Extending Program Slicing in Aspect-Oriented Programming with Inter-Type Declarations

Timon ter Braak
Faculty of EEMCS
University of Twente
t.d.terbraak@student.utwente.nl

ABSTRACT
Program slicing can be very useful for understanding and debugging aspect-oriented programs. Program slicing is a common technique, but it is not trivial to use in aspect-oriented programming. Current implementations of slicing tools lack support for features of popular aspect-oriented languages. This paper proposes a detailed algorithm for the construction of an aspect-oriented system dependence graph, with support for inter-type declarations. Known algorithms can then calculate a program slice based on a dependence graph.

Keywords
Aspect-oriented programming, program slicing, dependence graph, inter-type declaration, analysis

1. INTRODUCTION
Object-oriented programming was introduced to enable a more modular approach in designing software. Each object has responsibility for a certain task. Such an object contains code to accomplish that task. Also code is placed into objects to implement other tasks of the program, that can not be separated in a module. If the code of a task is spread across multiple objects, it is called a crosscutting concern.

Aspect-oriented programming (AOP) can be seen as an evolution of object-oriented programming. In AOP, code of a crosscutting concern can be relocated from an object to a single module, called an aspect. So programs in AOP can be split into aspects by the definition of a concern. A piece of code that is relocated is called advice. An aspect consists of multiple advices that belong to the same concern. Each advice must be executed at some point in the program. These join points can be specified with a complex expression, called pointcut expression. Thus, an aspect-oriented program can be split into two parts, the base code and the aspect code.

An aspect-oriented program should be compiled with an aspect-oriented compiler, called a weaver. The weaver evaluates the pointcut expressions, which results in a set of join points where advices should be executed. Weaving the advices with the base code then results in a traditional byte-code program.

The way aspects interact with the base code is defined by a join point model (JPM). Different aspect languages can have different JPMs. A popular aspect language for Java, AspectJ, has two JPMs. We just explained one of them in the introduction, called pointcuts and advice. The other one is called inter-type declarations.

1.1 Inter-type declarations
An aspect can add functionality to objects. Aspects may declare members that cut across classes and their hierarchies. These declarations are called inter-type declarations. An aspect can declare whatever is needed from other classes. That way aspects can express their concern in one place. To clarify this feature, we give an example.

Example: in figure 1 a concern is specified that some objects have a name. Aspect NameAspect declares a new interface HasName, and implements that interface with a private field name and method getName. As a result of the aspect specification, every instance of Point, Line and Square implements the interface HasName.

```
1 aspect NameAspect {
2 3 private interface HasName {} 4 declare parents: (Point || Line || Square)
5 implements HasName;
6 private String HasName.name;
7 public String HasName.getName() {
8 9 10 }
```

Figure 1: Example of an aspect that contains inter-type declarations.

1.2 Program slicing
Ishio points out in [IKI04] that because of the features of AOP, debugging, testing and verifying programs can be more complex then with traditional programming. In this domain program slicing could be a useful technique. This technique presents the programmer a set of code that is related to the problem he is working on. The set of code is determined by a specification given by the programmer, the slicing criteria. Slicing can be used to find the part of the program that affects the criteria, or is affected by the criteria. If we are interested in the part of the program that affects the criteria, we have to calculate a backward slice. That slice is calculated by backtracking the control and data flow paths. We
can also calculate a forward slice, the part of the program that is affected by the criteria. A forward slice is calculated by traversing all possible control and data flow paths starting with the criteria [RH06].

There are two types of program slicing, static slicing and dynamic slicing. Static slicing is used for program understanding and verification. This technique analyzes source code of a program to extract the possible behavior of the program. Dynamic slicing analyzes a program execution with a certain input. This type of slicing is used to support a debugging task [HJ90].

2. RELATED WORK
Ottenstein and Ottenstein were the first to consider slicing as a reachability problem in a dependence graph. That is a graph where each node represents a point in the program, and the edges of the graph represent dependencies between the nodes. They used a program dependence graph for static slicing of single-procedure programs [OO84]. Horwitz et al. introduced the system dependence graph (SDG) for multi-procedure programs. They also proposed a modeling system for parameter passing, on which our modeling technique is based [HRB90]. We will explain this technique later on.

Most work done in the domain of program slicing is based on an SDG. The algorithms of [LI96] use an SDG for slicing object-oriented programs. Slicing of aspect-oriented programs was proposed by [Zha02]. The algorithm they proposed uses an aspect-oriented system dependence graph (ASDG), and they proved that an ASDG can be regarded as an extension of an SDG. Therefore a program slicing algorithm as proposed by [LI96] can create a slice, based on the ASDG of the target program.

2.1 Static slicing
There are a few attempts to implement program slicing for AOP, but there is no tool available yet [KI04]. [BM04, BDCM05] attempt to create such a tool. Their solution uses a Java bytecode approach. After the weaver creates a traditional Java program, known slicing algorithms can be applied to create the program slice. Bytecode contains information about original source code lines, that can be used to map the generated slice onto the source code [BDCM05]. An advantage of this approach is the assurance of correct application of the language constructs. There are also some disadvantages of bytecode analysis. Changes in the compiler have a direct effect on the program slicing tool. Every time the compiler changes, the program slicing tool might need to be adapted. Another problem using this approach concerns inter-type declarations. That will be explained in the problem statement.

2.2 Dynamic slicing
Dynamic slicing is used to find a slice of a program with a certain input. This approach results in a possible smaller slice, because some options in the control flow might be eliminated. Dependence Cache is an advanced form of dynamic slicing. This technique extracts dynamic data during execution. That information is used to determine the data dependencies in the program. The control dependencies are statically determined from the source code, thus this technique is assisted by static slicing. DC-slicing is probably the best known slicing technique, mainly for its reasonable cost for calculation of a program slice [IKI04].

3. PROBLEM STATEMENT
To maintain some benefits of AOP, such as modularization, a programmer should not have to understand the whole program he is writing. Program slicing can be useful to understand a relevant part of the program, for example an advice. For model-checking a program, a relevant part (slice) of the program can be extracted [BM03]. This approach results in less state-explosion. Hence, static program slicing can be useful to analyze aspect-oriented programs. In this paper we focus on static slicing, and leave dynamic slicing for future work.

Program slicing should be as powerful as the aspect language that is used. Therefore it should support all features of the aspect language. Currently this is not the case, because inter-type declarations are not supported yet.

In the AspectJ weaver inter-type declarations are implemented by direct bytecode manipulation. After weaving the program, information about inter-type declarations is lost. Because [BDCM05] requires bytecode information, it is impossible to map the bytecode to the AspectJ source code. Therefore inter-type declarations can not be supported by the program slicing procedure of [BDCM05]. To support inter-type declarations in a program slicing tool, another implementation approach should be used.

The construction algorithm of Zhao does not depend on bytecode. This algorithm can be extended with support for inter-type declarations. In [Zha02] he also briefly mentioned inter-type declarations [BM03]. Zhao roughly described an approach to construct an ASDG. Our contribution is a more detailed ASDG construction algorithm, by which we focus on support for inter-type declarations.

4. APPROACH
As mentioned before, some work has been done about program slicing based on an ASDG. First we need to clarify some fundamentals of the construction of an ASDG, as proposed by Zhao in [Zha02]. Then we can extend the construction algorithm of an ASDG with support for inter-type declarations. As a consequence we can use the same program slicing technique as Zhao, who used the two-pass slicing algorithm proposed in [HRB90]. We use the example program in figure 4 to illustrate our findings.

4.1 Program dependencies
The edges of an SDG are dependencies between parts of the program. Two main types of dependencies are control dependencies and data dependencies [RH06]. These terms will be explained in the next example.

Example: in figure 2, line 4 will be executed if k==0 at line 3 evaluates to true. Otherwise line 6 will be executed. Thus, line 3 decides which line will be executed, so line 4 and line 6 are control dependent on line 3. In the same function, the value of the expression return v at line 8 is determined by the definition of v at lines 4 and 6. Hence, the expression return v at line 8 is data dependent on the lines 4 and 6.

4.2 Construction of an ASDG
Our algorithm is an extension of the algorithm proposed by Zhao in [Zha02]. The first step in that algorithm is the construction of an SDG of the base code. A SDG is constructed by modeling the dependencies between statements
of the base code. The SDG of (the base code of) our example is can be found in figure 5. The way parameters are modeled in the SDG will be explained in our results.

In the second step of the algorithm, an Aspect Dependence Graph (AsDG)\(^1\) is constructed for each aspect in the program. An AsDG is a graph, similar to an SDG, only with additional edges to represent features of aspect languages. We have to extend this part of the algorithm with support for inter-type declarations. We give more details about this part of the algorithm in the next section.

The final steps of the algorithm weave the graphs together, resulting in a single dependence graph. For this we need to evaluate the pointcut expressions to get the join points where advices should be applied.

The ASDG construction algorithm as proposed by Zhao:

- Constructing the SDG of the base code of the program.
- Tools used in object-oriented programming can be useful to generate the SDG automatically [LH96].
- Constructing an AsDG for each aspect of the program.
- Determining joinpoints in the SDG of the base code.
- Weaving the SDGs at the joinpoints and add edges between the SDG and AsDG to represent the behavior of aspect-oriented features.

5. RESULTS

5.1 Modelling variables

Almost any program uses some form of parameter passing. Both [BDCM05, Zha02] use a method to model parameter passing. We will describe this method here. Zhao also described this modeling technique in [Zha02], but he does not consider the dependence relations that exists when a method returns to its caller. We also explain indirect data dependencies between methods that use instance variables (class fields).

5.1.1 Parameter passing

When a method with parameters is called, the caller passes certain values to the parameters. On the side of the caller the value is called an actual parameter. On the side of the callee, we call this a formal parameter. The method temporarily stores the actual parameter in a formal parameter. After the method is finished, the formal parameters are no longer bound to a value. In figure 3 parameter passing is modeled.

![Diagram of parameter passing](image)

Figure 3: Parameter passing between caller and callee.

To calculate a more precise slice, we need to analyze the data flow of each parameter. This is called interprocedural data flow analysis and was already used by Horwitz et al. in [HRB90]. If there is a statement that can modify the parameter, it might affect following statements. Therefore we need to model the modification of the parameter, by creating parameter-out nodes for that parameter. We do not have to analyze a parameter if a primitive type is used as a value, because a primitive type can not be modified. We only have to check references [BDCM05]. If a parameter is not modified, we can discard the parameter-out nodes. This results in a more precise slice, because we also discard dependencies that are related to that parameter. Thus, for each parameter of a call we create an actual-in and formal-in node. The calling statement supplies a certain value for the actual-in node, so a control dependence exists between them. On calling the method, the value of the actual-in node is copied to the formal-in node. This is modeled by a parameter-in edge between the actual-in and formal-in nodes, and by a call edge between the caller and callee.

In most languages, a method can also have a certain return value. Similar to the parameter modeling, we can model this with a formal-out and actual-out node. In the special case of a constructor, the return value is a new object. That object can be seen as a set of values, so we create a new formal-out node that represents that object, and make it data-dependent on all other formal-out nodes of the constructor. As a result, we have modeled that every statement that refers to the created object, is data-dependent on every field in that object.

Example: in figure 5, node s9 represents the statement `Point p1 = new Point(3,4)`. The node creates actual-in nodes and then calls the constructor. After the constructor is finished, an object of type `Point` is returned. This is modeled by the nodes f5_out and a5_out.

---

\(^1\)Note the difference between ASDG and AsDG. An aspect-oriented system dependence graph (ASDG) is a representation of an AO program. An aspect dependence graph (AsDG) is a representation of an aspect. These terms are already introduced in previous research, and therefore we use them likewise.
5.1.2 Instance variables
An aspect, as with an ordinary class, can have instance variables. An implicit declared interface or class can also have instance variables through inter-type declarations. For each instance variable we create formal-in and formal-out nodes [Zha02]. This can occur both in the constructor and in a method that uses an instance variable. We indicate nodes that represent the same variable, with an equal number in their attributes.

Example: in figure 5, node me2 is the constructor of the class Point. The node me2 creates formal-in and formal-out nodes for the instance variables Point.x and Point.y.

As explained in the previous section, we have to check if parameters are modified in a method. Information about a modified parameter is then passed on to successor statements by a parameter-out edge. We should do the same for instance variables. Currently instance variables can be modified outside the notice of other methods. For example, a class can have an instance variable that is set by one method (setter), and read by another method (getter). The variable is then modified without propagation of information about that modification. Due to the imperfection of our modeling technique, a single instance variable is referenced in multiple nodes. We can accept that a variable is data dependent on itself. As a consequence, a data-dependence relation exists between nodes that represent the same instance variable. To indicate this relation, we add a data dependence edge from the formal-out to the corresponding formal-in. We noticed that both [BDCM05, Zha02] do not mention these dependencies.

So each “formal” node, that is not connected to an “actual” node, has to represent an instance variable. If we want to obtain a backward slice, we have to match formal-out nodes with the formal-in node (with the same number), and for a forward slice we have to match formal-in nodes with the corresponding formal-out node.

Example: in figure 5, node me5 controls f1_in and f2_in. These instance variables are data-dependent on f1_out and f2_out (connected by me2).

5.2 Handling advice
To model advices in our dependence graph, we first have to find the joinpoint where the advice should be applied. In this paper we have manually evaluated the pointcuts. There are many cases, especially the AspectJ-weaver, where this evaluation is already implemented in a tool. For each join point we find, we have to alter the control flow of the base code. Advices can be applied before, after and around a join point. Before and after advices are executed respectively before and after the join point. To weave such a type of advice into the graph, we have to connect the join point with the advice by a call edge [IKI04]. The pointcut expression is then removed, because it is no longer needed.

The application of around advice is less trivial. On triggering an around advice, first the body of the advice is called. With the statement proceed the methods body is called, which eventually returns to the advice. The proceed statement does not necessarily have to be called. Therefore we are interested in whether or not the proceed statement is called, because it has a huge impact on the program structure.

5.3 Representing inter-type declarations
Inter-type declarations are extensions of already existing classes, or implementations of implicit declared interfaces or classes in an aspect. When we create an AsDG of an aspect, we create a representation of each class that is referred to by an inter-type declaration in that aspect. In the graphs we represent an aspect, class or interface with a rectangle node. We can add each inter-type declaration in the form of a field or method as a successor of the particular class or interface. This relationship is represented with a membership edge. In the AsDG in figure 6 we used this method to model inter-type declarations. Note that we, as in our example, represent a referenced class as a member of the aspect. In this way the hierarchy of the graph remains, and we prevent orphaning. This is useful because we are in an intermediate stage of constructing the ASDG. After weaving the graphs, we remove all membership edges and class or interface representations.

5.4 Weaving the graphs
In this stage we have multiple graphs, depending on the number of aspects in our program. For each aspect a graph should be constructed. As before, we talk about an SDG for the base code of the program, and about an AsDG for an aspect. We evaluate each pointcut of each aspect to get a set of join points in the base code. We connect every advice in the aspects to the their join points in the base code with a coordination edge. A coordination edge is an ordinary call edge, and therefore introduces nothing new in the domain of program slicing. It differentiates itself to highlight the interaction between base code and advice. After adding the coordination edges, the pointcut expressions and the membership edges to the aspect are removed.

Example: around advice is a special case, as described earlier. Around advice puts itself before and after the joinpoint. In the example in figure 4 we have an around advice at ae24. If we evaluate the corresponding pointcut expression pe21, we see that the advice puts itself between the caller a10 and callee me5.

The last step in the process is the weaving of the inter-type declarations. At least one class or interface node is member of the aspect. The declare parents statement can declare a superclass or implementation of an interface. In the case of a superclass, the constructor of the specified class must be connected with a coordination edge to the constructor of the superclass. In our example, we have an implicit constructor of GeometricObject at ce12. In the case of an interface, additional functionality is specified that is added to certain types of classes. That functionality may or may not be used by the program. If it is not used, it will not be part of the program slice. Otherwise it is used by some advices and it will be connected by edges as if it was part of the base code.

5.5 The algorithm
In the previous part of this section we described our findings on parts of the problems with inter-type declarations. Here we give an algorithm to construct an ASDG of a program. We do not describe the construction of the SDG of the base code in detail. We think it is trivial, and there are even tools available to do this, like Indus [RH06]. Thus, we only need to know how to construct an AsDG, and how to weave all the graphs together to get the ASDG. In our algorithm we combined the last two stages of Zhaos algorithm into one step.
Step 1: we construct the SDG. A slicing tool for object-oriented programming can be used to create an SDG automatically.

Step 2: for each aspect of the program we construct an AsDG.

- Create a root node `ase` that represents the aspect.
- Connect `ase` with every pointcut expression `pe`, advice `ae` and aspect methods `ame` by a membership edge.
- For each advice `ae`, connect the related pointcut expressions `pe` with the advice `ae` by a data dependence edge.
- Connect `ase` to the nodes that represent each declared class or interface `ce` by a membership edge.
- Connect each inter-type declared class `ce` to its class fields and methods by a membership edge.
- Construct other control and data dependencies between ordinary methods and statements as done in an SDG.

Step 3: we weave the SDG and AsDGs together, which results in an ASDG.

- For each matching pair of nodes of type (`pe`, `ae`): Find the joinpoints in the SDG `jp` by evaluating `pe`. For each `jp`, if `ae` is of type:
  - before: connect `jp` leftmost with `ae` by a coordination edge.
  - after: connect `jp` rightmost with `ae` by a coordination edge.
  - around: let `ca` be the predecessor of `jp`. Connect `ca` with `ae` by a coordination edge. Connect the node with the `proceed` statement in the body of `ae` with `jp` by a coordination edge. Change the target of every outgoing edge from `jp` (or its successors), from `ca` to `ae`. For every incoming edge at `pe`, change the target from `pe` to `ca`.
- Remove every node `pe`.
- Connect each constructor with its declared super-class by a coordination edge.
- Remove all membership edges and root nodes.

6. APPLICATION OF A SLICING ALGORITHM

Horwitz et al. proposed a two-phase slicing algorithm based on an SDG in [HRB90]. That algorithm is more precise then the one Weiser proposed in [Wei81]. The main problem in program slicing is to keep track of the calling context when a method call becomes part of the slice, and the algorithm “descends” into that method. Horwitz et al. used the modeling of parameter passing to overcome this problem. The graph is traversed two times. In each traversal, we are allowed to follow different kind of edges. The direction of the traversal depends on the kind of slice we want. Upward traversal results in a backward slice, downward traversal in a forward slice. We did not have to make any changes in the slicing algorithm, but we substituted unfamiliar terms with our own terms.

Phase 1: we traverse the ASDG by following all edges, except parameter-out edges.

Phase 2: we traverse the ASDG again in the same direction, but we only follow control, data and parameter-out edges. We do not follow call and parameter-in edges.

6.1 An Example

In this paper we used the example program in figure 4 to illustrate our findings. Suppose we are interested in the statements that will be affected by statement `s23`. We need a forward slice to acquire this information. To obtain a forward slice we traverse the graph top-down, left-right from our starting node `s23`. We have to ignore parameter-out edges in phase 1 of the algorithm. When we traverse the graph, we find node `m16` with its related nodes. We mark all these nodes as reachable from `s23`. In phase 2, we need to traverse the graph starting with the marked nodes. We only follow control, data and parameter-out edges. Now we find additional nodes in the ASDG that are part of the program slice. The ASDG with the complete program slice is displayed in figure 7.

7. DISCUSSION

With program slicing we want to find the minimum slice. A slice is minimal if there is no other slice that contains fewer statements. A minimum slice does not have to be unique. [Wei81]. In most cases when we use static slicing, we find a slice that is not close to the minimum slice. The main reason for this is dynamic behavior of the program. Some joinpoints are based on dynamic context. In AOP it is also allowed to use wildcards in pointcut expressions. Dynamic behavior makes static analysis of the program harder. To ensure we capture every statement the program slice should contain, we need to consider every possible execution trace [BM03]. This results in possible unnecessary statements, and thus a bigger slice.

With dynamic slicing we calculate only one execution trace, which results in a smaller slice. It is difficult to give an idea on how big the difference is between these two techniques in general. Unfortunately the problem of finding the minimum slice is undecidable [Wei81]. Balzarotti et al. wrote about the precision of program slicing [BM04]. They concluded that the precision of a hypothetical tool, based on program slicing, is low. The main problem of the imprecision is keeping track of the calling context of the calling method [HRB90]. A technique that may increase the precision of slicing is called interprocedural data flow analysis. Information about variables that can be modified by a procedure is used to eliminate actual-out and formal-out nodes [Tpp95]. We applied this technique in modeling parameter passing, but more intensive analysis can be used to eliminate dependencies.

We introduced a technique to model data dependencies in the constructor of an object. We proposed that the statement that calls the constructor, data depends on all fields created by that constructor. This method is safe, but not precise. More analysis should be applied, that results in elimination of dependencies on unused variables.

Pointcut expressions are not represented in the ASDG. Depending on the slicing criteria, advice on certain joinpoints are part of the slice and therefore represented in the ASDG as a method. As with pointcut expressions, inheritance relations, such as statement `pd13` in figure 4, are used during the construction of the graph. In our example this results in `ce12` being part of the slice. For model-checking this behavior is desirable. But if a programmer uses program...
slicing to understand the structure of the program, it might be useful to include these declarations into the slice.

In the AspectJ compiler (ajc) inter-type declarations are implemented by direct bytecode manipulation. Therefore program slicing based on bytecode can not easily be extended with support for intertype-declarations. There exist more compilers for AspectJ, like the AspectBench Compiler (abc). Different compilers can have a different implementation strategy, so there might become a compiler available which is more suitable for slicing based on bytecode. It remains very complex to integrate the features of inter-type declarations in a compiler. Like ajc, the abc compiler does a lot of internal manipulation to correctly apply inter-type declarations [ag04].

8. CONCLUSIONS
In this paper we proposed a more detailed and refined construction algorithm of an aspect-oriented system dependence graph (ASDG). Now we can model inter-type declarations in an ASDG. An ASDG can be used in known slicing algorithms, and therefore inter-type declarations are now supported in program slicing. Likewise, other features of AOP that are still missing or might become available can also be integrated in an ASDG. If a program slicing tool based on an ASDG should be developed, it can support all features of an aspect-language such as AspectJ.

9. ACKNOWLEDGMENTS
We want to thank Marielle Stoelinga and Lodewijk Bergmans for their supervision and fellow students for their useful comments during the realization of this paper.

REFERENCES
[ag04] The abc group.
Building the abc aspectj compiler with polyglot and soot.

[BDCM05] Davide Balzarotti, Antonio Castaldo D’Ursi, Luca Cavalaro, and Mattia Monga.
Slicing aspectj woven code, 2005.

[BM03] L. Blair and M. Monga.
Reasoning on aspectj programmes.

[BM04] D. Balzarotti and M. Monga.

[HJ90] Agrawal H. and Hogan J.
Dynamic program slicing.

Interprocedural slicing using dependence graphs.
In ACM Transactions on Programming Languages and Systems, volume 12, pages 26 – 61, 1990.

Debugging support for aspect-oriented program based on program slicing and call graph.

[LH96] Loren Larsen and Mary Jean Harrold.
Slicing object-oriented software.

The program dependence graph in a software development environment.

[RH06] Venkatesh Prasad Ranganath and John Hatcliff.
Slicing concurrent java programs using indus and kaveri.

[Tip95] F. Tip.
A survey of program slicing techniques.

Program slicing.

[Zha02] Jianjun Zhao.
Slicing aspect-oriented software.
```java
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + this.x + ", " + this.y + ");
    }
}

public class Screen {
    public static void main(String[] args) {
        Point p1 = new Point(3, 4);
        System.out.println("Coordinates of point 1: " + p1.toString());
    }
}

public aspect ColorAspect {
    private static class GeometricObject {
        public static final int GeometricObject.RED = 1;
        private int GeometricObject.color;

        public void GeometricObject.setColor(int color) {
            this.color = color;
        }

        public int GeometricObject.getColor() {
            return this.color;
        }

        pointcut createObject(GeometricObject g): this(g) && execution(*.new(int, int));
        pointcut objectToString(GeometricObject g): target(g) && call(String *.toString());

        after(GeometricObject g): createObject(g) {
            g.setColor(GeometricObject.RED);
        }

        around(GeometricObject g): objectToString(g) {
            return proceed(g) + 
                    "Color: " + g.getColor();
        }
    }
}
```

Figure 4: Example of an aspect-oriented program.
Figure 5: SDG of the base code (classes Point and Screen). The attributes of the nodes relate to the lines in figure 4. Formal and actual nodes are used to model parameter passing. Their representation is clarified below the graph by matching attributes.
<table>
<thead>
<tr>
<th>PARAMETER-IN:</th>
<th>f7_in: this.RED=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a9_in:</td>
<td>color_in=GeometricObject.RED</td>
</tr>
<tr>
<td></td>
<td>f8_in: this.color=this.color_in</td>
</tr>
<tr>
<td></td>
<td>f9_in: color=color_in</td>
</tr>
<tr>
<td></td>
<td>f10_in: g=g_in</td>
</tr>
<tr>
<td>PARAMETER-OUT:</td>
<td>f7_out: this.RED_out=this.RED</td>
</tr>
<tr>
<td>f8_out:</td>
<td>this.color_out=this.color</td>
</tr>
<tr>
<td>f11_out:</td>
<td>retval_out=&lt;string&gt;</td>
</tr>
<tr>
<td>f12_out:</td>
<td>retval_out=this.color</td>
</tr>
<tr>
<td>a9_out:</td>
<td>ret = retval_out</td>
</tr>
<tr>
<td>a10_out:</td>
<td>ret = retval_out</td>
</tr>
</tbody>
</table>

Figure 6: AsDG of the aspect *ColorAspect* of the program in figure 4. The shaded nodes represent the inter-type declarations in the aspect.
Figure 7: ASDG of figure 4. The bold nodes are marked during phase 1 of the slicing algorithm. Together with the other shaded nodes, they indicate a forward slice with respect to statement s23.