Computation of Static Execute After Relation with Applications to Software Maintenance

Árpád Beszédes, Tamás Gergely, Judit Jász, Gabriella Tóth and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 544145
{beszedes,gertom,jasy,gtoth,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

In this paper, we introduce Static Execute After (SEA) *relationship among program components and present an efficient analysis algorithm. Our case studies show that SEA may approximate static slicing with perfect recall and high precision, while being much less expensive and more usable. When differentiating between explicit and hidden dependencies, our case studies also show that SEA may correlate with direct and indirect class coupling. We speculate that SEA may find applications in computation of hidden dependencies and through it in many maintenance tasks, including change propagation and regression testing.*

Keywords

Impact analysis, change propagation, regression testing, hidden dependencies, coupling, control flow analysis.

1 Introduction

In many software maintenance and evolution processes, the programmers deal with software components and their dependencies. For example whenever programmers change a software component, they must visit all dependent components and update them if necessary. Some of these dependencies are explicit, i. e., they can be understood as a mutual "awareness" among the components; examples of explicit dependencies between classes in object oriented systems are generalization, composition, association, and so forth. Typically these dependencies are expressed in the code as explicit references.

However besides explicit dependencies, there are also other dependencies that are not based on this mutual awareness; we call these dependencies *hidden dependencies*. Yu and Rajlich [27] explored hidden dependencies that are based on the existence of data flows between otherwise explicitly unrelated components. They used the Abstract System Dependence Graph (ASDG) that is based on that System Dependence Graph (SDG) [14], but whose computation is expensive and therefore not feasible for large programs.

In this paper, we propose an alternative way to determine the explicit and hidden dependencies by employing *Static Execute After (SEA) relation* among program components. The approach is motivated by Apiwattanapong *et al.* who introduced the notion of Execute After relations [1] and applied them in dynamic impact analysis; SEA is a static counterpart of this approach. In the paper, we show that SEA may approximate static slicing (which is a way to compute dependencies using SDG), while being much less expensive and more usable. When differentiating between explicit and hidden dependencies, the early data also show that SEA may correlate with direct and indirect class coupling. We speculate that SEA may find applications in computation of hidden dependencies and through it in change propagation and regression testing.

The paper is organized as follows: Section 2 contains examples and applications of SEA. Section 3 explains algorithms of SEA analysis, while Section 4 discusses implementation issues. Section 5 contains the case study, Section 6 summarizes related work, and Section 7 contains conclusions and future work.

2 Applications of SEA

Our goal was to find a simple analysis technique that captures dependencies among classes.[1] In our technique, we treat class B as dependent on class A if and only if B may be executed after A in any possible execution of the program. We compute such dependencies using the SEA relation between procedures (methods or functions). We say that $(f, g) \in SEA$ if and only if any part of g may be executed after any part of f in any of the possible executions of the program. Now, class B is said to be dependent on class A very simply if and only if a method f of A is in SEA relation with a method g of B.

2.1 Hidden dependencies and their computation using SEA

A simple example of a hidden dependency is shown in Figure 1. In it, class A contains the member function `int A::get()` that receives a choice of a color from the user input and encodes it as the returned integer, where value "0" means "red", "1" means "yellow", and so forth. Class B contains member function `void B::paint(int)` that receives the color code as an argument, decodes it, and paints the screen by the corresponding color. Class C contains the data flow between an instance of class A and an instance of class B. If the programmers change encoding of colors in class A, they also have to change the decoding in class B and vice versa, so although classes A and B do not reference each other and hence are not aware of each other, there still is the hidden dependency between them.

SEA may be used for approximate computation of hidden dependencies very easily. Namely, we can identify that method call `a.get()` is followed by calling `b.paint` which results in B being dependent on A according to our approach. (Note that the actual data flow is not checked, only the mere execution order is taken into account.) Finally, since A and B are explicitly unrelated we are able to identify that they are hidden dependent. With this approach any "invisible" interaction among classes may be captured.

The hidden dependencies have applications in a variety of software maintenance situations.

2.2 Application in change propagation

Whenever programmers make a change in the code, the modified component may no longer fit with the other components because it may no longer properly interact with them. In that case, secondary changes must be made in neighboring components, which may trigger further changes, and so forth. This process of repeated secondary changes is called change propagation and although each change starts and ends with consistent software, during the change propagation the software is often inconsistent.

```
class C { ...
    A a;
    B b; ...
    void foo() { ...
        b.paint(a.get());
        ...
    }
};
```

Figure 1. Example of hidden dependency

The changes propagate through the dependencies between the components, including hidden dependencies. Software maintainers need to trace those dependencies, make corresponding changes, and guarantee that the change has been propagated correctly and the software is returned back to consistency. A model of the change propagation [19] keeps track of the inconsistent program dependencies and proposes to the programmers the locations where the subsequent changes are to be made.

Hidden dependencies are harder to detect than the explicit dependencies and are more likely to be a source of residual bugs in the system. The software compilers do not detect errors that are caused by hidden dependencies and since they interconnect seemingly unrelated classes, the programmers are more likely to overlook them. Therefore it is very important to detect these dependencies and give programmers warnings about their existence and their potential to introduce subtle bugs into the software.

2.3 Application in Regression testing

After the programmers completed the update of the code including change propagation, they have to validate both new code and the old code. The regression testing validates the old code and its purpose is to prove that the old code, that was not touched by the change, does not contain any residual bugs, i. e., that the change propagation was completed without any omissions.

A primitive way how to do regression testing is to rerun the whole testing suite after each change. However, the regression testing is repeated often, making the use of the complete regression test suite too expensive. Firewall testing was developed as a heuristic to solve this problem and it retests only those components that directly interact with the changed components and does not run the tests that retest the rest of the system. The firewall heuristics assumes that

[1] In the applications we identified (change propagation, regression testing) the granularity of analysis is usually on class level for object oriented systems.

the residual bugs are most likely to be caused by a failure of the programmer to update directly interacting program elements.

While explicit program dependencies were used for the construction of the original firewalls, hidden dependencies are included in the construction of more sophisticated firewalls and they increase the chance to intercept the residual bugs [25].

2.4 Coupling measures

Coupling measures quantify the strength of relationship of pairs of classes; unrelated classes should have coupling measure equal to 0, while strongly related classes should have a high value of coupling measure. A set of coupling measures and their relationship to impact analysis was investigated by Briand *et al.* [8]. Coupling is also an important topic in software measurement, when metrics are used as predictors for different quality indicators like maintainability and error proneness [4, 13]. Different kinds of static coupling metrics have been proposed in the literature [7]. Recent research deals with the problem whether indirect coupling is to be investigated in addition to direct coupling [26]. In this paper, we investigate the question whether classes with high coupling measures would also have more hidden dependencies. For this we used the well-known Coupling Between Object Classes (CBO) metric [9].

3 Computation of SEA

Our approach is motivated by the work by Apiwattanapong *et al.* who introduced the notion of *Execute After* relations between functions [1] and applied them in dynamic impact analysis. By definition, functions f and g are in Execute After relation if and only if any part of g is executed after any part of f in any of the selected set of executions of the program. This relation can be easily computed.

As a static counterpart of this approach we define the notion of the *Static Execute After* (SEA) relation. We say that $(f, g) \in SEA$ if and only if any part of g may be executed after any part of f in any of the possible executions of the program. It is obvious that SEA is a superset of the detailed System Dependence Graph based dependencies [14] that arise between procedures (i. e., static slices presented at procedure level). Control and data dependency may occur between two program points if there is a control or data flow between them, but data flow also requires control flow. If there is a control flow between these points, it means that one of them must be executed after the other, thus the procedures of the two program points are in SEA relation. An intrinsic property of the SEA relation is that it is safe but imprecise with respect to static slicing.

Following Apiwattanapong *et al.* [1] and Beszédes *et al.* [5] the SEA relation can be divided into three (non-distinct) sub-relations:

$$SEA = CALL \cup SEQ \cup RET \ ,$$

where

$$\begin{aligned} (f,g) \in CALL &\iff f \text{ calls } g \\ (f,g) \in SEQ &\iff \exists h : h \text{ calls } f \text{ first,} \\ &\quad \text{then } h \text{ calls } g \\ (f,g) \in RET &\iff f \text{ returns into } g \ , \end{aligned}$$

where both "call" and "return into" are treated transitively.

3.1 Algorithm for computing SEA

For computing the SEA relation a suitable program representation is needed. The traditional Call Graph [20] is unsuitable for our needs since it says nothing about the order of the procedure calls within a procedure. On the other hand, an Interprocedural Control Flow Graph (ICFG) [16] contains too much information and is expensive to work with. Thus, we define a new representation.

First we define (intraprocedural) *Component Control Flow Graph (CCFG)*, where only call site nodes are considered. Each CCFG represents one procedure and contains one *entry node* and several *component nodes* with *control flow edges* connecting them. Furthermore, strongly connected subgraphs are collapsed into single nodes; this means that if two call sites are reachable from each other by control flow edges then they are represented by the same node. *Interprocedural Component Control Flow Graph (ICCFG)* represents the whole system and for each procedure, there is a corresponding CCFG interconnected by *call edges* with other CCFGs. In the ICCFG there is a call edge from a component node C to a procedure entry of m if and only if at least one call site of C calls m. An example of ICCFG can be seen in Figure 2.

An algorithm for computing the SEA relation is presented in Figure 3. Initially, we determine the transitive calls of each procedure and so the $CALL$ relation (steps 1–4). In the next phase all procedures are processed again in order to determine the SEQ relation. For this we first topologically order the components of the current procedure and put them into a queue for further processing, which means that no component will be processed before any of its preceeding components. For a given component c, in step 8 we determine the set of procedures that might have been called in this procedure before the component ($prev[c]$), and in step 9 we compute the set of procedures transitively called by c ($calls$). Next, if the component directly calls multiple

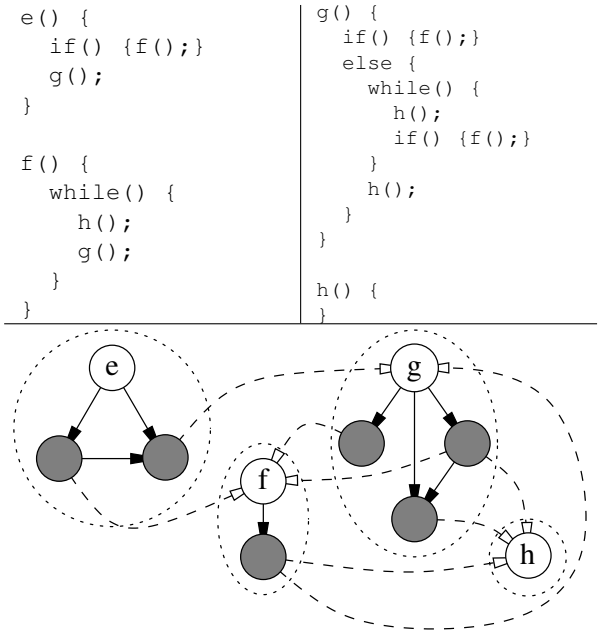

Figure 2. Example ICCFG

procedures, then the extension of $prev[c]$ with the calls is done at this point (steps 10–11). Now we can extend the SEA relation with all (f, g) pairs where f is called before the component and g is called from the component. Finally, we prepare the $prev[c]$ set for the next iteration if we have not done it yet in step 11.

The worst case computational complexity of the algorithm is the following. Let n be the number of procedures, k and e be the maximum number of component nodes and edges in the procedures, respectively, and m be the maximum number of procedure calls in a component. The algorithm first determines the transitive calls of all procedures, then it computes an ordering of the components and performs set operations for each component. If appropriate data structures are used, this is done in $O(n \cdot e + n \cdot k \cdot m)$ time, which is roughly the same as the computational complexity of the detailed SDG-based static slicing algorithm [14]. However there are significant differences between the two approaches. The main difference is in the building of the SDG and ICCFG. The building of either requires an Interprocedural Control Flow Graph. The ICCFG can be easily derived from it by deleting nodes and performing two depth-first graph traversals for finding strongly connected components. On the other hand, the building of the SDG requires the computation of control and data dependencies, which are additional (also complex) algorithms. Finally, the number of nodes in one procedure's graph (k) is also larger in the SDG than in the ICCFG. Thus, an overall computation complexity of ICCFG is significantly better than that of

program computeSEA(P)
input: P : ICCFG of program P
output: SEA : the SEA relation for all procedures

begin
1 $SEA := \emptyset$
2 **forall** m procedures of P
3 $transCallsOf[m] :=$ transitive calls of m
4 $SEA := SEA \cup (\{m\} \times transCallsOf[m])$
endforall
5 **forall** m procedures of P
6 $topOrder :=$ componentQueue(m)
7 **for** $c :=$ first($topOrder$) **to** last($topOrder$)
8 $prev[c] := \bigcup_{p \in \text{previous components of } c} prev[p]$
9 $calls :=$ transitive calls of c
10 **if** c directly calls multiple procedures
11 $prev[c] := prev[c] \cup calls$
endif
12 $SEA := SEA \cup (prev[c] \times calls)$
13 **if** c directly calls only one procedure
14 $prev[c] := prev[c] \cup calls$
endif
endfor
endforall
end

Figure 3. Computation of SEA

the SDG (on which the Abstract System Dependence Graph is based on as well [27]).

3.2 Handling dataflow

In its basic form, the algorithm presented above has the disadvantage that it captures data flow which is realized through procedure calls only. Data flow between global variables or direct class member variables are invisible to the algorithm due to the lack of corresponding nodes in the ICCFG. This problem is solved by a technique in which we convert all global variables into private members of new classes that can be accessed through *set* and *get* methods, before the ICCFG is built. Furthermore, each time a variable `X` is read (either global or member), it is replaced by `get_X()`, and when the variable is set to the value of an expression `e`, `set_X(e)` is called instead.

4 Implementation

We implemented the computation of the SEA relations and SEA-based hidden dependencies using a language-independent representation of programs. This means that

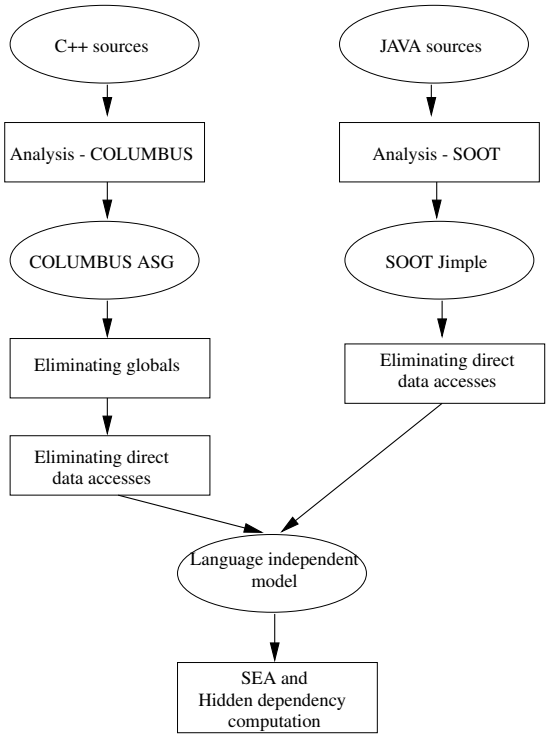

Figure 4. Tool chain

the core algorithm is able to compute the dependencies for arbitrary object oriented or procedural language simply by replacing the syntactic analyzer front end. The dependencies are computed between procedures based on the algorithm from the previous section, and then the dependencies are lifted to class level. We experimented with C++ and Java programs since we had access to analyzer front ends for these languages. Figure 4 shows the high level architecture of our experimental tool chain.

4.1 Building the ICCFG

In the first step the syntactic analysis of the source code is performed for which existing front ends have been used. Namely, for C++ programs we applied our analyzer, the Columbus reverse engineering tool [10], and for Java we used the Soot framework [21]. In both cases the provided APIs to the internal representations were used to extract relevant information. In the next step the conversion of data accesses to procedure calls is done as overviewed above. In the case of C++ programs an additional step is required; namely, the conversion of global variables into wrapping structures. An example of these modifications (projected back to source code) for C++ can be seen in Figure 5 (Java is very similar).

The next step is the extraction of the Interprocedural Control Flow Graph (ICFG). It is important to note that

```
int global=0;

class A {
  public: int i;
};

class B {
  A a;
  public: int foo() {
    return global+a.i;
  }
};
```

Original code

```
struct globalStruct {
  static int global;
  static int& get_global() { return global; }
};
int globalStruct::global = 0;

class A {
  public: int i;
  int& get_i() { return i; };
};

class B {
  A a;
  public: int foo() {
    return globalStruct::get_global()
           + a.get_i();
  }
};
```

Modified code

Figure 5. Handling globals and dataflow

we needed a safe way to determine call edges. Therefore we apply a conservative computation of calls narrowed by Rapid Type Analysis [3][2]. In the final step this graph is reduced to obtain the ICCFG, which is done in the following way. First, all but procedure entry and call site nodes are eliminated from the graph, and then the strongly connected components within the procedures are determined, while control flow and call edges are kept up to date.

4.2 Computation of the dependencies

The computation of the SEA dependencies is done on the built up ICCFG using the algorithm from the previous section. We then determine hidden dependencies between classes as follows. A class B will be hidden dependent on class A if and only if a method f of A is in SEA relation with a method g of B, and B is not explicitly dependent on A. In our implementation we choose to use UML static structure modelling concepts to determine explicit dependencies (but one can use any other specific interpretation

[2]Safety can be guaranteed only if there is access to the whole system's source code and with the restriction of highly dynamic aspects like function pointers and reflection.

instead). We say that B is explicitly dependent on A if one of the following holds:

- there is an association or aggregation between A and B (we identify these relations from source using heuristic that A has a member whose type references B),
- A directly calls a method of B,
- A and B are connected by generalization,
- A creates an object of type B.

4.3 Filtering dependencies

All nontrivial software systems consist of multiple classes, some of which belong to libraries. For the purposes of this work the analysis generally needs to be filtered in order to remove unimportant dependency data (like those connecting library classes). For this there are three possibilities:

Examine everything. This results in a huge set of dependencies, and includes even those that hold between two library classes. We can (and have to) do nothing with these classes, they are usually the same for all programs, so this approach needs to be refined.

Cut classes before computation. In this case all computations are made on a set of classes belonging to the subject system, and all other classes are neglected. This means that SEA is also computed for these classes only. But completely excluding library classes may result in loosing some important hidden dependencies among non-library classes. For example, if a library class "calls back" methods of subject classes A and B, the hidden dependency between A and B will be missed.

Cut classes after computation. This is the combination of the first two approaches. Namely, we compute the relations for the whole system, but then drop all library classes after the dependencies have been computed. This way no dependencies will be missed among subject system classes.

In our measurements, we employed the third variant.

5 Case study

5.1 Hypotheses and empirical design

We accepted the following hypotheses about the approach:

- SEA technique approximates static slicing; SEA captures all dependencies captured by slicing (the recall is 100%), but contains false dependencies as well (precision is lower than 100%).
- SEA technique is highly conservative, but even with the lowered precision, the number of dependencies is not excessive.
- SEA technique is more efficient than static slicing, and therefore based on the above can be used instead of it.
- Direct coupling (such as the Coupling Between Object Classes measure) is insufficient in predicting the effects of the changes made to the software, indirect couplings need to be taken into account as well.

For the first three hypotheses we compared SEA results with the results provided by a static slicer employing System Dependence Graph, where the slicer results were interpreted as the relationship between classes rather than relationship between statements. To empirically validate the fourth hypothesis we chose to use the CBO metric [9] as it is probably the most widely accepted coupling indicator in object oriented systems. However, we expect that any other direct coupling measure would produce similar results.

The experiments were performed on small to medium size C++ and Java programs; see Table 1. We had access to the Java slicer Indus [15], which allowed us to compute precision and recall in the case of the Java programs. For C++ programs, we were able to compute the CBO metric using the Columbus tool [10], so the comparison to coupling measure was done for these programs. The subject programs originate from open source projects, except for AbsHiddenDep, which is our implementation of the SEA technique. This program was investigated in two ways: alone, and with the associated libraries of the Columbus framework.

Program	Language	Number of classes/structs/unions
unrar	C++	72
mysqlcc	C++	135
licq	C++	172
stellarium	C++	203
AbsHiddenDep	C++	6
AbsHiddenDep (with libs)	C++	406
JSubtitles	JAVA	14
RayTracer	JAVA	12
java2html	JAVA	50
dynjava	JAVA	301

Table 1. The test programs

5.2 Case study results

Number of dependencies In our first experiment we counted the number of explicitly and hidden dependent class pairs in the subject programs, along with average numbers of dependencies per class. The results as absolute values are shown in Table 2, while Figure 6 graphically depicts the same average data relative to the program size. It can be observed that generally there are more hidden dependencies than explicit ones (as expected), but not in all cases. However, it is important to observe that although our approach is a highly conservative one, a comparably small amount of hidden dependencies can be observed.

Program	Explicitly dependent class pairs	Hidden dependent class pairs
unrar	396 (5.5)	2912 (40.44)
mysqlcc	1381 (10.23)	8952 (66.31)
licq	2082 (12.1)	3822 (22.22)
stellarium	2083 (10.26)	10700 (52.7)
AbsHiddenDep	12 (2)	14 (2.33)
AbsHiddenDep (with libs)	11872 (29.24)	95158 (234.38)
JSubtitles	76 (5.43)	98 (7)
RayTracer	86 (7.17)	58 (4.8)
java2html	398 (7.96)	1762 (35.24)
dynjava	14923 (49.58)	60672 (201.56)

Table 2. Number of dependencies. Numbers in parentheses are the average number of dependents per class.

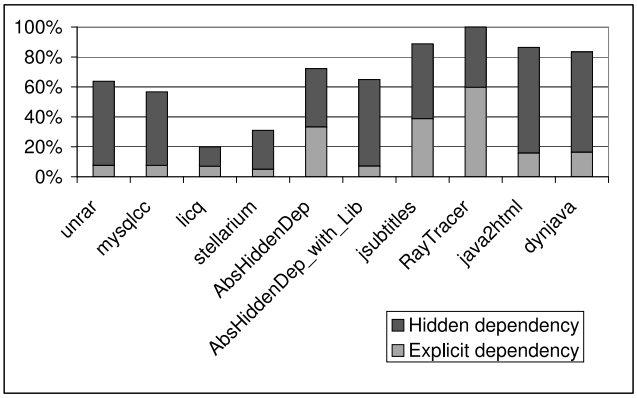

Figure 6. Average number of dependencies per class (relative to the total number of classes)

Beyond the total and average, a more complete view on dependency sizes can be obtained by investigating the distribution of sizes over all classes of the programs. Figure 7 shows this distribution in form of histograms of the number of dependencies per class (shown only for the largest programs). An interesting observation is that generally there some dependency sizes among the larger ones to which significantly more classes belong than to other sizes. By investigating this phenomenon we found that it is probably the same effect as other researchers observed working on program slice sizes, and called the "dependence clusters" [6, 22]. Namely, it turned out that not merely the dependency sizes were the same but the dependency sets were common too.

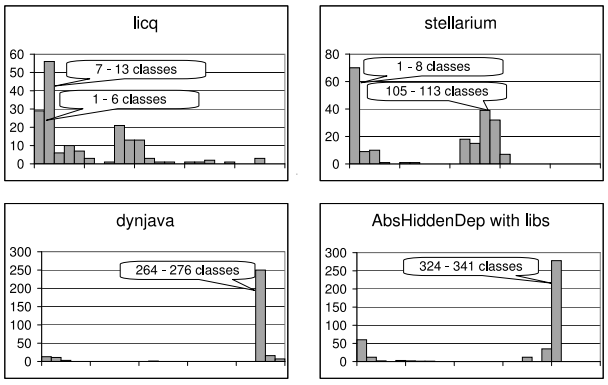

Figure 7. Distribution of dependency sizes. In the histograms the horizontal axis contains the dependency sizes in increasing order, while the vertical axis shows the number of classes with their dependencies belonging to the corresponding size.

Precision and recall We measured the precision and recall of our method with respect to the dependencies that come from class level slices. The program slices in each class and for all possible criteria in it were computed, and the resulting slices were lifted to class level (in other words, if a slice with criterion in class A includes a program point located in class B, then A and B are taken as dependent).

In Table 3 the number of classes, the average number of explicit and hidden dependent classes, and the corresponding average slice sizes (also in the number of classes) are compared for programs for which we had slicing results. Note that in these experiments the largest Java program is not present, since the Indus slicer could not produce slices for that program due to excessive memory consumption.

Program	Classes	Explicit dep.	Hidden dep.	Slice
JSubtitles	14	5.43	7	11.57
RayTracer	12	7.17	4.83	10.08
java2html	50	7.96	35.24	41.2

Table 3. Comparison of dependency sizes with slice sizes

Table 4 shows the precision and recall rates for these three programs. We compare SEA dependencies (explicit and hidden together) to static slice dependencies, and with precision we measure how many false dependencies we produce, while recall shows how many dependencies are missed. The first thing to observe is that the recall was not 100% in all of the cases. We investigated this issue thoroughly since our method is supposed to cover static slicing. Fortunately, we found that the problem was not with our method but with the slicer. Namely, in a few cases due to its conservative nature the slicer produced a false positive, while our method was more precise. In the same table we can see that the precision values are relatively high; in fact, much higher than we initially expected. After careful investigation we found that this is probably also due to the conservativism of the slicer. Namely, even the slices include many false dependencies, hence both the slicer and SEA share some false positives when compared to the intuitive meaning of hidden dependencies.

Program	Precision	Recall
JSubtitles	92.8%	100%
RayTracer	65.5%	100%
java2html	96.82%	99.36%

Table 4. Precision and recall (explicit and hidden dependencies together compared to static slice)

These results indeed justify our approach, since if SEA approximates static slices so well our method could be used instead of inefficient and also imprecise slicing algorithms.

Comparison to direct coupling measures Briand *et al.* [8] give a model for impact analysis based on coupling measures, and they conclude that not all consequences of a change are captured using only direct coupling measures like the Coupling Between Object Classes (CBO). To verify this hypothesis we compared CBO metric values computed for classes to the number of hidden dependencies found by our method.[3]

Figure 8 shows our findings in this experiment for the programs with most typical results. Even having in mind that we also get false dependencies, we may say that this result supports the hypothesis about the inadequacy of CBO measurement in many cases. We can find some classes in the systems with both low CBO and hidden dependency, but the most interesting areas are those that correspond to high hidden dependency values while having small CBOs. This suggests that in many cases CBO is not sufficient in predicting problematic parts of the system in terms of the amount of (direct and indirect) coupling. This can lead to serious problems if one relies solely on direct coupling like CBO for the prediction of impacts and error proneness, for instance.

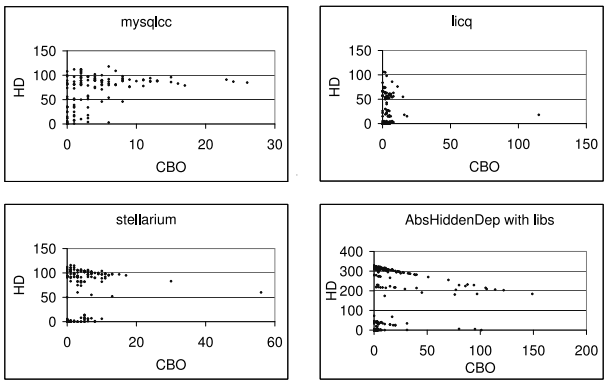

Figure 8. Connection between CBO metric and hidden dependency number. Each dot corresponds to a class in the system.

5.3 Threats to validity

As noted previously, precision and recall were computed for the comparison between a slicer and SEA technique, which raises the issue of construct validity. We maintain that these are valid comparisons for the purpose of the decision whether to use SEA or slicer techniques, and our results favor SEA. The question whether SEA (and the so computed hidden dependencies) are a better way to solve practical software maintenance related problems remains to future work.

The basis of our approach is the computation of procedure calls. This needs to be done in a conservative manner, thus all possible targets of the call must be considered, including for instance polymorphic calls. Furthermore, SEA theoretically handles any data dependency, including pointers, pointer arithmetic, dynamic memory handling, and so forth. However, there is still a question whether this approach trully handles *all possible* data dependencies. In our case study we did not encounter any problems related to these issues.

Dynamic aspects of the languages analyzed present another problem. For instance in C++, function pointers and reflection in Java will hinder the construction of an absolutely safe call graph. However, even in these cases it would be possible to provide a very conservative approach in which all possible procedures are called. This will lead however to much lower precision. Our measurements regarding precision and recall were done against a research

[3]Note that although the number of explicit dependencies is highly correlated to CBO values, they are not the same due to differences in their computation. We chose to use CBO since it is a widely known and accepted coupling measure.

slicing tool, which is also dealing with these issues and shares a similar bias as SEA.

Although different languages and program sizes have been used, we are aware of the fact that the test programs do not fully represent all programs. It remains to future work to investigate the applicability of our approach in real life scenarios. The results reported here are obtained for specific subject programs and specific tools and should be generalized to other situations with caution.

6 Related work

Different approaches exist to compute relations between higher level software structures supporting impact analysis [2]. Instruction level dependency and flow analyses are used in, e. g. program slicing [14, 24]. We employed a simplified approach, working on procedure level and only with flow information so as to be able to design more efficient algorithms, rather than computing detailed dependencies. The basis for our method is the computation of reachability on interprocedural flow graph. Tonella *et al.* presented a variable precision algorithm to determine reachability for program understanding, but this approach is also at instruction level [23]. Orso *et al.* mention an approximate static slicing method to be used in impact analysis [18]. They also employ reachability on the CFG, however no further details are given. Our work was motivated by the dynamic procedure level Execute After relations introduced by Apiwattanapong *et al.* [1].

The Context Sensitive Control Flow Graph (CSCFG) introduced by Ng is used for visualization purposes [17]. In this concept the details of unimportant parts of the CFG can be eliminated. However, no nodes are deleted from the CFG to produce the CSCFG, but unconcerned nodes are collapsed into single nodes, in the way that the original control flow of concerned nodes remains unchanged.

Yang *et al.* propose a method to compute the indirect couplings, which can also be seen as a way to detect hidden dependencies [26]. Génova *et al.* presented a work on how to cope with dependencies that are less easily observable from the code [12]. However, this work is about distinguishing some less explicit UML associations from firm "knows-about" associations, and not about dealing with dependencies spanning possibly across many classes and message chains.

7 Conclusions and future work

In this paper, we introduced Static Execute After relations to approximate static slicing, while being much less expensive and more usable. We applied SEA technique to the problem of finding hidden dependencies in the code and speculate about applications related to software maintenance. We followed a highly conservative approach with which we ensured that all dependencies computed by static slices are captured, but eventually false ones are captured as well. Compared to the dataflow-based methods, this approach enables us to design more efficient analysis algorithms with high recall rates but a lower precision. We are convinced that this is a sound trade-off in many applications, although this should be verified in real life case studies.

Static Execute After covers all possible interactions between classes, even the ones that are syntactically undetectable, like a semantic link between two different yet semantically linked constant literals. In the future, we want to broaden the notion of hidden dependency beyond the dependencies caused by data flows. We are also planning to investigate additional heuristics that increase precision of the analysis, for example to define pre- and post-conditions for the endpoints of the dependency relations. A possible enhancement to the method is to incorporate the level of indirection in the SEA relation, similarly to the DFC approach used in dynamic analysis [5]. We also plan to investigate the application of hidden dependencies as metrics for the prediction of software quality attributes like maintainability and error proneness.

The hidden dependencies are due to a "mediating" component that behaves as a link between the dependent classes (see our example at the beginning of the paper). We observed that often these mediators are apparent in the program design (c. f. Mediator Design Pattern [11]) and in some cases should be treated as a design problem. We plan to investigate this issue more deeply in the future.

Acknowledgements

This work was supported, in part, by grant no. RET-07/2005 of the Péter Pázmány Program of the Hungarian National Office of Research and Technology. Václav Rajlich was partially supported in this work by US National Science Foundation Grant CCF-0438970, US National Institute for Health Grant NHGRI 1R01HG003491, and the 2006 IBM Eclipse Innovation Award.

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] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[3] D. F. Bacon. *Fast and effective optimization of statically typed object-oriented languages*. PhD thesis, EECS Department, University of California, Berkeley, 1997.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct. 1996.

[5] 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 2007)*, pages 103–112, Mar. 21–23, 2007.

[6] 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, Sept. 2005.

[7] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[8] L. C. Briand, J. Wüst, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 475–482, Sept. 1999.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[10] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – reverse engineering tool and schema for C++. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 172–181, Oct. 2002.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.

[12] G. Génova, J. Lloréns, and J. M. Fuentes. UML associations: A structural and contextual view. *Journal of Object Technology*, 3(7):83–100, 2004. (electronic edition).

[13] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

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

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

[16] 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.

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

[18] 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.

[19] V. Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the 1997 IEEE International Conference on Software Maintenance (ICSM'97)*, pages 84–91, Oct. 1997.

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

[21] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`

[22] A. Szegedi, T. Gergely, Á. Beszédes, T. Gyimóthy, and G. Tóth. Verifying the concept of union slices on Java programs. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR 2007)*, pages 233–242, Mar. 21–23, 2007.

[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] L. White, K. Jaber, and B. Robinson. Utilization of extended firewall for object-oriented regression testing. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 695–698, Sept. 2005.

[26] H. Y. Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 212–221, 2005.

[27] 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, May 2001.