This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Illgen, T., Program Modeling for Fault Definition Based Static Analysis, Proc. 14th Int. Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, Springer-Verlag, pp. 911-912

Program Modeling for Fault Definition Based Static Analysis

Thomas Illgen
University of Paderborn
Department of Electrical and Electronics Engineering
D-33095 Paderborn, Germany
ti@adt.uni-paderborn.de

Keywords: Static Analysis, Program Modeling, Fault Classification, Static Slicing

Abstract

Detecting faulty structures in programs is a difficult part of software reliability engineering and takes a lot of effort. Generating effective test data to show that a faulty structure could produce a failure poses many complexity problems. Code reviews are known as an effective method for detecting potential critical program structures. But detecting this structures in a code review takes a lot of effort and is error prone.

This paper presents a method to detect potential faulty structures by program modeling. Therefore we build a database containing informations about classified faults. For analyzing the program we generate a first order predicate logic model implemented in horn clauses. We show how to detect the fault structures defined in the database and how to reduce the effort for the code review by slicing the relevant parts of the program.

1 Introduction

Detecting potential critical structures in a program is a difficult but important part in software development and re-engineering. To show that a critical part of a program can produce failures in some cases poses a lot of complexity problems, because in most cases we have to find test cases which generate a marginal value for a variable at specific program points. On the other hand most of these potential critical structures are well known and understood. This is the first step to reveal them. A formal definition of these potential faulty structures is useful and enables the automation of program analysis.

In industrial practice code reviews, inspections and walkthroughs are effective methods for detecting potential faulty program structures. Unfortunately the effort of these methods is very high and the support of automation fells short. On the other hand, if we have defined these critical structures formally human static analysis of a program could be automated. Therefore we need an adequate representation of the program. First order predicate logic implemented in horn clauses seems to be a very good representation for this analysis. A first order predicate logic model for the program under test could be generated by a parser. The parser builds the well defined interface between the program and the model. So the analysis is easy to adopt to different programming languages.

The formal definition of critical program structures and an effective model of the program in horn clauses enables the automation of analysis. The goal is a detection of a critical program part. To show that an input or variable definition generates a failure in a potential faulty program structure in some cases is a very complex problem. To show that a failure occurs at a specific program point for some test cases we often have to generate an input that produces a marginal value at this critical program point. Because of the complexity of this problem it is not possible to automate but there exist some effective methods to reduce the complexity of this problem without complete automation of test-case generation. One of these methods are slicing techniques. Program slicing reduces the complexity of the program to be analyzed. The reduction of complexity depends on the program structure. The problem of slicing criteria is, that the tester has to fix a critical program point by his/her own.

The method presented in this paper detects a critical program point depending on a potential faulty structure and generates a static slice. The reduction of this slice depends on the program structure. If the critical part contains only constants we can fix the potential fault directly. If the critical program part contains variables we have to build a static slice including all program paths effecting each variable of the critical program part. The slice reduces the program complexity and narrows the focus of attention to the relevant parts of a program during the test process.

The faulty structures are defined in a knowledge base

and has the character of a fault specification. We define a critical program point $CPP(v, l, Def)$ as a critical node or statement at location l for a variable v based on a fault definition *Def*. For this critical program point we generate a static program slice $SPS(P, v, l, Def)$ for a given program P. Depending on the defined knowledge about the fault structure it is possible to generate a checklist that tells the tester which parts of the program are critical and what to analyze.

2 Background

We use first order predicate logic to define the potential fault structures and to model the program under test. Implementing these definitions in horn clauses represented in PROLOG makes the definitions executable. The declarative model enables an easy analysis of the program. It enables the detection of critical program structures and static slicing of this parts. In this section we give an overview to the used methods for modeling the program, complexity reduction using slicing techniques and fault classification.

2.1 Horn Clauses

We use PROLOG as modeling language and for the definition of potential faults. PROLOG is a logic programming language consisting of a set of Horn clauses. Horn clauses are universal quantified disjunctive clauses with at most one positive literal. These are classified in three types called *fact*, *rule* and *goal*. Definite clauses, or program clauses are facts and rules. A rule has a head A and a body $B_1, ..., B_n$. As usual, we denote Horn clauses by

$$\begin{array}{ll} A, & (\textit{fact}) \\ A \leftarrow B_1, ..., B_n, & (\textit{rule}) \\ \leftarrow B_1, ..., B_n. & (\textit{goal}) \end{array}$$

A fact states something which is true, a rule has the meaning that its head A is true provided that the conjunction of its body literals $B_1, ..., B_n$ is true. A goal is used to activate a logic program P, i.e., it is a question if the conjunction of the literals $B_1, ..., B_n$ follows from P [3]. The representation in PROLOG-clauses enables execution of the program model and the call of the specified potential faults as goal. The goal is a call of a PROLOG-rule that specifies the fault to be detected in the program. The call of a fault specification initiates the program analysis.

2.2 Slicing

The definition of a program slice differs lightly in different papers. There are two main classes of slicing criteria.

Static Slicing
In Weiser's approach, slices are computed by computing consecutive sets of transitively relevant statements, according to data flow and control flow dependences. Only statically available information are used to compute slices [12, 11]. Weiser defines a static slice as an executable subset of statements of a program. The static slicing criteria could be further break down in two types [8]:

- The static slicing criterion $C = (p, v)$ requires the computation of a static slice with respect to a program point p and a variable v, which is not necessarily used or defined at p [12, 6].
- The static slicing criterion $C = (p, v)$ requires the computation of a static slice with respect to a program point p and a variable v, which is used or defined at p [9]

The computation of static slices is shown as a reachability problem in a program dependence graph [7], i. e. data dependence and control dependence. Weiser computes slices with respect to a set of data flow equations [12].
Figure 1 shows an example for a static slice $C = (p, v)$, with variable v =`prod` at the program point p, with p is the output of the calculation for variable `prod` in line 13.

```
1  program calc;
2  var x, prod, sum, n: integer;
3  begin
4    prod := 1;
5    sum := 0;
6    read(x, n);
7    while n > 0 do begin
8      prod := prod * x;
9      sum := sum + x;
10     n := n - 1;
11   end;
12   write(sum);
13   write(prod);
14 end.
```

Figure 1. An example program which calculates the n power and n sum of x.

Dynamic Slicing
The second main class of slicing criterion is dynamic slicing. The dynamic slicing criterion $C = (x, p, v)$ requires the computation of a dynamic slice of a program executed on input x with respect to an execution point p and a variable v, which is not necessarily used or defined at p [1, 4]. In the case of dynamic slicing, only dependences that occur for a specific execution based on an input x are taken into account. A dynamic slice is build by a trace

```
1  program calc;
2  var x, prod, sum, n: integer;
3  begin
4    prod := 1;
5
6    read(x, n);
7    while n > 0 do begin
8      prod := prod * x;
9
10     n := n - 1;
11   end;
12
13   write(prod);
14 end.
```

Figure 2. An executable slice of the program in figure 1 using the criterion (prod, 13).

of the program. It is very well regarded by a variation of flow-back analysis [2].

The difference between static and dynamic slicing is that dynamic slicing assumes fixed input for a program, whereas static slicing does not make any assumptions regarding the input. In this paper we use static slicing to reduce the complexity of the program. Therefore we use the static slicing definition of Weiser [12], e. g. we build executable static slices. We defined the location of the considered program point p as the *critical node CPP(v,l,Def)*. Therefore it is important, that the static slice contains the critical path *CP(S(Def), CPP(v,l,Def))*. A *path* is a sequence of nodes $path(n_s, n_e) = (n_s, n_2, ..., n_{e-1}, n_e)$ with n_s is a unique entry node and n_e is a unique exit node. The critical path has a unique starting node and a unique exit node, the critical point. The starting point *S(Def)* depends on the fault definition *Def*.

2.3 Fault Classification

The definition and classification of faults is a fundamental requirement for their systematic and analytic detection. The set of faults classified in their characteristics depends on the requirements of the analysis. It is difficult to find a complete, disjunctive set of classes in which all faults could be classified. There are several ways to classify faults. One fault classification is distinguished in computational errors and domain errors. Domain errors can be further broken down in two classes, path selection errors and missing path errors [13]. These are classifications oriented on the affects in the control-flow or output of the program. Our requirement is to define fault classes for an effective static analysis. Therefore it is useful to build a set of fault definitions with common characteristics. These definition is the knowledge base for detecting them in our program model. We initiate the analysis by an inquiry to the system. Therefore we have to build a search tree or graph containing a set of fault definitions in their nodes. The edges in the tree or graph couple the nodes in a logical AND. Figure 3 shows a small cut of a fault classification tree.

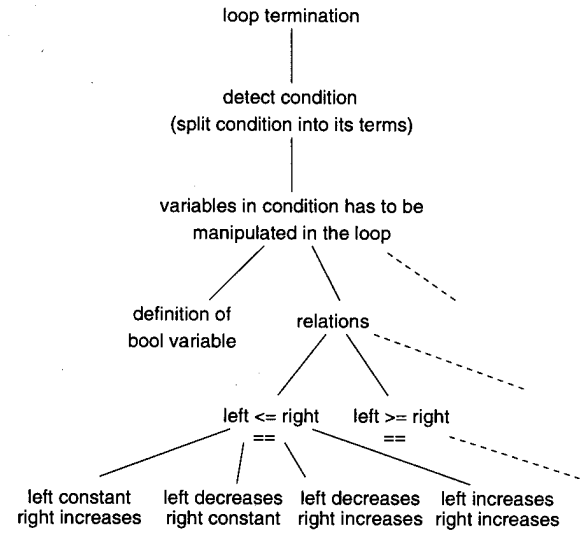

Figure 3. Cut of classified fault

The fault classification describes some possible faults which cause non terminating loops. Off course the tree is not complete. The dotted lines signal that additional branches are possible. So the tree could grow with the experiences of the tester. Off course we cannot solve the terminating problem with our method but we can analyze potential critical structures and give checkpoints for an inspection.

In the example we are looking for the condition for the loop termination. In the first step we have to find the predicate of this condition. After that we have to split the relation into its terms. If the condition is a single boolean variable we have to find out, if the variable is defined in the loop again. We cannot decide, if the variable is defined in the right manner but we can analyze the loop body for a definition use of the boolean variable. If there is a relation as loop condition one possible fault is that the terms are not redefined in a correct way. If for example a loop is iterated until the left side of the relation is greater than the right side, the requirement of the redefinition is that the value of the left side increases, the right side decreases, both or that the left side increases faster than the right side.

In addition to that we need some knowledge about the influence of the operators to the redefinition of the variables. Look at the table below.

operation	action
$+< var >$ $< var >< = 0$	decrease
$-< var >$ $< var >> = 0$	decrease
$*< var >$ $< var >< = 0$	decrease / alternate
. .	

The table defines our knowledge about the relations between the arithmetic operations and their influence to the variables in the conditions. The table must also contain the knowledge about the influence of combined operations. For example the system has to know, that a multiplication increases the value of a variable faster than an addition. Chapter 4 shows an example for detecting a fault of this type. Even if we have this knowledge we cannot decide if the loop will terminate or not. This depends on the values unified with the variables. But we can locate potential critical parts in the program and reduce the complexity of an inspection by building static slices for this program point.

3 Program Modeling

In this section we summarize the terminology to be used in the paper, without being complete and formal in order to brief our analysis and modeling approach. We model the program in three types of facts.

- Facts to describe the data flow.
- Facts to describe the control flow.
- Facts to describe the program instructions and to couple the data flow with the control flow.

The facts to describe the data flow are outlined in the typical manner of variable use, e.g. d-use for the definition of a variable, c-use for the computational use of a variable and p-use for the predicative use of a variable [10, 5]. The data flow of the program is modeled in the following PROLOG-facts:

- `d_use(<variable name>,<line number>, <operation>).`
  (Convention for the representation of the d-use of variable `variable name` consists of `line number` to identify the location of the definition of the variable and the `operation` manipulating the variable.)
- `c_use(<variable name>,<line number>).`
  (Convention for the representation of the c-use of variable `variable name` consists of `line number` to identify the location of the computational use of the variable.)
- `p_use(<variable name>,<line number>, <operation>).`
  (Convention for the representation of the p-use of variable `variable name` consists of `line number` to identify the location of the predicative use of the variable and the `operation` using the variable.)

The variable name defines the specifier of the variables data-flow. The line number specifies the line in which the variable is used. The operation declares the manipulation or use of the variable. The program under test is expressed by the parser to instrument the program (adding of line numbers, write each instruction in a new line) for identification of the variable use by line numbers.

The possible control flow of the program is modeled in directed edges of the form:

```
edge(<line number>, <line number>).
```

The first `line number` identifies the start line of the edge, the second `line number` identifies the end line.

The third type of PROLOG-facts generated describes the program instructions. Examples are:

Functions:
```
func(<line of function head>,
  <function name>).
```

Statements:
```
state(<line number>,
 <list of data-flow predicates>).
```

Decisions:
```
decision(<line number>,
 <if, else, switch>).
```

Loops:
```
loop(<kind of loop>,
 <line of loop condition>,
 <list of data-flow predicates>).
```
The third argument in `for`-loops contains a list of the three arguments defining the data flow n the loops head.

End of block:
```
end(<line number>, <kind of structure>,
 <line of structure>).
```

Inputs:
```
input(<list of variables>).
```

Outputs:
```
output(<line number>,
 <list of outputs>).
```

```
 1 main() {
 2   int i, x, y, a;
 3   scanf("%i %a", &x, &y);
 4   a = -3;
 5   FOR( i = a; i = i + 1 <= x; i = i + 1) {
 6     y = y + i;
 7     i = i * 2;
 8   }
 9   printf("%i", y);
10 }.
```

Figure 4. An constructed example program containing two potential faults

Declarations:

```
declaration(<line number>, <type>,
 <list of variables>).
```

The `line number` specifies the line of the instrumented file to locate the instruction presently under work. The list of data-flow predicates contains the names of the variables which are manipulated or used by the specified command. They close the connection between data-flow and control flow. Inputs could also be seen as a definition use of the variable but they have the character of a start-node. Thus, it is useful to model them separately. These clauses contain all informations required to generate the fault specification based static slices.

Following we explain the program model in an example before we in the next chapter explain the analysis. Figure 4 shows a part of a program containing a FOR-loop.

In the first step we have to model the program in the presented manner. The model is oriented on the lines of the statements. To unify a statement exactly with a program line we first have to expand the program, especially the loop. The loop contains three statements in its head. Between this statements exists a defined control-flow to be modeled. To be able to identify the statement at the code line we have to write them in separate lines. In the first parse the program in figure 5 is generated.

The parser generates the following clauses for the data-flow:

```
declaration(4, int, i).
declaration(5, int, x).
declaration(6, int, y).
declaration(7, int, a).
d_use(x, 8, scanf).
d_use(y, 8, scanf).
d_use(a, 9, a = -3).
d_use(i, 12, i = a).
d_use(i, 13, i = i + 1).
d_use(i, 14, i = i + 1).
```

```
 1 main()
 2 {
 3   int
 4     i,
 5     x,
 6     y,
 7     a;
 8   scanf("%i %i", &x, &y);
 9   a = -3;
10   FOR
11     (
12        i = a;
13        i = i + 1 <= x;
14        i = i + 1;
15     )
16     {
17        y = y + i;
18        i = i * 2;
19     }
20   printf("%i", y);
21 }
```

Figure 5. Expanded program in figure 4

```
d_use(y, 17, y = y + i).
d_use(i, 18, i = i * 2).
c_use(a, 12).
c_use(i, 13).
c_use(i, 14).
c_use(y, 17).
c_use(i, 17).
c_use(i, 18).
p_use(i, 13, i = i + 1 <= x).
p_use(x, 13, i = i + 1 <= x).
```

The directed edges model the control-flow graph of the program without respect to conditions. The potential possible control-flow graph of the program in figure 5 has the structure below.

```
edge(1,2).     edge(2,3).     edge(3,4).
edge(4,5).     edge(5,6).     edge(6,7).
edge(7,8).     edge(8,9).     edge(9,10).
edge(10,11).   edge(11,12).   edge(12,13).
edge(13,15).   edge(13,14).   edge(13,19).
edge(14,15).   edge(15,16).   edge(16,17).
edge(17,18).   edge(18,13).   edge(19,20).
edge(20,21).
```

The third type of clauses, adopting the data flow with the control flow and representing the program instructions are for the example outlined in the following PROLOG-notation.

```
func(1, main).
func(8, scanf).
loop(FOR, 10, [[d_use(i, 12, i = a),
   c_use(a, 12)],
```

```
  [p_use(i, 13, i = i + 1 <= x),
   p_use(x, 13, i = i + 1 <= x),c_use(i, 13)],
  [d_use(i, 14, i = i + 1),c_use(i, 14)]]).
state(9, [d_use(a)]).
state(12, [d_use(i), c_use(a)]).
state(14, [d_use(i), c_use(i)]).
state(17, [d_use(y), c_use(y), c_use(i)]).
state(18, [d_use(i), c_use(i)]).
state(20, [output]).
output(20, [["%i", y]]).
end(15, loop_condition, 10).
end(19, loop, 10).
end(21, func, 1).
```

Program modeling is also a powerful method to handle pointer. We could represent a pointer in a predicate structure containing the pointer name, the name of the structure to be pointed to and a symbolic address.

```
pointer(<pointer name>, <points to>,
        <symbolic address>).
```

The symbolic address is important to handle arrays or strings, where manipulations directly on the address are useful. The symbolic address is also a fact containing the value or program part to be pointed to. Of course we cannot model an exact copy of the storage but this is not necessary for static slicing. We have to distinguish if a manipulation of a pointer affects the address or the element to be pointed to. Therefore we use the C-typical notation (reference = &, dereference = *) ahead the variable name. In the model structure we replace variable names in the presence of pointer with the fact defined above.

4 Analysis

The method we introduce is a static analysis of the program code. The analysis process will be carried out in following steps.

1. Define potential faults as knowledge base for the static analysis.

2. Expand the program (Each statement in separated lines).

3. Generate a model of the program under test.

4. Detect critical structures in the program under test by an inquiry to the fault database. This initiates the analysis process.

5. Build static slices containing all dependencies of the variable v at location l for a given fault definition *Def.*

The analysis is initiated by an inquiry to the system. The program in figure 5 contains two potential critical points. The first one is a potential error in line 13. Depending on the compiler the relation $1 <= x$ could be evaluated before the calculation $i = i + 1$. In this case the loop will not terminate. The detection is not difficult. It is a *depth one fault.* That means, the search tree has only a search depth of one. The fault is classified as priority fault. To detect it we have to expand the relation in the third term of all p_use-facts and control if one term in the relation is calculated. If one term is calculated, is it enclosed in brakes. We can throw a message that the tester obviously means $(i = i + 1) <= x$ instead of $i = i + 1 <= x$. This fault is not very critical, because it occurs in every test-run that involves this loop and could be detected by a code coverage.

The second potential fault is much more critical. It depends on the test-cases to detect it with conventional test methods. Consider the example in figure 5. i is initiated in the loop by a. a is defined with the value -3. In the loop-head i is increased with one and in the loop-body multiplied with 2. Because the multiplication manipulates the value of i faster then the term in the loops head, i will decrease in every run of the loop. The other variable used in the termination conditions of the loop x is not manipulated in the loop. So this situation causes a not terminating loop.

Let us now see how we could detect this fault by using heuristics. First we have to detect the loop condition. It is described in the clause `loop(FOR, 10, [p_use(....` After that we have to find all d_uses in the loop for the variables used in the loop condition. This terms are also described in the rule above. We find $i = i + 1$ in the relation of the condition, $i = i + 1$ in the third argument of the loop and $i = i * 2$ in the loops body. The last term is described in the `d_use(i, 18, i = i * 2)` clause. This clause must be in the loops body, because the loop starts in line ten and ends in line 19 (clause `end(19, loop, 10)`). With this knowledge we can create a list of manipulations for the loop index i. Last we inquire the database defined in the tabular in chapter 2.3. In the example we could solve the problem with the result (critical for $x >= 0$) because the initial value of i has a defined value of a. This is not possible at all. If we don't know the initial value of i or it is calculated in the program the problem is to complex to solve. In this case we have to add the checkpoint (critical for $i <= 0$) in the analysis report. That means, the automation of the analysis depends on the number defined variables in the critical structure. If all used variables are constant we could defintly detect a fault. In presence of variables in this structures we could give some checkpoints for a further manual analysis like described above. This exactness of this checkpoints depends on the number of variables and on the knowledge about the faulty structure

```
 1 main()
 2 {
 3   int
 4     i,
 5     x,
 6
 7     a;
 8   scanf("%i %i", &x, &y);
 9   a = -3;
10   FOR
11     (
12        i = a;
13        i = i + 1 <= x;
14        i = i + 1;
15     )
16     {
17
18        i = i * 2;
19     }
20
21 }
```

Figure 6. Static slice for the example program in figure 5 and a given fault definition

defined in the fault classification.

To reduce the complexity of the program for further manual analysis we generate a static slice. Therefore we have to consider all dependencies of the critical path, e. g. each statement which influences i in the loop condition and the loop body up to line 13. We generate the slice shown in figure 6.

5 Conclusion

We have presented a method for a fault classification based static analysis to detect potential critical program structures. Therefore we build a model of the program implemented in horn clauses. Most of these faulty structures are dependent on the data flow of variables. Thus we could definitely detect a fault, if all variables are defined, e. g. they are constant. If some variables are not constant in the critical program path, we generate a static slice containing all program paths which influence the not defined variables in the critical path to reduce the complexity for a further inspection. In the analysis report we outline checkpoints for these inspection based on our knowledge about the fault structure.

The method is easy to adopt to different programming languages. The parser which generates the model is the well defined interface between the program syntax and the analysis. Even if some faults are typical for a programming language, global defined rules for the analysis like for example finding a possible path could be used. The presented method does not respect the presence of goto's. In this case we additionally have to analyze which program path jumped to influence variables in the critical path.

References

[1] J. Laski B. Korel. Dynamic slicing of computer programs. *J. Syst. Software*, 13:187–195, 1990.

[2] R. M. Balzer. Exdams-extendable debugging and monitoring systems. *Proceedings of the AFIPS SJCC*, 34:567–586, 1969.

[3] O. Jack F. Belli. A test coverage notation for logic programming. *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, pages 133–142, 1995.

[4] J. R. Horgan H. Agrawal. Dynamic program slicing. *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementations*, pages 246–256, 1990.

[5] John H. R. May Hong Zhu, Patrick A. V. Hall. Software unit test coverage and adequacy. *ACM Computer Surveys*, 29:366–427, December 1997.

[6] M. Weiser J. R. Lyle. Automateic program bug location by program slicing. *IEEE Symposium on Computers and Applications*, pages 877–883, 1987.

[7] L. M. Ottenstein K. J. Ottenstein. The program dependence graph in a software developement environment. *ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments*, 19(5):177–184, 1984.

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

[9] D. Binkley S. Horwitz, T. Reps. Interprocedural slicing using dependence graphs. *ACM Trans. Progr. Lang. Syst.*, 12:26–61, 1990.

[10] E. J. Weyuker S. Rapps. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, pages 367–375, 1985.

[11] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[12] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[13] Steven J. Zeil. Perturbation technique for detecting domain errors. *IEEE Transactions on Software Engineering*, 15(6), June 1989.