PROLOG & TXL: A CASE STUDY FOR PROTOTYPING STRUCTURAL TESTING SUPPORTING TOOLS

Adenilso da Silva Simão
Tatiana Sugeta
José Carlos Maldonado
Maria Carolina Monard

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
São Carlos — São Paulo — Brazil
P.O. Box 668 ZIP Code 13560-970
Fone: ++55 (16) 273-9669 Fax: ++55 (16) 273-9751
{adenilso, tatiana, jcmaldon, mcmonard}@icmc.sc.usp.br

ABSTRACT

Structured testing criteria are usually used to assess the adequacy of test case sets, defining coverage measures. Control and data flow based criteria employ information about the program graph as well as definition and usage of variables to establish the testing requirements. In this paper, we present an approach to prototype supporting tools for control and data flow based criteria. In the proposed approach, we use TXL — a language based in the transformational paradigm — to analyze and instrument the program under test. The instrumentation aims at making it possible to process the data by a Prolog program which allows the tester to assess the test case set adequacy. A simple example is used to illustrate the main ideas of our approach.

Keywords: Testing Criteria, Testing Tool Prototyping, Prolog, Transformational Paradigm, Pascal.

1. INTRODUCTION

The establishment of testing criteria is fundamental to achieve a systematic and high quality testing activity, causing a positive impact in the quality of the

released software products. In the last decades several testing criteria have been proposed, particularly data flow based criteria [6, 9, 15].

The use of any testing criterion in real production environments depends on its cost/benefit tradeoffs. In this context, supporting tools are essential to aid the accomplishment of empirical studies, aiming at analyzing these tradeoffs. However, the development of supporting tools is a costly and time consuming activity. Therefore, providing rapid prototyping mechanisms to ease the implementation of a prototype tool is relevant in this scenario. Although its performance may be compromised, a prototype is easier to build, maintain and evolve. Moreover, the prototype tool can be useful as an *oracle*, in the sense discussed by Weyuker [18], during the actual implementation.

In this work, we present an approach to assist the prototyping of supporting tools for structural testing, and illustrate its application by prototyping a data flow based tool for Pascal. The approach, named ProTesC — **Pro**log for Software **Tes**ting based on Structural **C**riteria, is based on code instrumentation using the transformational paradigm [10] and analysis of the test case set adequacy using Prolog procedures. The instrumentation permits to obtain the required information about the program execution. We employed the instrumentation model presented in [9] and implemented it using TXL [3]. We aim at obtaining a generic description for instrumentation that could be adapted to other languages, as discussed in [14].

Prolog is a logic programming language which has features that make it adequate to describe the testing criteria requirements in a flexible and abridged way. We can verify the adequacy of a test case set in relation to a given criterion by defining Prolog procedures, i.e., a set of clauses about the same relation (or predicate), that calculate and check the satisfaction of the criterion requirements. This mechanism is usually simpler than developing a full tool. Additionally, if we consider a proposal of new criteria, the use of a prototyping language like Prolog facilitates the tuning of the definition and the implementation of the underlying criteria.

To illustrate, the ProTesC approach has been instantiated to support structural testing of Pascal programs, building a prototype named ProTesC/Pascal. We also have developed Prolog procedures to assess the adequacy of test case sets according to data flow based criteria defined by Rapps and Weyuker [11].

In this paper we are not concerned on the usage and applicability of these criteria but rather on supporting experiments with them, easing their empirical evaluation. The main ideas of our approach are illustrated using the All-Uses data flow based criterion [11].

Although Pascal is a bit old-fashioned language, it is a well known language and is widely used in academy for teaching purposes, making it suitable to illustrate our approach. Notwithstanding any other procedural language, such as C, Fortran or

Cobol, could be used as well to instantiate the ProTesC approach.

The structure of this paper is as follows: in Section 2 we present basic concepts, terminology and definitions related to the All-Uses criterion; in Section 3 we present the ProTesC approach, discuss its general architecture and illustrate its instantiation with Pascal. Also, the operational aspects of ProTesC/Pascal prototyping is briefly described. In Section 4 related work is presented. Finally in Section 5 we draw some concluding remarks about this work.

2. BASIC CONCEPTS

In this section we present the main concepts of, as well as a brief introduction to, program graph and testing requirements used in this work. More details can be found in [11].

A *statement block* is a sequence of statements that are *always* executed in sequence in a program. When the first statement of a block is executed, so are the remaining statements; branching only occurs either to the beginning or from the end of the block.

$G = (N, E)$ is a *program graph* of a program P if for each statement block of P there exists a node $n \in N$ and for each possible control transfer between a block represented by n_1 and a block represented by n_2 there exists an edge $(n_1, n_2) \in E$. In a program graph, the node corresponding to the statement block whose first statement is the first statement of the program is denominated *start node*. Conversely, the node corresponding to the block whose last statement is the last statement of the program is denominated *exit node*.

The program graph can be enhanced with information obtained by analyzing the data flow of the program. We associate to the nodes of the graph information about the assignments (*definitions*) of variables and their usages in computation (computational-usages, or *c-usages*). We associate to the edges of the graph information about the usage of variables in the expression that rule the control flow of the program to that edge (predicative usages, or *p-usages*). This enriched graph is denominated *def-use graph* [11]. Figure 1 presents a Pascal program[1] to calculate $z = x^y$ and the corresponding def-use graph.

Each execution of a program induces a path in the graph that corresponds to the blocks traversed during the execution. Considering a def-use graph G, a path or subpath $\pi = (n_i, \ldots, n_k)$ is *definition clear w.r.t. a variable* x if x is not (re-)defined in π (except eventually in the last node of π). A path is *simple* if all nodes, except possibly the first and last, are distinct. A path is *complete* if its initial node is the start node and its final node is the exit node.

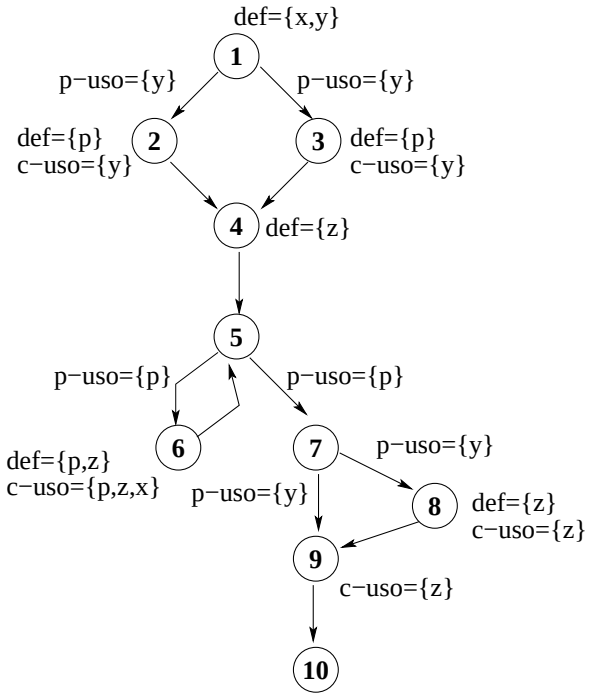
A triple (n, n_1, x) is a *definition-c-usage association* in G if there exists a definition of x in n, a computational usage of x in n_1 and a path π from n to n_1 that is definition clear w.r.t. x.

1. This program is a Pascal version of the pseudo-code used by Rapps and Weyuker [11].

POWER.PAS

```
{1}  begin
{1}    readln(x);
{1}    readln(y);
{1}    if y > 0
{2}      then p:=y
{3}      else p:=-y;
{4}    z := 1;
{5}    while p <> 0 do
{6}      begin
{6}        p := p - 1;
{6}        z := z * x;
{6}      end;
{7}    if y < 0
{8}      then z := 1/z;
{9}    writeln(z);
{10} end.
```

(a)



(b)

Figure 1: (a) A Pascal program to calculate $z = x^y$ and (b) the corresponding def-use graph [11]

Conversely, a triple $(n, (n_1, n_2), \mathsf{x})$ is a *definition-p-usage association* for G if there exists a definition of x in n, a predicative usage of x in (n_1, n_2) and a path π from n to (n_1, n_2) that is definition clear w.r.t. x. A path π executes an *association* (n, n_2, x) or an *association* $(n, (n_1, n_2), \mathsf{x})$ if there exists a subpath $\pi' = (n, \ldots, n_2)$ in π such that π' is definition clear w.r.t. x.

Based on these concepts, Rapps and Weyuker [11] defined a family of data flow testing criteria. Next, we present the definition of a subset of these criteria which are implemented in ProTesC/Pascal. Considering a def-use graph G and a set Π of complete paths in G, then:

All-Nodes: Π satisfies the criterion if every node of G is included in at least one $\pi \in \Pi$;

All-Edges: Π satisfies the criterion if every edge of G is included in at least one $\pi \in \Pi$;

All-Uses: Π satisfies the criterion if for every node n and every variable x defined in n, at least one $\pi \in \Pi$ includes *at least* one definition clear

path w.r.t. x from n to every node and every edge that contains a usage of x;

In this paper, we will use the All-Uses criterion to illustrate both the Prolog procedures (Section 3.2) and the operational aspects of the ProTesC/Pascal prototype (Section 3.3).

3. ProTesC APPROACH

ProTesC is a prototyping approach that supports the development of tools for adequacy analysis based on structural testing criteria. The approach is illustrated in Figure 2 and can be characterized by two parts. In the first part, the program (`prog`) is instrumented using the system TXL. The instrumentation schema has been adapted from the one proposed for POKE-TOOL [1, 9]. The instrumentation creates another file, called `instrum`, that in addition to the functionalities presented in `prog` writes in the file `paths.pl` Prolog facts corresponding to the paths executed for the test cases. Moreover, during the instrumentation phase, a set of facts related to the def-use graph is collected and stored in the file `facts.pl`.

In the second part of the approach, we use the Prolog program `test.pl`, as well as the bases `facts.pl` and `paths.pl`, generated in the first part, to allow the tester to assess the adequacy of the test cases w.r.t. the testing criteria. Section 3.1 presents more details about the instrumentation phase and Section 3.2 discusses the Prolog program `test.pl`. The operational aspects of ProTesC/Pascal are presented in Section 3.3.

3.1. Transformational Paradigm: Instrumenting

TXL is a transformational programming paradigm language, in which a program is composed by two main components: a grammar and a set of transformation rules. The grammar describes how the input program (considered as a stream of *tokens*) is to be analyzed and converted into a syntax tree, by means of a process called *parsing*. Then, the set of transformation rules is applied to this tree in order to perform transformations in its structure. Usually, a transformation rule consists of a pattern and a substitution model. When, and if, the pattern is found in the tree or in a subtree, the rule is applied by exchanging the (sub-)tree by the substitution model. Moreover, actions can be triggered whenever a substitution takes place. After the transformation phase, the resulting tree is converted into a stream of tokens by traversing the tree and collecting the leaf tokens. Figure 3 illustrates this process. The context free grammar of the language [12] is furnished in the file `grm` in a notation equivalent to BNF [16]. The transformation rules are in the file `rules.txl` and instrument the code by introducing, in the syntax tree, subtrees corresponding to the statements in the given language that will log the program execution.

Moreover, the rules in this file collect information about the structure of the def-use graph and store this information

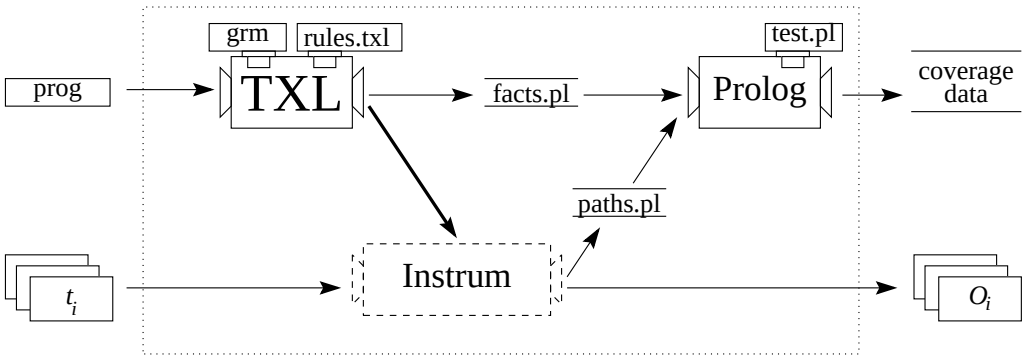


Figure 2: ProTesC Overall Structure

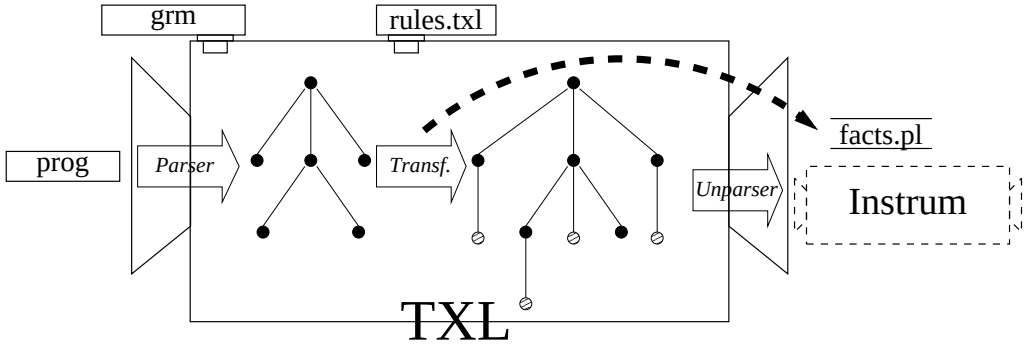


Figure 3: Code Instrumentation Using TXL

```
1  rule process_if_then_else curr_node [id] after_node [id]
2     replace [statement_list]
3        'if e [expression] 'then thenstm [statement]
4                           'else elsestm [statement]
5     construct then_node [newnode]
6     construct else_node [newnode]
7     where
8         e       [process_expression curr_node after_node]
9     where
10        thenstm [process_statement  then_node after_node]
11    where
12        elsestm [process_statement  else_node after_node]
13    by
14       checkpoint '( curr_node ') ';
15       'if e 'then
16         'begin
17           checkpoint '( then_node ') ';
18           thenstm
19         'end
20       'else
21         'begin
22           checkpoint '( else_node ') ';
23           elsestm
24         'end
25       checkpoint '( after_node ') ';
26 end rule
```

Figure 4: TXL code sample (for Pascal)

in the file `facts.pl`. Figure 4 presents a sample of a TXL rule used to instrument a Pascal *if-then-else* statement. Lines 2 to 5 declare the pattern to be looked for in a Pascal program. Lines 13 to 25 define the substitution model. Note that Lines 14, 17, 22, and 25 in the substitution model include the checkpoint statements that are required to register the execution of the *if-then-else* statement. In Lines 5 and 6 new graph nodes are declared for representing the *then* and *else* statements, respectively. Lines 8, 10 and 12 invoke rules to process the control expression and the *then* and *else* statements, respectively. It can be observed in this example that the TXL code is grammar-oriented and eases

the accomplishment of analyses based on the language grammar structure.

Figure 5 illustrates the instrumented file of the program in Figure 1(a). It can be observed that statements were included in order to indicate the execution of each block, as discussed in [9]. The statement `checkpoint(`n`)` writes n in the file `paths.pl`, whereas the statements `output_path_init` and `output_path_finish`, respectively, opens and closes this file. For example, `checkpoint(6)` represents the beginning of block 6.

Figure 6 shows some facts generated during the instrumentation of the program in Figure 1(a) which are stored in the fact base `facts.pl`. The predicates that can be included in the fact base are:

- `node/1` to represent a node in the program graph. For example, `node(`1`)` indicates that node 1 belongs to the program graph.
- `edge/2` to represent the edges in the graph. For example, `edge(`$1,2$`)` indicates that edge $(1, 2)$ belongs to the graph.
- `definition/2` to represent variable definitions. Thus, `definition(`1`,x)` indicates that variable `x` is defined in node 1.
- `c_usage/2` to indicate computational usages. For example, `c_usage(`2`,y)` indicates that there is a computational usage of `y` in node 2.
- `p_usage/3` to indicate a predicative usage. For example, `p_usage(`$1,2$`,y)` indicates that there is a predicative usage of `y` in edge $(1, 2)$.

INSTRUM.PAS

```
begin
  output_path_init;
  checkpoint (1);
  readln (x);
  readln (y);
  if y > 0 then
    begin
      checkpoint (2);
      p := y
    end
  else
    begin
      checkpoint (3);
      p := - y
    end;
  checkpoint (4);
  z := 1;
  checkpoint (5);
  while p <> 0 do
    begin
      checkpoint (6);
      p := p - 1;
      z := z * x;
      checkpoint (5)
    end;
...
  checkpoint (10);
  output_path_finish
end.
```

Figure 5: Partial Instrumented Program

3.2. Prolog Procedures

Having the base of facts related to the def-use graph of the program and to the

FACTS.PL

```
node(1).
node(2).
node(3).
node(4).
...
edge(1, 2).
edge(2, 4).
edge(1, 3).
edge(3, 4).
...
definition(1, x).
definition(1, y).
definition(2, p).
definition(3, p).
definition(4, z).
...
c_usage(2, y).
c_usage(3, y).
c_usage(6, p).
c_usage(6, z).
c_usage(6, x).
...
p_usage(1, 2, y).
p_usage(1, 3, y).
p_usage(5, 7, p).
p_usage(5, 6, p).
...
```

Figure 6: File `facts.pl`: Examples

paths executed by a given set of test cases, the tester can evaluate, through defined Prolog[2] procedures, the coverage of the test cases w.r.t. a given criterion, in this case, the Rapps and Weyuker's criteria. Several procedures were defined for each criterion. All of them use information held in file `paths.pl`.

In Figure 7 we illustrate the procedures defined for the All-Uses criterion, most of them consisting of only one clause. In particular, we discuss the predicates related to computational usages (Figure 7(a)). Similar predicates were created for predicative usages (Figure 7(b)). Observe that the combination of both predicative and computational usages characterizes the All-Uses criterion. The associations are calculated by procedure `calcDefCUsage` and are stored as a predicate `defCUsage/3`. Procedure `dcu(Var, N, N1)` checks whether there exists an association involving the definition of `Var` in node `N` and its computational usage in node `N1`. This is accomplished by verifying the existence of at least one definition clear path from `N` to `N1` w.r.t. `Var`. Procedure `executedCUsage(Node, N, Var)` determines whether an association involving the definition of `Var` in node `Node` and its computational usage in node `N` has been executed. This procedure also verifies whether there exists an executed path π in `paths.pl` such that

1) `Node` is in π,
2) `N` appears after `Node` and
3) there is no redefinition of `Var` in the subpath from `Node` to `N`.

2. We have used the LPA-Prolog system version 3.5 [13, 17]; only standard predicates were employed.

TEST.PL

```
calcDefCUsage :-
   definition(N,Var),
   dcu(Var,N,N1),
   assertz(
      defCUsage(N,N1,Var)
   ),
   fail.

dcu(Var,Node,N):-
   c_usage(N,Var),
   once(
      clearDefPath(Node,N,Var)
   ).

executedCUsage(Node,N,Var):-
   path(P),
   append(_,[Node|Y],P),
   append(Z,[N|_],Y),
   not (member(Nd,Z),
      definition(Nd,Var)
   ),!.
```

(a)

```
calcDefPUsage :-
   definition(N,Var),
   dpu(Var,N,(N1,N2)),
   assertz(
      defPUsage(N,(N1,N2),Var)
   ),
   fail.

dpu(Var,Node,(N1,N2)):-
   p_usage(N1, N2, Var),
   once(
      clearDefPath(Node,N1,Var)
   ).

executedPUsage(Node,(N1,N2),Var):-
   path(P),
   append(_,[Node|Y],P),
   append(Z,[N1,N2|_],Y),
   not (member(Nd,Z),
      definition(Nd,Var)
   ),!.
```

(b)

```
clearDefPath(Node,J,Var):-
   clearDefPath1(Node,J,Var,[Node]).

clearDefPath1(B,J,_,_):-
   edge(B,J).
clearDefPath1(Node,J,Var,L):-
   edge(Node,N1),
   not definition(N1,Var),
   not member(N1,L),
   clearDefPath1(N1,J,Var,[Node|L]).
```

(c)

Figure 7: Prolog Procedures for Checking (a) Computational Usages; (b) Predicative Usages; and (c) Definition Clear Path

3.3. ProTesC/Pascal: Operational Aspects

After instantiating ProTesC with the Pascal language, we obtained the ProTesC/Pascal prototype. From the tester's viewpoint, ProTesC/Pascal is composed by two different levels. In the first level, the tester uses TXL to generate the instrumented program and the base of Prolog facts related to the def-use graph. This is done by executing TXL with the source program, the file with the transformation rules and the file with the grammar. In the example, TXL is run with the source file `power.pas`. The transformation rules and the grammar are furnished in the files `pas.txl` and `pascal.grm`, respectively. TXL will generate the instrumented program `instrum.pas` and the base of facts `facts.pl`.

The instrumented program is then compiled and run with the test cases. Since the instrumented program has essentially the same functionality of the source program, the tester can verify whether the resulting output is in accordance with the program specification. Besides generating the outputs, the instrumented program writes in the file `paths.pl` a fact with the path traversed in the graph during the execution. Table 1 presents the actual output, the expected output and the traversed path for two test cases applied to the program in Figure 1(a).

To evaluate the adequacy of the test cases, the tester loads the files `test.pl`, `facts.pl` and `paths.pl` in a Prolog session. Then, the tester checks the adequacy by means of consults made through the procedures in `test.pl`. For example, using the procedures in Figure 7, the tester can execute the following consult to verify which associations have not been executed. Note that procedure `calcDefCUsage/3` should have been previously executed in order to determine the associations.

```
?- defCUsage(N, N1, Var),
 not executedCUsage(N,N1,Var).
  N = 3, N1 = 6, Var = p;
  N = 4, N1 = 8, Var = z;
  N = 6, N1 = 8, Var = z;
  N = 8, N1 = 9, Var = z;
no
```

To continue the test activity, test cases should be included to execute the associations that have not been executed yet. For example, to execute the association $(3, 6, \mathsf{p})$, we run the instrumented program with the test case $x := 2$, $y := -1$, whose result is $z := 0.5$ and traversed path is `path([1,3,4,5,6,5,7,8,9,10])`. After reloading `paths.pl` and redoing the consult, we obtain the following result:

```
?- defCUsage(N, N1, Var),
 not executedCUsage(N,N1,Var).
  N = 4, N1 = 8, Var = z;
no
```

In particular, the association $(4, 8, \mathsf{z})$ is non-executable, i.e., there is no value for the input variables of the program that leads to the execution of this association which can then be removed [11].

```
?- retract(defCUsage(4, 8, z)).
 yes
```

Table 1: Test Cases

Test Case t_i	Expected Result	Obtained Result	file `paths.pl`
$x := 2; y := 2$	$z := 4$	$z := 4$	`path([1,2,4,5,6,5,6,5,7,9,10]).`
$x := 2; y := 0$	$z := 1$	$z := 1$	`path([1,3,4,5,7,9,10]).`

To ease the usage by the tester, we also have defined several auxiliary procedures, e.g. to calculate the list of not executed associations (procedure `notExecuted-CUsages(L)`) and to calculate the percentage of executed associations (procedure `cUsageCoverage(N)`). For the remaining criteria of Rapps and Weyuker, similar procedures were defined.

4. RELATED WORK

Chaim [1] presents a tool, named POKE-TOOL, that supports the coverage analysis of data flow based criteria. Initially, POKE-TOOL supported C program testing based on the Potential-Uses family criteria [9]. Later, the tool was extended to support the testing of programs written in other programming languages, such as Cobol [8] and Fortran [4] and to support Rapps and Weyuker's data flow criteria [2, 11]. The instrumentation in POKE-TOOL is based on the instrumentation model defined in [9] and is done by converting the program to an intermediate language and the coverage is evaluated using specialized algorithms implemented in C.

From an abstract viewpoint, ProTesC can be considered a rapid prototyping mechanism when comparing to POKE-TOOL. Moreover, as argued in the final section, the cycle to introduce a new functionality is relatively shorter in ProTesC.

Herbert and Price present a strategy for coverage analysis and automatic test case generation based on structural criteria [7]. This strategy was implemented using Prolog in the environment LOGTEST, which supports validation of Pascal programs. Prolog procedures (written in DCG - *Definite Clause Grammar*) describe the Pascal grammar, constituting lexical and syntactic analyzers. Thereby, program information about data and control flow is obtained and stored to derive test cases. LOGTEST uses successive program executions, symbolic and/or real, to increase the program knowledge base and to aid the test data generation. On the other hand, in ProTesC the Pascal grammar is separately represented and handled by TXL, which is responsible for the program data analysis, instrumentation and data collection. Moreover, ProTesC uses only real executions.

In [5], Hamlet discusses the use of Prolog to create rapid mechanisms for prototyping testing tools, considering C programs and some of the criteria presented in [11]. From the viewpoint of the employed strategy, the proposal presented in this paper is similar to Hamlet's, in that the

code is also instrumented to make possible the coverage analysis of testing criteria by consulting Prolog procedures. However, the instrumentation in ProTesC is done by using the transformational paradigm, while Hamlet uses programs written using `yacc` and `lex` UNIX applications. The transformational paradigm in ProTesC permits a more abstract description of the instrumentation, so it can be easily adapted to different languages and criteria.

5. FINAL REMARKS

In this work we presented a prototyping approach, named ProTesC, that makes easier the generation of supporting testing tools for structural testing criteria. The approach was employed to instantiate a prototype for the Pascal language and data flow testing criteria. Studies involving other languages are required to evaluate the efficiency and applicability of this strategy.

In general, the most important characteristic of a prototyping mechanism is not the performance of the resulting prototypes, but rather the ease of implementing, maintaining and evolving them. With this respect, the case study conducted for instantiating the mechanism considering Pascal programs and data flow criteria indicates that the cost is relatively low, taking about three weeks of two graduate students' work to be accomplished. However, other experiments should be accomplished in order to obtain a more precise insight on the application cost of the mechanism.

The approach proposed in this paper is flexible and can be extended to support both other programming languages and other testing criteria. To support another language it is necessary to provide the language grammar description in TXL and to adapt the respective instrumentation transformation rules in order to properly instrument the code, as well as to generate the file `facts.pl`. In general, obtaining the grammar is not a hard task, since it is traditionally supplied with the language definition. Although there are not real data to be presented regarding this affirmation yet, we believe that adapting the transformation rules to another language would not require a significant effort. This point should also be explored by conducting some future empirical studies. Considering imperative languages (e.g. Pascal, C, Fortran, etc.), the control structures that determine the program graph topology do not essentially change, in spite of the sensible variations in its concrete syntax. For example, all languages have selection (*if-then-else*) and iteration (*while, for*) structures. Although each language possesses its own syntax, all are structurally equivalent variations. Note that by saying that "*if* statements in Pascal and C are structurally equivalent" we are neither considering the semantic of such structures nor their dynamic aspects. Rather, we consider the fact that in both languages an *if* will lead to a potential deviation of control flow and, consequently, to a new node in the program graph. Given a specific language, to support another test criterion it is

necessary the creation/adaptation of the corresponding Prolog procedures in file `test.pl`.

The reader should not be misled considering that by saying prototyping this activity would be a simple one. It would require from the prototyper (designer) expertise on the language and on the criteria, as well as on the prototyping mechanisms, including Prolog and TXL languages. Even so, the time and effort required would be quite less the traditional implementation way.

Forthcoming steps in this research are:

- to instantiate ProTesC to other programming languages, specially to languages commonly used in industry, e.g. C, C++ and Java, allowing to evaluate the efficiency of the approach.
- to include support for another testing criteria, (e.g. Potential-Uses family criteria [9]);
- to include a test case generation strategy based on information obtained from the base of facts; and,
- to create a graphical interface that integrates all procedures aiming to simplify the use of this approach.

ACKNOWLEDGEMENTS

The authors would like to thank the partial financial support from the brazilian funding agencies FAPESP, CNPq and CAPES and from Telcordia Technologies (EUA), as well as the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] M. L. Chaim, *Poke-Tool — A Tool for Supporting Data Flow based Structural Testing*. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, Brazil (1991). (in portuguese).

[2] M. L. Chaim, M. Jino, J. C. Maldonado and E. Y. Nakagawa, Poke-Tool: Current state of a tool for structural software testing based on data flow analysis. In *Tool Session of XII SBES - Brazilian Symposium on Software Engineering*. Maringá, PR, Brazil (1998). 37–45. (in portuguese).

[3] J. R. Cordy and M. Shukla, Practical metaprogramming. In *IBM Centre for Advanced Studies Conference*. Toronto, Canada (1992). 215–224.

[4] R. P. Fonseca, *Fortran Program Test Supporting in Poke-Tool Environment*. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, Brazil (1993). (in portuguese).

[5] D. Hamlet, Implementing prototype testing tools. *Software — Practice and Experience*, **1**(1), (1988), 1–4.

[6] M. J. Harrold and M. L. Soffa, Selecting and using data for integration testing. *IEEE Transactions on Software Engineering*, **8**(2), (1991), 58–65.

[7] J. S. Herbert and A. M. A. Price, Data test generation strategy based on symbolic and dynamic program

analysis. In *XI SBES - Brazilian Symposium on Software Engineering*. Fortaleza, CE, Brazil (1997). 397–411. (in portuguese).

[8] P. S. Leitão Jr., *Cobol Program Test Supporting in Poke-Tool Environment*. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, Brazil (1992). (in portuguese).

[9] J. C. Maldonado, *Potential Uses Criteria: A Contribution to Structural Software Testing*. Ph.D. thesis, DCA/FEEC/UNICAMP, Campinas, SP, Brazil (1991). (in portuguese).

[10] J. Neighbors, The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, **10**(5), (1984), 564–574.

[11] S. Rapps and E. J. Weyuker, Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, **11**(4), (1985), 367–375.

[12] A. Salomaa, *Formal Languages*. Academic Press, New York, 1973.

[13] R. Shalfield, *Win-Prolog User Guide*. Logic Programming Associates Ltd, London, England, 1997.

[14] A. S. Simão, A. M. R. Vincenzi, J. C. Maldonado and A. C. L. Santana, *Software Product Instrumentation Description*. Tech. Rep. 157, ICMC/USP, São Carlos, SP, Brazil (2002).

[15] H. Ural and B. Yang, Modeling software for accurate data flow representation. In *15^{th} International Conference on Software Engineering* (1993). 277–286.

[16] D. Vladimir, *Formal Languages and Automata Theory*. Computer Science Press, 1989.

[17] D. Westwood, *LPA-Prolog Technical Reference*. Logic Programming Associates Ltd, London, England, 1997.

[18] E. J. Weyuker, On testing non-testable programs. *Computer Journal*, **25**(4), (1982), 465–470.