algorithms. Intuitively, a much larger difference would be expected. This raises a question: given such a small difference between TSD and SEA/SEB, and at the same time SEA/SEB being a simpler and more scalable algorithm with a chance of providing some additional important dependencies, would it be advisable to replace TSD by SEA/SEB in some applications?

The paper is organized as follows. Section 2 reviews related work, particularly TSD and its relation to SEA/SEB. The SEA/SEB are described in Section 3. In Section 4 we empirically investigate relation of SEA/SEB and TSD and we conclude in Section 5.

2. Previous work

2.1. TSD in imperative programs

In this work, we use TSD of imperative programs that are based on System Dependence Graphs [15]. The SDG construction includes calling context problem, the existence of arbitrary control flow, the use of pointers, and other related issues, and has been a challenge for the slicing community for decades [6], [22].

In this paper, we mainly deal with procedure level TSD (or procedure level slices); we illustrate such TSD through an example of four simple procedures, shown in Figure 1. Figure 2 shows the SDG of the program obtained by applying the algorithm of Horwitz *et al.* [15]; some simplifications have been made to improve readability. Parameter, call and summary edges represent procedure calls. A control dependency between two program points means that the execution at the dependent point is determined by the starting point, while a data dependency exists between a point where a variable value is defined and another point where it is used without redefinition [18].

Traversal of the SDG will yield procedure level TSD dependencies shown in Table 1. In this simple example, these dependencies are easily readable directly from the code of Figure 1.

2.2. Flow-based dependencies

There is a simple, yet very effective alternative to compute software dependencies. In *Interprocedural Control Flow Graph (ICFG)* [14, 17] dependencies can be computed using the graph reachability approach [5]. Figure 3 shows ICFG program representation for our example program. In it, we can easily verify that all dependencies captured by the SDG are captured by the ICFG as well. This replacement is safe, meaning that all dependencies captured by SDG will be captured by ICFG (as any data- and control-dependency implies a flow-based dependency),[1] but there will always be additional dependencies with this approach which do not correspond to any data or control dependency as computed by the SDG.

We are not aware of any other work that investigated this approach as thoroughly as we do in this paper. Orso *et al.* mention in their work on impact analysis [20] that they also employ reachability on the ICFG, however no further details are given. Our work was in part motivated by the works of Orso *et al.* [20] and Apiwattanapong *et al.* [1].

```
texts[] = {...}
procedure getIndex(out index){
  read(index);
}

procedure printParam(in index){
  write(texts[index]);
}

procedure printLast(){
  write(texts[texts.length]);
}

procedure main(){
  i = 0;
  while (i >= texts.length)
    getIndex(i);
  if (i != 0)
    printParam(i);
  printLast();
}
```

Figure 1. Example program

Procedure	`printLast` dependent on	`printParam` dependent on
`main`	√	√
`getIndex`		√
`printLast`	[√]	
`printParam`		[√]

Table 1. Static dependence sets of the example (self-dependency may be considered in some situations)

[1] If TSD is determined in a flow-insensitive manner then there may be some (false) TSD dependencies not covered by the ICFG-based approach.

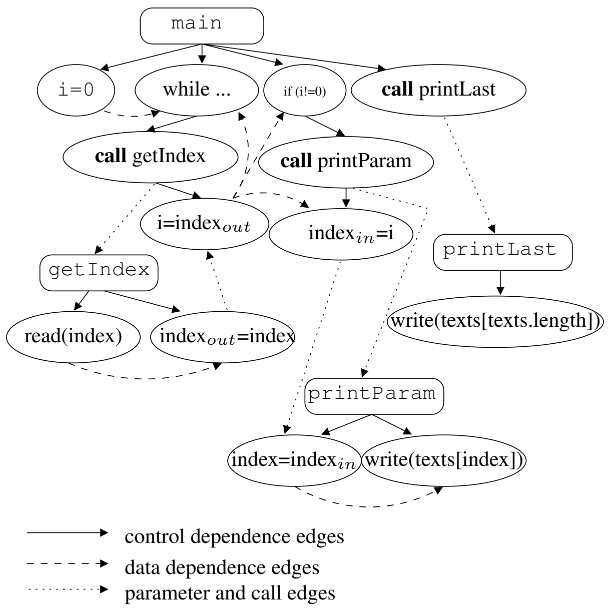

Figure 2. The SDG of the example

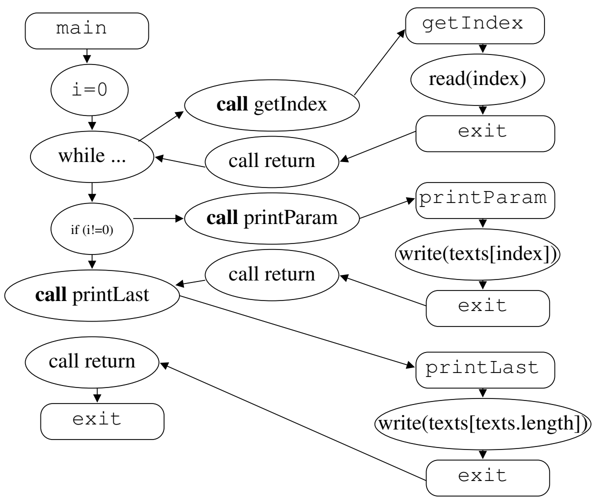

Figure 3. The ICFG of the example

In [5], we proposed SEA relations as a technique to discover hidden dependencies, i. e. software dependencies that are not directly captured by TSD, hence we concentrated on the dependencies that are SEA but are not TSD. In the present paper we present an evidence that there actually may be a small difference between SEA/SEB and TSD. We conduct an extensive case study about the relationships between TSD and SEA/SEB and provide a description of a practical tool support for the approach. Hence this paper complements and completes the work presented in [5].

2.3. Other relevant work

Since computation costs and the space requirements of the SDG are significant, especially in the case of large software systems, there have been attempts to cut down these costs. In the approach of Atkinson and Griswold [2], although there can be a-priori determined representations such as the callers of a procedure, other easily computable representations are computed or recomputed on demand. Also at instruction level, Tonella *et al.* presented a variable precision algorithm to determine reachability for program understanding [23].

Another possibility for reducing the costs is the reduction of the program representation, as we do in our approach. Badri *et al.* [3] use the so-called control call graph in which nodes which do not influence the execution of the procedure calls are left out. This approach is similar to ours in that it also considers procedure level dependencies for impact analysis purposes. However, they use a different algorithm for computing the dependencies and present no findings about the efficiency and precision of the method.

The *Context Sensitive Control Flow Graph (CSCFG)* introduced by Ng is used for visualization purposes [19]. In this approach the details of unimportant parts of the intraprocedural control flow graph are collapsed into single nodes in order to simplify the graph. Yu and Rajlich used the *Abstract System Dependence Graph (ASDG)* to compute hidden dependencies [25], which is based on the detailed SDG and represents both data and control dependencies.

Binkley *et al.* investigated the size of static program slice in their large scale empirical study [7]. Among other issues, the authors compared the size of statement-level slices with the size of function-level slices. They found that while function-level slices are 33% larger than corresponding statement-level slices, they may be useful predictors of the statement-level slice size.

3. The Static Execute After/Before relationships

In this section we define the notion of *Static Execute After (SEA)* and its counterpart *Static Execute Before (SEB)*, which will be used to formulate more accurately the flow-based dependencies mentioned above. We say that $(f, g) \in SEA$ if and only if it is possible that any part of g is executed after any part of f. Similarly, procedures f and g are in SEB relation if and only if, it is possible that any part of g is executed before any part of f. It can be observed that these relations are inverse to each other just as is the case with backward and forward slices. A dynamic counterpart of this relation has been defined previously by Apiwattanapong *et al.* [1]. The dynamic Execute After relation is based on the analysis of execution traces and not static program representations.

Following the notation of Apiwattanapong *et al.* [1] and Beszédes *et al.* [4] we may formally define the SEA relation involving (f, g) as follows:

$$SEA = CALL \cup RET \cup SEQ\,[\cup\, ID],$$

where

$$\begin{aligned}
(f, g) \in CALL &\iff f \text{ (transitively) calls } g,\\
(f, g) \in RET &\iff f \text{ (transitively) returns into } g,\\
(f, g) \in SEQ &\iff \exists h : h \text{ (transitively) calls } f \text{ first,}\\
&\quad \text{then } h \text{ (transitively) calls } g\text{, and}\\
&\quad \text{the second call site is flow-reachable}\\
&\quad \text{from the first one.}
\end{aligned}$$

Here, ID is used to denote the identity relation that can optionally be part of SEA/SEB since some notions of traditional dependencies are also reflexive. The SEB relation is defined as an inverse of SEA.

In order to compute SEA/SEB, the traditional call graph representation [21] is not sufficient since it says nothing about the order of the procedure calls within a procedure body. It is easily readable from Figure 3 that the dependency between `getIndex` and `printParam` will be missed when using the call graph. On the other hand, the ICFG of the previous section contains too much information, which is not necessary for deriving flow-based dependencies between procedures.

Hence, we propose a specialized program representation called the intra- and interprocedural *Component Control Flow Graph* (*CCFG* and *ICCFG*, respectively). Each CCFG represents a procedure's intraprocedural CFG but only call site nodes and corresponding flow edges are considered. It contains one *entry* node and several *component* nodes which are connected by control flow edges. Component nodes are obtained by collapsing strongly connected subgraphs into single nodes.[2] The ICCFG consists of CCFG of each procedure and in addition it includes call edges from each call site (a component) to the entry nodes of the called procedures.

Figure 4 shows the ICCFG graph of our example program. The nodes with the procedure names represent procedure entry nodes, while the darkly filled nodes correspond to the components. These are connected by control flow (solid) and call edges (dotted). Procedures `getIndex`, `printLast` and `printParam` are represented only by their entry nodes as these procedures do not have any call sites. We can easily see that this program representation is suitable for deriving SEA and SEB relations: we just need to traverse the flow and call edges in the respective direction. For example, we can follow that `printParam` may be executed after the procedures `main`, `getIndex` (and `printParam` itself), while `printLast` may be executed after all the procedures of the program including itself. Note, that there are two false dependencies with `printLast`, the procedures `getIndex` and `printParam`, which are the consequence of the imprecision of the approach.

[2]For technical reasons, if the call sites in a component node are part of a loop the component will have a reflexive control flow edge.

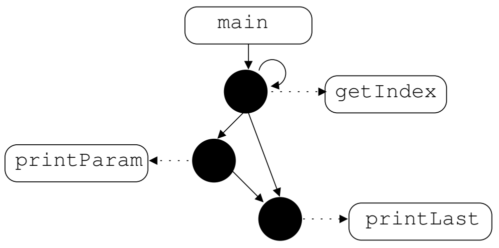

Figure 4. The ICCFG of the example

3.1 Building ICCFG and computing SEA/SEB dependencies

To compute the ICCFG we start from the ICFG of the program, which can be obtained using traditional compiler algorithms and which is available in many source code analysis front ends. The computational complexity of determining the strongly connected components in the ICCFG is $O(n + e)$ where n is the number of basic blocks and e is the number of the control and call edges among these nodes [9].

For computing a dependency set for a particular procedure, a reachability algorithm can be used, similar to the SDG reachability algorithm. An example of such algorithm for computing SEB relations is provided in Figure 5. The algorithm first traverses the ICCFG in backward direction and colors certain nodes that will be part of the dependency set as white and some other nodes that need to be further investigated as grey. Then in the second pass it iterates through the dependencies in forward direction coloring them black, and in this way it completes the dependency set.

A similar algorithm can be constructed for computing SEA relations as well.

Based on the algorithm in Figure 5, we designed a more complex but optimized algorithm which reuses already completed dependencies and produces the whole SEA/SEB relation globally, which is more suitable for our empirical investigations. The details of this algorithm can be found in our earlier work [5].

4. Empirical study

The first hypothesis of our empirical study states that TSD can be approximated to a sufficient extent by

```
program   computeSEB$(P, f)$
input:    $P$ : ICCFG of program
          $f$ : a procedure in $P$
output:   $S$ : set of procedures that are in SEB
                relation with $f$
begin
Empty $S$
Mark procedure entry nodes of $P$ uncolored
Color entry node of $f$ to grey
Traverse $P$ from entry node of $f$ in backward direction
    If component $c$ is reached from another component
    (not from an entry) then color the entry nodes of pro-
    cedures called by $c$ to grey
    If entry $e$ is reached by the traversal and it is not grey
    color it to white
    During the traversal each edge may be touched at
    most once
While there are grey entry nodes
    Let $e$ be a grey entry node
    Color $e$ to black
    Color the uncolored and white entry nodes of proce-
    dures called by components of $e$ to grey
Insert all colored procedures into $S$
Output $S$
end
```

Figure 5. Computation of SEB relation by graph reachability

SEA/SEB. The second hypothesis states that computation of SEA/SEB is significantly more efficient than computation of TSD, which makes it applicable to larger programs.

4.1. Design

To prove our hypotheses, we set off to perform practical experiments with a significant number of measurement data. We decided to treat the procedure level TSD as the "golden standard" and investigate the precision of SEA/SEB, i. e. the amount of false additional dependencies identified by SEA/SEB which do not correspond to any TSD.

Choosing procedure level granularity was natural for several reasons. We deal with large scale software sometimes comprised of millions of lines of code and statement level analysis would be too fine and cluttered. Also, from performance point of view procedure level analysis is more beneficial while still having the ability to predict statement level analysis [7]. We perform our case study for C programs where coarse granularity (files, for instance) would be too abstract. Finally, in many applications of dependency analysis, like in change impact analysis, this kind of granularity is typical. Nevertheless, extending the approach to other granularities, like classes in object oriented programs, would be possible; in fact, we already implemented a class-level analysis in our previous work [5].

To check the precision of SEA/SEB we compute a large number of dependencies for medium size subject programs and compare different sizes and size distributions gained for TSD and SEA/SEB sets. The efficiency of the algorithms is verified on several large software systems in order to find the limits of the different approaches in terms of space and time costs.

An additional goal was to verify that the recall is 100% in every case, meaning that SEA/SEB does not produce false negatives with respect to TSD. (Note, that recall less than 100% is conceivable if flow-insensitive TSD algorithms are used.)

We wanted to minimize internal validity threats so we designed our experimental architecture so that the two different approaches share as many common parts as possible.

4.2. Tools used

One of the most popular program slicing tools is CodeSurfer of Grammatech Inc. [13]. Various program analyses can be performed using it on C/C++/Ada programs, and CodeSurfer has been reported as one of the most accurate, robust and efficient program slicers available. Furthermore, it includes a highly usable Application Programming Interface with which different program analyses can be performed by plug-ins. We decided to use CodeSurfer as a basis for our tool architecture, which is supplemented by our own experimental algorithm implementations. We used in our experiment the version 2.1p1 of CodeSurfer.

It turned out that the Codesurfer API is appropriate for our SEA/SEB plug-in. It is important to note, though, that our method is language independent and requires only an ICFG to be available.

4.3. Architecture

The CodeSurfer API provides sufficient functionality for extracting all the necessary data to compare the TSD and the SEA/SEB relations. Therefore our experimental plug-in tool supports two separate dependency computation parts for TSD and SEA/SEB sets (see Figure 6). CodeSurfer is used as the common front end which performs source code

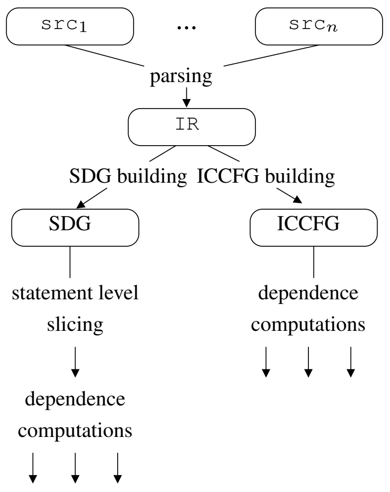

Figure 6. Experimental tool architecture

parsing and produces the common internal representation (IR), which may be slightly different in the case of the two dependence computation parts. After this the computation separates into two different program representations, SDG for program slices and the procedure level TSD, and ICCFG for SEA/SEB relations.

We used different presets of the front end for the two methods in order to gain a more optimal performance for each of the approaches (see Table 2). These presets are different in the SEA/SEB computation and in slicing, because the SEA/SEB computation requires less information and also uses less expensive computations than TSD. Namely, for SEA/SEB we generate the ICCFG graph on the basis of the call and the control flow information among basic blocks, while TSD requires that the SDG graph contains all the necessary dependence edges, including the control, the data and the summary edges as well. If any of the data- or control-depedencies are not taken into account or if call relations are incomplete (due to, for example, inappropriate handling of function pointers) then the computed TSD would be incomplete as well.

Preset of CodeSurfer	SDG	ICCFG
-control-dependence	yes	no
-data-dependence	yes	no
-compute-gmod	yes	no
-compute-summaries	yes	no
-cfg-edges	no	yes, both directions
-basic-blocks	no	yes

Table 2. Different presets of CodeSurfer used

4.4. Subject programs

For experiments, we started with the suite of C programs of Binkley and Harman [8], but in some cases we used different versions and also added several new programs; in some cases the version of the program was unknown. Table 3 lists the subject programs programs and their number of procedures and lines of code (TL means the total lines in the project while LCode means the number of non blank lines). For the experiments with the efficiency of the techniques we use several large C/C++ software systems available as open source. The basic features of these systems are listed in Table 4.

Program	Number of procedures	TL	LCode
time v1.7	12	1 314	757
replace v?	21	563	512
compress v?	24	1 937	1 335
wdiff v0.5	27	1 862	1 080
which v2.17	28	1 989	1 246
acct v6.3	50	3 510	1 996
termutils v2.0	57	3 685	2 518
barcode v0.98	62	3 885	2 331
indent v2.29	111	11 539	7 582
ed v0.8	120	3 052	2 267
EPWIC v?	149	9 597	5 249
flex v2.4.7	152	14 184	9 134
byacc v1.9	178	3 553	2 737
diffutils v2.8	192	15 022	9 735
bc v1.06	204	7 794	5 290
userv v0.95.0	239	7 909	6 016
copia v?	242	1 168	1 085
gnuchess v5.07	261	16 533	11 045
tile-forth v2.1	286	5 730	3 549
li v?	357	7 597	4 793
espresso v?	361	22 050	21 780
go v?	372	29 629	22 118
ijpeg v?	467	28 185	15 253
ctags v5.0	518	13 750	10 018
sendmail v8.14	548	123 965	77 950
findutils v4.2.31	608	41 661	27 261
a2ps v4.13	902	54 954	33 573
gnubg v1.2	192	6 705	4 330
gnugo v3.6	2 188	151 376	110 631

Table 3. C language test programs for precision measurements

System	Number of procedures	TL	LCode
valgrind v3.3.0 (C)	5 318	228 763	141 631
gdb (C) v6.7.1	8 095	473 793	303 552
gcc (C) v4.0	16 108	1 052 353	725 620
mozilla (C++) v1.6	83 432	2 382 459	1 414 946

Table 4. C/C++ language test systems for efficiency measurements

4.5. The empirical process

We computed TSD and SEA/SEB dependency sets for all procedures of the subject programs. A TSD dependency set of a particular procedure f included those procedures of the program that were involved in statement level slices having their criterion in f, while SEA/SEB dependency sets were obtained as described previously. An additional program was used for the calculation of precision and to record performance of the tools, namely running times of the analysis phases and information about the sizes of the graphs produced.

We must note here a deficiency of CodeSurfer, which has a problem in the handling of some code structures related to unstructured control when determining forward slices (in these cases false dependencies are computed by the tool). Fortunately, this problem is not present with backward slices, so in all of the remaining parts of the study we use solely SEB relations, and the TSD consist of the dependencies in the backward direction only. Note that this restriction does not invalidate our results regarding precision since the overall size of forward dependencies is the same as for the backward dependencies.

4.6. Precision

We calculated precision of SEB compared to backward TSD as the golden standard and therefore we investigated the differences in the sizes of the respective dependency sets. For a given procedure, the precision value is the ratio of size of the set of dependencies identified by TSD divided by size of the set of SEB dependencies.

Since SEA/SEB does not produce false negatives, we always get 100% recall; TSD set is always a subset of a SEA/SEB dependency set. In our first experiment we investigated such precision values by determining, for each subject, the average precision values of the procedures of the given program. Figure 7 shows the average precision data for all of the programs. It can be observed that, apart from one outlier where the precision is only about 67%, we get values between 77,28% and 98,77%, which we think is very high.

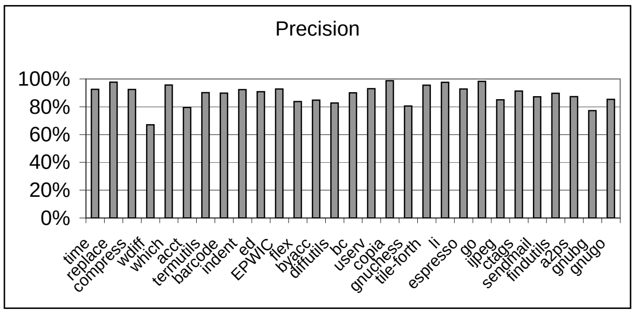

Figure 7. Precision of the SEB sets relative to backward TSD (recall is always 100%)

It is also interesting to see how the sizes of average dependency sets relate to each other and the size of the program. Figure 8 compares the average sizes of the sets of backward TSD with the average sizes of the sets of the SEB dependencies. It can be seen also visually that the SEA/SEB relations are very close approximations of the TSD relations.

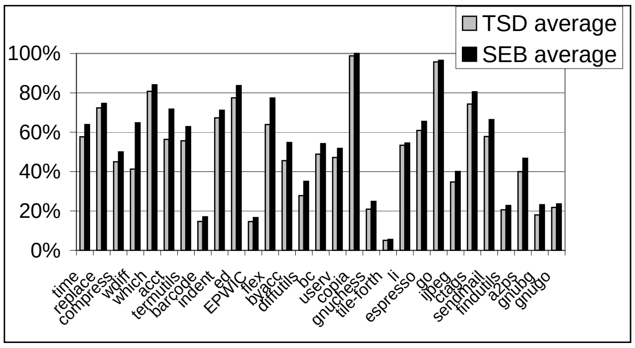

Figure 8. Dependency set sizes relative to program size

Figure 9 shows the distribution of the sizes of differences between TSD and SEB. The differences have been computed for each procedure as the ratio of the false dependencies of SEB over the total number of procedures of the given program. In most cases the differences of the sizes of the two sets are below 15%, the average being 4,27%.

However, there are some outliers as well. We found that three programs were solely responsible for the cases where the difference was greater than 25%. We investigated these situations and found that 13 procedures out of the 21 procedures in this group belong to the program `wdiff`. We

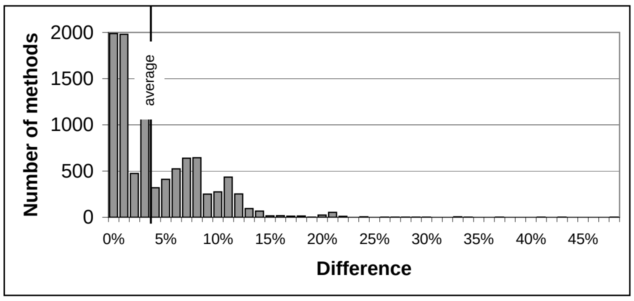

Figure 9. Distribution of the extent of differences between SEB set and backward TSD set sizes

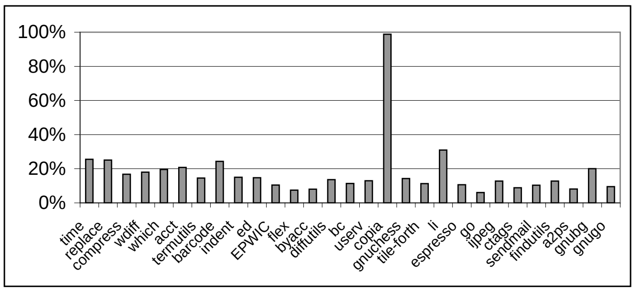

Figure 10. The recall values of dependency sets using call graphs only (precision is always 100%)

found that some of the dependencies in SEB were correct dependencies that have not been identified by TSD because the slicing algorithm has certain problems related to structure-fields. Hence SEA/SEB in these cases improves recall of the analysis, giving the programmer important dependencies that are missed by TSD algorithm. Besides that, there were also several true false dependencies in SEB that can be attributed to the conservatism of the SEA/SEB approach. A typical example is when a specific data is accessed by two different procedures interchangeably, for instance when they are part of a loop, and none of them modifies the data. Then there will be no TSD between these procedures, but SEB will identify a dependency due to the sequential execution of them.

As an interesting aside, we were able to compute call graph dependencies easily and we investigated them from the point of view of precision and recall as well. It is interesting to observe that in this case the precision will be 100% every time, but the recall will be very low as shown in Figure 10.

To conclude, we may say that our hypothesis about precision of SEA/SEB is correct and that SEA/SEB is a real alternative to TSD in this respect, see the summary in Table 5.

	precision	recall
SEA/SEB	**good**	**100%**
Call graph	100%	bad

Table 5. Summary of precision and recall

4.7. Performance

The two program representations used in our case study share the same structure on the highest level, namely both include a node for each procedure of the program. There are, however, significant differences in the amount of data to be stored for a procedure. Table 6 contains the relevant numbers. It can be easily deduced that ICCFGs require a significantly smaller number of nodes and edges and the difference is about two degrees of magnitude.[3] CodeSurfer could not handle the SDG of the biggest program so the corresponding data are missing from the table.

valgrind	SDG	vertices	1 920 150
		edges	6 947 024
	ICCFG	vertices	154 509
		edges	179 626
gdb	SDG	vertices	10 086 409
		edges	48 876 108
	ICCFG	vertices	160 340
		edges	185 611
gcc	SDG	vertices	18 775 143
		edges	81 492 908
	ICCFG	vertices	467 185
		edges	584 972
mozilla	SDG	vertices	N/A
		edges	N/A
	ICCFG	vertices	1 587 499
		edges	1 723 611

Table 6. The sizes of the different graph representations

Finally, we compared the analysis times of the two methods using an AMD Opteron 2.2 GHZ processor with 4G memory. Table 7 shows the parsing times (the common front end part) and the required building times of the SDG and the ICCFG graphs. Due to the size differences in the program representations and the computation costs for the required additional dependencies, the building time of the

[3]Theoretically, in one moment of time the ICFG must be completely stored in the memory to derive ICCFG, however in a practical implementation it could be done in an optimized way by the analysis of one procedure at a time, for instance.

SDG is substantially longer; the SDG for the largest program, `mozilla`, could not be built at all due to resource exhaustion, while the ICCFG could be built in reasonable time.

System	Parsing time	SDG building time	ICCFG building time
valgrind	5 min	16 min	2 min
gdb	8 min	124 min	4 min
gcc	69 min	571 min	11 min
mozilla	113 min	N/A	54 min

Table 7. Graph representations' building times

We note that we did not investigate the times for performing the actual dependency computations themselves, since we treat the complexity of these reachability-based operations as being similar and inexpensive.

With these results we can support the other hypothesis that there is a superior performance of SEA/SEB approach over TSD. Furthermore, it is to be expected that a specialized and optimized implementation of this approach could produce even better performance than the one listed in Table 7. In experiments shown here we used a general source code analysis framework which probably performs many superfluous operations that are unnecessary for SEA/SEB computation; a specialized implementation would further improve efficiency.

4.8. Threats to validity

Several issues may limit interpretation of our results. As noted previously, the precision was computed for the comparison between a traditional slicer that provided the golden standard and new SEA/SEB algorithm, which raises the issue of construct validity. These results are applicable where a traditional slicer is considered to be the standard tool against which the results of an analysis are to be measured, and may not translate to other situations.

Although different program sizes have been used in the empirical work, we are aware of the fact that the selected test programs may not fully represent all programs, particularly programs that employ different technology. The results reported here are obtained for specific subject programs and specific tools and should be generalized to other situations with caution. In particular, we used procedural programs written in C in our experiments, while object oriented programs may behave differently in terms of the structure of ICCFG and SDG, which determines the precision of the approach.

We performed the study on granularity of procedures; other granularities (statements, files or classes) may produce different results.

5. Conclusion

In this paper, we presented data that compare TSD that are used in slicing and other software tools, and SEA/SEB relation. The data we explored show that the computation of SEA/SEB is much more efficient while the precision declines only very slightly, by some 4% on average. In the majority of the cases this difference remains below 15%. Because of the higher efficiency, SEA/SEB scales up to larger programs.

In order to explain our results, we believe that algorithms that compute TSD contain numerous small imprecisions. For instance, often more efficient but less precise algorithm is used for handling points-to information, arbitrary control flow, structure fields, arrays, and so forth. These little imprecisions accumulate and lead to a situation where the big picture, computed by TSD algorithms, is imprecise. SEA/SEB algorithms have intuitive imprecision built into them from the beginning, but in the end they produce comparable results. We are aware that dependencies observed at the granularity of procedures are less precise than dependencies observed at the granularity of statements. Because of this issue and since statement-level slicing tools are so popular, we are going to investigate statement level SEA/SEB tool in the near future and compare the size of the slice generated by such tool with the size of the traditional slice. However we also note in this context that according to Binkley *et al.* [7], the dependencies at different granularities correlated well to each other.

There are several obvious optimizations that can further improve efficiency of SEA/SEB. We are planning to design and implement an optimized demand driven SEA/SEB algorithm. We are also planning to implement a specialized front end that may produce additional efficiency. Based on these observations, we are planning to implement the SEA/SEB algorithm in the Columbus tool [11, 12].

Finally, there is still the problem of a realistic set of software dependencies that the software contains. We want to continue with the research of hidden dependencies, see whether SEA/SEB still misses some of them, and develop further algorithms that would produce a more complete dependency set.

Acknowledgements

The authors wish to thank the help of Tamás Gergely in designing the algorithms and we also want to acknowledge several useful discussions with Radu Vanciu. We are also grateful to Dave Binkley for sharing with us some of the test programs, and to GrammaTech, Inc. for providing help with using CodeSurfer.

This work was supported in part by Hungarian National grants no. RET-07/2005 and OTKA K-73688. Václav Rajlich was supported in part by grants from the National Science Foundation (CCF-0438970) and by the 2005 and 2006 IBM Faculty Award. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF or IBM.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. Efficient and precise dynamic impact analysis using Execute-After sequences. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 432–441, May 2005.

[2] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany, Mar. 1996.

[3] L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 167–175. IEEE Computer Society, 2005.

[4] Á. Beszédes, T. Gergely, Sz. Faragó, T. Gyimóthy, and F. Fischer. The dynamic function coupling metric and its use in software evolution. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 103–112, Mar. 2007.

[5] Á. Beszédes, T. Gergely, J. Jász, G. Tóth, T. Gyimóthy, and V. Rajlich. Computation of Static Execute After relation with applications to software maintenance. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM'07)*, pages 295–304, Oct. 2007.

[6] D. Binkley. Source code analysis: A road map. In *Future of Software Engineering (FOSE'07), at 29th Int. Conference on Software Engineering*, pages 104–119. IEEE Computer Society, May 2007.

[7] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Transactions on Software Engineering and Methodology*, 16(2), Apr. 2007.

[8] 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, Sept. 2003.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[10] D. C. de Leon and J. Alves-Foss. Hidden implementation dependencies in high assurance and critical computing systems. *IEEE Transactions on Software Engineering*, 32(10):790–811, 2006.

[11] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[12] FrontEndART Software Ltd. `http://www.frontendart.com`.

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

[14] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[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] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[17] W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103. ACM Press, Jan. 1991.

[18] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] J.-K. Ng. Context-sensitive control flow graph. Master's thesis, Iowa State University, Ames, Iowa, USA, 2004.

[20] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering held jointly with 9th European Software Engineering Conference (ESEC/FSE'03)*, pages 128–137, Sept. 2003.

[21] B. G. Ryder. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5(3):216–226, May 1979.

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

[23] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Variable precision reaching definitions analysis for software maintenance. In *Proceedings of the First Euromicro Conference on Software Maintenance and Reengineering*, pages 60–67, Mar. 1997.

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

[25] Z. Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*, pages 293–299. IEEE Computer Society, 2001.