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: Belli, F., Illgen, T., Logic Representation of Programs to Detect Arithmetic Anomalies, Proc. 26th IEEE Euromicro Conference, Vol. 2, pp.216-223

Logic Representation of Programs to Detect Arithmetic Anomalies

Fevzi Belli, Thomas Illgen
Universität Paderborn
Fachbereich 14 — Elektrotechnik
Fachgebiet Angewandte Datentechnik
D-33095 Paderborn, Germany
{fb, ti}@adt.uni-paderborn.de

Keywords: Arithmetic Anomalies, Program Testing, Logic Programming, Program Transformation, Software Legacy and Re-engineering

Abstract

Much interest, especially by banks and insurance companies is paid to detect arithmetic anomalies and inexactness of arithmetic expressions. Numerous examples in the past show that although mathematical methods for correct implementation of arithmetic expressions exist and are well understood, many programs contain arithmetic anomalies, impreciseness or faults.

Software tests based on conventional coverage criteria [3] and functional tests are not well-suited for detection of these faults. The detection of arithmetic anomalies by these methods strongly depends on the adequateness of test-cases. The selection of effective test-cases needs a lot of effort to detect context-sensitive arithmetic inexactness.

In this paper we introduce a novel approach for detecting arithmetic anomalies. The method will be based on the specification of fault classes combined with the transformation of the program under test into a predicate logic model. The number of potential context-sensitive faults will be deployed as a criteria to precisely select modules in large software-systems to increase the test effectiveness.

1 Introduction

For a large scale of software products it is a fundamental requirement to provide correct arithmetic calculations. Even if algorithm for the correct implementation of arithmetic calculation exists [17, 12], a large scale of re-engineered programs as well as new implementations contain arithmetic faults or inexactnesses caused by errors of programmers or inexactness of compiler, arithmetic unit of the computer, a.s.o.. Most of these faults are well-known. They could be classified in fault classes. To define these faults is the first step to reveal them, no matter which method will be used. 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 [19]. These are classifications oriented on the affects in the control-flow or output of the program. For detecting classified faults it is useful to build fault oriented classes, e. g. classes who build a upper set of faults. In this case it is possible to analyze the fault structures for typical common attributes. This enables an easier detection of a large scale of faults by detecting common attributes of a class first. Following some examples for fault classes:

- **Faults caused by numerical overflow and underflow:**
  An overflow condition occurs, when the result of numerical calculation is too large for the program to handle. Underflows occur in floating point calculations by inaccuracy in the fractional part.

- **Wrong order of operators:**
  These is a class of errors based on the priority of operations. For example, if ** represents exponation, so 2 ** 3 is two cubed. Is 4 * 2 ** 3 equal to 36 or to 512? Another kind of fault of this class occurs by function calling. The execution of the following example is non deterministic: $f(x, y) + g(y)$. If variable y is manipulated in function f or g and if it is a call by reference, the result depends on the sequence of derivation.

- **Faults by the lack of precision in the calculation of arithmetic expressions:** E. g. the calculation of the following example $z = \frac{x+y}{2}$ with *float x=2.4, float y=2.4, float z* derives 2.4000000953674316 on the used machine and compiler.

- **Faults caused by successive executions, even if mathematical rules are correct implemented, e. g.**

rules over distribution, association, ...: Even if mathematical rules are correct implemented, faults could occur because of the limited exactness of number representation, e. g. the result of the following equation could depend on the sequence of execution $\frac{x*1000}{y}$. If y is a large value and the division is executed before the multiplication, in some cases the division cause an underflow. The multiplication with a big factor brings the inexactness in the derivation.

Detecting these faults in the program implementation needs considerable amount of effort. Conventional Software-test methods which are provided by commercial test tools cannot reveal these faults. The detection depends strongly on the selection of test cases, which produce a marginal value at a critical point of the program. An effective way to detect arithmetic anomalies are code reviews [4]. The disadvantage of this method is the great effort and the lack of automation.

Proving by formal methods also symbolic execution appears to be the ultimate technique for producing reliable programs. Unfortunately the practical accomplishments in this area fall short of a tool for routine use [11]. Especially for detecting potential context-sensitive faults, which often causes arithmetic failures, cannot be detected by these methods. Such faults endanger Software, especially embedded in safety-critical and capital intensive systems. Once these potential faults are defined by a formal specification, the program could be analyzed to check the presence of these anomalies. Therefore, it is necessary to find a useful, formal representation of the program to be analyzed. By analyzing the program directly, for each potential fault the parser must be prepared. The effort to realize this is considerable high. A vital requirement to solve this problem is to model the program in a practicable way before analyzing. This paper presents our approach to detect defined arithmetic anomalies by transforming the program under test in predicate logic.

2 Overall Structure

The critical program point is defined by the definition of the potential fault. E. g. it is a program point were a potential fault or error could affect the control-flow or output of the program. In this case it produces a failure. As critical program point we define *CPP(v,l,Def)*, consists of a critical node or statement at location l for a variable v based on a specific fault definition *Def*. Of course we cannot generate test cases automatically which produces a marginal value for v at location l, but we can reduce the program complexity by building static slices and find dependences of all statements in the program that affect the current value of a variable v at location l for a potential fault definition *Def*. With this data we could build a static slice *SPS(P,v,l,Def)* consists of all statements in the given program P that effect the variable v at location l for a given fault definition *Def*. We define a Program P with its control-flow graph and its data-flow. The control-flow is a set of directed edges E to describe the possible transfer of control from a node n_i to a node n_j, with $n_i, n_j \in N$ and N is the set of all program nodes. 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. Now we could define a set of critical paths $CP(S, CPP(v, l, Def))$ were $S = (n_{s1}, n_{s2}, .., n_{sk})$, $k \in N \cup CPP(v, l, Def)$ describes a vector of starting nodes and the critical node $CPP(v, l, Def)$ is the exit node.

2.1 Predicate Logic

We use first-order predicate logic to model the program under test in PROLOG-clauses. Logic programs consist of a finite set of definite 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 [5]. 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 [18, 16]. 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 [10]:

- 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 [18, 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 used or defined at p [14]

The computation of static slices is shown as a reachability problem in a program dependence graph [9], i. e. data dependence and control dependence. Weiser computes slices with respect to a set of data flow equations [18]. 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, 6]. 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 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

```
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).

input. In this paper we use static slicing to reduce the complexity of the program. Therefore we use the static slicing definition of Weiser [18], e. g. we build executable static slices. The difference of our approach is, that our slices depend on the definition of a fault to be detected. In case of our approach it could be useful to manipulate the slice of the program to verify the expected output with the actual output. 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, CPP(v,l,Def))*. A static slice includes the critical path, if all variables affect v at location l directly. In this case $C = (p, v) \subseteq$ *CP(S, CPP(v,l,Def))* for a potential fault Definition *Def*. It could be useful to handle the definition in this way for considering pointer and structures. Before explaining our slicing technique on an example, we give a brief instruction to program modeling based on predicate logic.

3 Program Modeling in Logic Structures

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 [15, 7]. 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>).
```

```
1 main() {
2  float x, y, average;
3  while(scanf("%f %f", &x, &y)>0) {
4   if(x < 1.0 || x > 5.0) {
5    printf("Wrong input of mark x %f", x);
6   } else {
7     if(y < 1.0 || y > 5.0) {
8     printf("Wrong input of mark y %f", y);
9     } else {
10      average = (x + y)/2;
11      if(average <= 2.4) {
12       printf("Give Admission\n");
13      } else {
14        printf("Admission declined\n");
15        }
16      }
17    }
18  }
19 }
```

Figure 3. An example program which calculates the average of two scores of a student.

The second argument in `for`-loops contains a list of the three arguments.

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

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

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

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.

Example
Following we generate a model for the example program in figure 3, written in C. It calculates the arithmetic average of two scores of a student. The data flow of this program could be modeled in following clauses:

```
declaration(2, float, [x, y, average]).
input(x, y).
d_use(average, 10, average = (x + y)/2).
c_use(x, 10).
c_use(y, 10).
p_use(scanf, 3, scanf > 0).
p_use(x, 4, x < 1.0 || x > 5.0).
p_use(y, 7, y < 1.0 || y > 5.0).
p_use(average, 11, average <= 2.4).
```

The directed edges model the possible control flow of the program. It only contains the edges of a control flow graph and does not respect predicates in decisions. The potential possible control flow of the program in figure 3 has the structure below.

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

The third type of rules coupling the data flow with the control flow and representing the program instructions are outlined in the following PROLOG-notation:

```
func(1, main).
loop(while, 4, [input, p_use(scanf)]).
decision(4, if, [p_use(x)]).
state(5, [output]).
decision(6, else, [p_use(x)]).
decision(7, if, [p_use(y)]).
state(8, [output]).
decision(9, else, [p_use(y)]).
state(10, [d_use(average), c_use(x),
      c_use(y)]).
decision(11, if, [p_use(average)]).
state(12, [output]).
decision(13, else, [average]).
state(14, [output]).
end(15, decision, 13).
end(16, decision, 9).
end(17, decision, 6).
end(18, loop, 3).
end(19, 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. The lack in this case is the direct address calculation. This could not be handled using this model. The functionality is limited by address calculations of arrays or strings, where the address of each element n directly follows the address of element $n-1$. Finally 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 of the variable name. In the model structure we replace variable names in the presence of pointer with the fact defined above.

4 Detecting Arithmetic Anomalies

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. Generate a predicate logic model of the program under test.
3. Select potential faults in the program under test.
4. Find critical paths in the program.
5. Build static slices containing all dependencies of the variable v at location l for a given fault definition *Def*.

The representation uses first-order predicate logic clauses suited for automatic processing with PROLOG. This has several advantages e.g. portability for different programming languages and flexibility in building program-slices [16]. The parser is the well-defined interface between the imperative program and the logic structure. The definition of classified faults in form of PROLOG-clauses has the character of a specification.

Static analysis for arithmetic faults is performed using the clause representation of the program in connection with a database containing classification of arithmetic faults. The analysis is started by the call of a PROLOG-rule which declares the fault class. Depending on this fault class, slices

for a specific variable under test will be extracted containing the manipulations of the data-flow of this variable. In connection with the control-flow of the considered variables it is possible to automatically generate hints to potential arithmetic anomalies. The test-report lists the paths of the detected anomalies. The number of detected potential faults is a criteria to select modules in large software-systems to increase the effectiveness of tests in case of correctness of calculations. Figure 1 shows the general method to detect specified arithmetic anomalies.

Before demonstrating the methodology on a small example, in following the program modeling and the representation in predicate logic will be explained and depicted by examples.

Consider the example in figure 3. It contains a potential fault of the class *lack of precision in the calculation of arithmetic expressions*. We assume that the average must be under 2.4 to succeed a test.

The test-run with input values $x = 2.4$ and $y = 2.4$ derives the average of 2.4000000953674316 on the used compiler. This inexactness in calculation of the average declines the admission of the student. In this case the critical node is the decision in line 11. We define the potential fault in the following five definitions, which are effortless to implement in PROLOG-clauses. The definition is of the class *lack of precision in the calculation of arithmetic expressions*. We define this class as *Def = dp*.

Definition of the fault class *Def = dp*:

Let D be the set of all clauses to describe the data-flow and E all clauses to describe the possible edges.

Definition 1 *Tr is the* closure *of the variable x, when* $Tr(x) = \{d\_use(V, \_, \_) \in D | V = x\} \cup \{p\_use(V, \_, \_) \in D | V = x\}$.

Definition 2 *A* path *from a_1 to a_n is a sequence* $path(a_1, a_n) = (a_1, a_2, ..., a_{n-1}, a_n)$ *with a_1 as starting node and a_n as end node, if there exists a set of edges* $edge(a_1, a_v), ..., edge(a_w, a_n) \in E$. *A* $path(a_1, a_2)$ *is* cycle-free, *if its elements are disjunct.*

Definition 3 *The set of* predicative data-flow paths *for a definition* $d\_use(x, n, \_)$ *of a variable x is the set of* $PD(x, n) = \{P_1, ..., P_k\}$ *at which P_i is a cycle-free path from n to m with* $p\_use(x, m, \_)$.

Definition 4 *An arithmetic expression is called* complex, *if it contains a minimum of one arithmetic operation.*

Definition 5 *A potential fault $C(x)$ exists in a program, if* $C(x) = \{d\_use(x, n, T) \in Tr(x) | T$ *is a complex arithmetic expression* $\wedge\ PD(x, n) \neq \emptyset\}$.

```
 1 main() {
 2   float x, y, average;
 3   while(scanf("%f %f", &x, &y)>0) {
 4     if(x < 1.0 || x > 5.0) {
 5
 6     } else {
 7         if(y < 1.0 || y > 5.0) {
 8
 9         } else {
10             average = (x + y)/2;
11             if(average <= 2.4) {
12
13             }
14
15
16           }
17       }
18   }
19 }
```

Figure 4. Static Slice for the example program in figure 3.

In this case the critical node *CPP(v,l,Def)* is the decision in line 11 for variable *average* with respect to the fault definition *Def = dp* (*CPP(average,11,dp)*). The critical path is *CP(S, CPP(average,11,dp))*, with $S = (10, 11)$. The example in figure 4 only contains one critical path of the defined potential fault. The number of possi ble path is finite, because we are looking for cycle free paths. We generate the static slice in following steps:

1. Detect the type of the variable to be analyzed. In the example real numbers are critical. To detect potential faults of the defined type with respect to variable *average*, we detect the declaration fact:
   `declaration(2, float, [x,y,average]).`

2. Detect the critical path and the critical node respecting a given definition of a potential fault. In the example, we first have to look for the p-use and d-use of a variable. The p-use of the variable is the critical node. The definition of the variable must contain an arithmetic expression on the right side of the equation. This is the reason for inaccuracy. After that we have to detect all possible directed edges from the definition to the predicate use of the variable. These are the critical paths. If a critical node and a critical paths exist, we have detected a potential fault of the given definition. The facts defining the critical path of the example are:
   `d_use(average,10,average=(x + y)/2).`
   `p_use(average,11,average<=2.4)`

3. Detect the dependences on the critical paths. Depen-

dences are variables which affect the control flow or data flow from the definition of the variable to the predicative use in this case. (Here: *Dep(CP)* = {})

4. For each critical path, find the computing variables. In this case variables x and y.
   `c_use(x,10) c_use(y,10)`

5. Detect the possible paths of the affecting variables to a node of the critical path. In the example we find the paths $P(x) = (3, 4, 6, 7, 9, 10)$ and $P(y) = (3, 4, 6, 7, 9, 10)$.

6. Connect the critical path and the path of dependecies in the right order.

7. Outlining the statements and nodes on the found paths generates a static slice for each critical path.

Figure 4 shows the slice respecting the critical path. In the presence of several critical paths, we generate a static slice for each critical path, if the new slice differs from existing slices. The generated slices are executable. The generated slices turns the testers eyes toward to critical program parts and helps him to select efficient testcases. The testcases are executable on the slice. Therefore we add an input and output statement in the slice to provide testinputs and to verify the actual result with the expected results.

A testreport supports the tester to construct testcases. The testreport contains expressed data selected by analysing the program under test. The data below shows the standart output of a testreport for the example program.

- Number potential faults of a defined fault class.

  ```
  Potential faults of type: dp: 1
  ```

- Description of initiation of the critical path.

  ```
  expression: average = (x + y)/2
  line 10
  ```

- Description of the critical node.

  ```
  critical node: average <= 2.4
  ```

- Description of the critical expression.

  ```
  expression: (x + y)/2 = 2.4
  ```

- Description of the relevant variables used in the arithmetic expression.

  ```
  definitions: x input, y input
  ```

- List of predicates in decisions on cycle-free paths from the definition of the variable to the predicative use of the variable.

  ```
  variable x: x < 1.0 || x > 5.0   Line 4
  variable y: y < 1.0 || y > 5.0   Line 7
  ```

- Conditions to reach the predicative use of the variable on paths $Dep(CP)$.

  ```
  Conditions on critical path:
  ```

5 Related Work

Most of the popular approach to build slices are based on graph oriented methods. Slices are build for a node at a specific location for a specific variable. In our approach we select these data based on fault definitions. Therefore we generate a knowledge base containing classified faults. We extract common features of each fault class to reduce the effort for fault detection and to detect a large group of faults. The fault definition is not a program specification, it is a data base containing the knowledge about potential arithmetic faults. Based on these faults we detect critical nodes and critical paths. The detected critical parts of the program are sliced to reduce the program complexity and to help the tester in finding efficient testcases to reveal if a detected potential fault could cause a failure.

A program model based approach for generating static slices is presented. Slices are extracted, containing all relevant parts of the program to detect arithmetic faults based on a fault specification. The extracted slices differ lightly from the popular slice presentations. DeMillo, Pan and Spafford [13] present a method for critical slicing for fault localization in a debugging process. They have shown that critical slicing provide a cost-effective method of significantly reducing the search domain. In their study the reduction from the whole program was around 64% and 80% of their obtained critical slices contain faulty statements. Their method is based on dynamic slicing for given testcases.

We use static slicing to reduce the program complexity. Our method is based on a program model and a data base for potential fault definitions. In the presented work, the extracted slices are depending on these fault definitions. The data selection from the predicate logic model is effortless because it has the form of a specification. In the present work we use our knowledge about arithmetic anomalies to extract critical program slices. The method is effective in all kinds of generating program slices containing context sensitive faults, anomalies or data dependences.

6 Concluding and Future Work

Detecting arithmetic anomalies, inaccuracy and faults is still an effort work. A lot of programs do not comply the requirements for calculations. In this paper we have presented an approach for static slicing based on a predicate

logic program model. We complete the model using pointer and functions. We have to complete the requirements for static slices of programs containing complex and dynamic data types. The presented method is in our previous work used to count the number of potential faults in large program systems to find arithmetic critical program modules. Especially banks and insurance companies pay a lot of interest in detecting arithmetic inaccuracy. Slicing is an efficient way for reducing the program complexity and to help the tester detecting critical program parts. Detecting arithmetic inexactness or faults strongly depend on the selection of test cases. Most failures are produced by marginal values at a specific program point. The detection of this values takes a lot of effort. We cannot generate testcases to produce a marginal value at a specific location in the program but we can locate potential critical program slices.

Program modelling using an executable predicate logic is a very efficient method for selecting data. It is also a well defined interface for different program languages.

In future work we have to examine if it is possible to generate effective testcases automatically to determine if a potential fault produces a failure in some cases. Additionally we have to describe exactly, which data we need for these testcase generation and how efficient the method could be. Therefore it is maybe possible to adopt existing methods to the special requirements of our approach adding heuristics of the program model. We also have to examine if the presented method is useful to generate dynamic slices for iterative generation of effective testcases.

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] Fevzi Belli. Methoden und Hilfsmittel für die systematische Prüfung komplexer Software. *Informatik-Spektrum*, 21:337–346, 1998.

[4] Radu Crisan. *Towards Automation of Code Revies Based on Checklists*. Dissertation, Universität-GH Paderborn, 1998.

[5] 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.

[6] 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.

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

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

[9] 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.

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

[11] James C. King. Symbolic execution and program testing. *Comunications of the ACM*, 19(7):385–394, July 1976.

[12] Ulrich Kulisch, editor. *Wissenschaftliches Rechnen mit Ergebnisverifikation*. Friedr. Vieweg & Sohn, 1989.

[13] E. H. Spafford R. A. DeMillo, Hsin Pan. Critical slicing for software fault localization. *Proceddings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–133, 1996.

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

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

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

[17] Willard L. Miranker Ulrich W. Kulisch. *Computer Arithmetic in Theory and Practice*. Academic Press, 1981.

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

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