IDEA GROUP PUBLISHING

701 E. Chocolate Avenue, Suite 200, Hershey PA 17033-1240, USA
Tel: 717/533-8845; Fax 717/533-8661; URL-http://www.idea-group.com

ITP4998

Potential Context Coverage

Boris Cogan
London metropolitan university, 166-220 Holloway Road, London N7 8DB, UK, Tel: +44 (0) 20 7314 4248,, Fax: +44 (0) 20 7753 7009, E-mail: b.cogan@londonmet.ac.uk

Tamara Matveeva
London metropolitan university, 166-220 Holloway Road, London N7 8DB, UK, Tel: +44 (0) 20 7314 4248,, Fax: +44 (0) 20 7753 7009, E-mail: t.matveeva

Nataliya Nikiforova
Institute for Automation and Control Processes, Russian Academy of Sciences, 5 Radio Str, Vladivostok, 690041, RUSSIA,
E-mail: nikif@iacp.dvo.ru

ABSTRACT

In this paper we present a new testing strategy that we are calling the potential context coverage *for program units written in algorithmic programming languages (both 'conventional' and object-oriented ones: Pascal, Fortran, C, Java, etc.). We introduce the notion of* context *and* potential context *for algorithmic programming languages with the aid of a model for a program fragment, and define a testing strategy in support of our definition of potential context coverage. A comparison of this strategy with traditional ones showed that it covers all of them except all path coverage.*

1 INTRODUCTION

Testing software units is a stage in *software coding and testing* activity during *software development* [1]. One category of testing strategies is *structural testing* (sometimes called *white-box* or *glass-box* etc. testing) where tests are derived from knowledge of the software structure and implementation [1-3]. Strategies of this category are applied individually to every software unit (here a procedure, a function, and a method of a class are considered as a program unit).

There are different testing strategies to cover the control-flow structure of a program unit (in terms of specific structural program models or in terms of programming languages). They differ by efforts required: a more exhaustive strategy requires a higher minimal number of test cases needed to satisfy the strategy for a given program unit, and hence this strategy tests the program unit 'better' than one required a less number of test cases [3]. In general, only a combination of testing strategies allows examining a program unit with a reasonable level of confidence. This usually means usage of several different measurement tools, which is impractical and does not applied in practice.

This paper presents a new testing strategy, the *potential context coverage* that takes into account both: the control-flow structure [3] and some information about program unit variables. It 'covers' all conventional coverage strategies except all path coverage. The strategy is rather simple to implement, and does not have apparent disadvantages.

2 THE IDEA OF THE STRATEGY

For a long time, people use the term *context* for their natural languages. According to [4], this term means '*the words before and after a word or passage in a piece of writing that contribute to its meaning*'. Linguists use more sophisticated definitions saying that *meaning* and *value* of any language unit have to be strictly defined in the context (taking into account even social, historical, and religious factors). To use a similar approach for program unit testing, we need to define strictly the meaning and values of programming language units and an approach to use them for program analysis. The so-called *language-oriented approach* allows doing it.

The *language-oriented approach to software measurement and assessment* (*LOA*) defines software elements/components and software measurement notions formally [5–10], and hence it is a sound basis for definitions of more sophisticated software testing strategies taking into account both, control and information flows. This paper uses notions of a *simple statement* and *program unit execution trace* defined formally in [5–10].

Research reports [11–13] present the strategy of the paper in full.

3 CONTEXT AND POTENTIAL CONTEXTS IN A PROGRAM

There is no consensus to treat the term *context* in computer science. There is an important distinction between texts written in natural and programming languages: the first are read and interpreted simultaneously, the second are 'read' (compiled) and 'interpreted' (executed) usually in two steps. So values of many variables and expressions are defined in the program dynamically; different paths can be passed in the program when the same variables have different values; and in general it is impossible to find the paths statically, by the text of the program.

Analysis given in [11] allows defining the notion *context* for a programming language that is similar to the notion for a natural language. The main meaning of the notion is to reflect mono-semantic understanding of some fragment of the continuous information flow in a program.

We use the term *context for a unit of a programming language in a program* (or just *context in a program*) for a part of an execution trace of the program that has the following properties:

- The trace consists of a sequence of fragments going through semantic units of the program text;
- Meaning of all the semantic units is strictly defined in this part of the trace;
- All variables of the part get defined values when the control goes through the part.

So a context in a program (i.e., a specific trace, a path passed) is formed as a result of an execution of the program. Before the execution, there are potential program execution traces only that all together form the program control flow.

Then a ***potential context in a program*** is a part of a potential program execution trace, and this part consists of a sequence of fragments of a potential trace going through semantic units of the program. This sequence is getting a context if the meaning of all its units is strictly defined, and all its variables get defined values when the control goes through the part. If some variables do not get defined before being used; then corresponding paths (traces) through the program unit are *non-contextual*.

A *context in a program unit* is a path in the program unit that has been gone between the first statement of the program unit (the entry to the unit) and its last statement (the exit) during a program unit

This conference paper appears in the book, Innovations Through Information Technology, edited by Mehdi Khosrow-Pour. Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 1. A program unit in Turbo Pascal 7.0 *and its potential contexts.*

```
1  procedure EXAMPLE (J  : real;
2                     N : integer);
3  var  I      : integer;
4       NEW : real;
   begin
5     for I := 1 to N do
      begin
6        NEW := 0;
7        if (I < J)
         then
8           NEW := NEW + Sin(J)
         else
9           NEW := NEW + Cos(J);
      end;
10    writeln(NEW);

11 end;
```

The set of potential contexts

(1, 2, 3, 4, 5, 6, 7, 8, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 9, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 9, 5, 6, 7, 9, 5, 10, 11)

The 'non-contextual' path

(1, 2, 3, 4, 5, 10, 11)

In this path, the variable NEW is used in the simple statement 10 whose value is non-defined to the statement.

execution, if all semantic units of the program unit had meaning, and all program unit variables got defined values during the execution.

Then a potential context in a program unit is a path between the first and the last statements of the program unit that can be passed with permissible values of program unit variables.

Then one context in a semantic language unit (e.g., in a program unit) is one path through the unit (the program unit) only. It does not necessarily cover all the text of the unit (of the program unit). A trace through-passing a control flow construction usually covers a part of the construction. So to cover all the text of the unit, we need to consider several contexts: a *set of potential contexts of a program fragment*, and build not separate models of the potential contexts but a *model of the set of potential contexts of a program fragment* (in short, a Program Fragment Model – *PFM*).

Figure 1 presents a Turbo Pascal 7.0 program unit, potential contexts and a non-contextual path for it. This program unit has no semantic sense; all its statements are given to illustrate execution traces through the unit only. Any potential context (a path) is represented as a potential *Sequence of executed s-statements* (see explanations below).

4 A POTENTIAL CONTEXT MODEL

4.1 Principles to Build the Potential Context Model

Let us formulate principles to build a model of potential contexts.

1. The 'higher-level' executable semantic unit of a program is a program unit.
   Some part of a program execution trace treated as an execution trace of a particular program unit can be reduced to a model of this program unit including complete information about attributes of variables whose values this program unit returns to a caller. So the 'higher-level' model of potential contexts shall be a *model of the set of potential contexts of a program unit* (a Program Unit Model, *PUM*).
2. Minimal executable semantic unit of a programming language with complete meaning is a *simple statement* (see the ***Note*** in the end of the paper; below we use the shortening *s-statement*).
   In general sentences are built from these units. So the 'lower-level' model of potential contexts shall be a *model of the set of potential contexts of a simple statement* (a Simple Statement Model – *SSM*).
3. A *model of the set of potential contexts of a program fragment* (*PFM*) has to have a description of the set of variables visible in the fragment and a description of the set of states of used variables.
   As it follows from the definition of a *context in a program*, any variable of any program fragment (starting from an s-statement) has to get a value before any usage of the variable. A possible state of a variable in the beginning and in the end of a fragment we call the *value definitiveness*. The analyser that parses the program and builds the model assigns values of the state: defined, non-defined, formal parameter, defined in a subprogram etc. (10 possible values), to the corresponding attributes of the model.
4. The model has to use a common practical approach to test loops.
   It is usual for testing strategies not to take into account all paths through a loop because the number of these paths can be huge. Usually they consider till three paths through a loop [3]. However, if the loop body has more complex structure than just a linear sequence of s-statements, it is not enough to test the loop 'well'. So we use the same restrictions that are used in the visit–each–loop coverage strategy:
   • for a loop with a pre-condition or a parameter: one path around the loop body, all possible paths between the first and the last s-statements of the loop body (including one return to the first s-statement), and one path with a double pass through the loop body (i.e., with two returns to the first s-statement) for one of all possible paths through the loop body;
   • for a loop with a post-condition: all possible paths between the first and the last s-statements of the loop body with a single pass through the loop body (without return to the first s-statement) and one any path with a double pass of the loop body (i.e., with one return to the first s-statement) for one of all possible paths through the loop body.
5. The process of building the model for a particular program has to be based on a 'conventional' process of program syntactic and semantic analysis.
   Building a model of the set of potential contexts of two sequential program unit fragments is realised by means of some 'union' of models of these fragments. So building a PUM, we always start uniting a model of the program unit heading s-statement and a model of the next executable s-statement or a model of a block. The result of the union is a PFM that is a *model of the set of potential contexts of an s-statement sequence* (an s-Statement seQuence Model – *SQM*). This SQM is united with the model of the next s-statement or the block, etc. Building a *model of the set of potential contexts of a block* (a Block Model – *BM*) is always started by uniting the corresponding model for the first s-statement of the block and the model of the next executable s-statement of the block or the model of the second level block, etc.

4.2 A Potential Context Model

Let us define a general *model of the set of potential contexts of a program fragment – PFM*, and then consider peculiarities of those models for different types of semantic language units (they are always program fragments).

PFM = (*Fragment identification*,
Set of variables visible in the fragment,
Set of variable states before execution of the fragment,
Set of potential paths through the fragment,
Set of variables used,
Set of variable states after execution of the fragment)

Fragment identification is an exhaustive description of the position of a program fragment and some its attributes.

Set of variables visible in the fragment is an exhaustive description (for a particular set of tasks solved) of variables accessible by statements of the program fragment.

Set of variable states before execution of the fragment is the set of sets of values of the attribute *value definitiveness* before a fragment execution for variables accessible by statements of the program fragment.

Set of potential paths through the fragment is an exhaustive description (for a particular set of tasks solved) of all potential control flow paths through the program fragment.

Set of variables used is the set of sets of descriptions of variables used in this program fragment. For each couple (*a potential path through a program fragment, a variable state before execution of the fragment*), there is its own set of descriptions of variables used.

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Set of variable states after execution of the fragment is the set of sets of values of the attribute *value definitiveness* after a fragment execution for variables accessible by statements of the fragment. For each couple (*potential path through a program fragment, variable state before execution of the fragment*), there is its own set of variable states after potential passing the path.

Formal definitions of all these notions can be found in [11–13].

According to the principles of Sub-section 4.1, there have to be defined four building processes for four models: for a s-statement, for a sequence of s-statements, for a block, and for a program unit. Building the models is the topic for a separate paper [12]. Here we note some features of the processes only. For example, the model of the set of potential contexts of an s-statement sequence is defined as follows.

First, we introduce the operation of *union* (denoted by the symbol **È**) to get the model of the set of potential contexts of a s-statement sequence (SQM) from: (1) two corresponding models for s-statements (SSM), (2) SQM and the corresponding model of the next s-statement (SSM), (3) SQM and the corresponding model for the next block (BM), and SSM and BM.

These unions are denoted as follows.

If there are a s-statement with the number *i* and the next s-statement with the number *i+1* in a program unit, then the model *SQM*{*i, i+1*} of the set of potential contexts of the s-statement sequence (*i, i+1*) is formed as a union of two models: *SSM*{*i*} and *SSM*{*i+1*}:

SQM {*i, i+1*} = *SSM* {*i*} **È** *SSM* {*i+1*}

Similarly we may write for three other unions:

SQM {*i, i+j+1*} = *SQM* {*i, i+j*} **È** *SSM* {*i+j+1*};

SQM {*i, i+j+l*} = *SQM* {*i, i+j*} **È** *BM* {*j+j+1, i+j+l*};

SQM {*i, i+l*} = *SSM* {*i*} **È** *BM* {*i+1, i+l*}.

Each of these four models is a model for a program fragment, and hence, they have the structure of *PFM*. The operation **È** relates to all components of the models. When a program unit has been analysed till the end, the corresponding *PUM* has been built for it. Then the argument *Set of potential paths through the fragment* of the model keeps the set of all possible paths through the program unit; this set is the set of potential contexts of this program unit. Full description of all the models and operations on the models are presented in [12].

5 THE POTENTIAL CONTEXT COVERAGE STRATEGY

First we need to define two notions: the *set of program unit components that have to be covered* during the tests (the minimum number of test cases required for the strategy [3]) and the *set of components* (really) *covered* during the tests.

Formal definitions of these notions in terms of the set theory and the measuring language model [5-10] are given in [12].

The test effectiveness ratio (TER) for this strategy is defined as in [4]:

$$TER = \frac{\left| Covered\ \ potential\ \ contexts \right|}{\left| Potential\ \ contexts \right|}$$

This metric shows the extent to which a particular set of the test cases satisfies the testing strategy under consideration.

Thus to get the value of this metric for the potential context coverage strategy applied to a particular program unit for a particular set of test cases, there have to be built:

- The set of potential contexts of the program units;
- The set of execution traces for this program unit and for this set of test cases;
- The set of potential contexts covered by the tests.

Figure 2 presents calculation of the metric for the program unit shown in Figure 1.

Figure 2. The potential context coverage metric for the program unit of Figure 1.

The set of potential contexts

(1, 2, 3, 4, 5, 6, 7, 8, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 9, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 9, 5, 6, 7, 9, 5, 10, 11)

The set of execution traces

(1, 2, 3, 4, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 8, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 9, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 8, 5, 6, 7, 9, 5, 10, 11)

The set of covered contexts

(1, 2, 3, 4, 5, 6, 7, 8, 5, 10, 11)
(1, 2, 3, 4, 5, 6, 7, 9, 5, 10, 11)

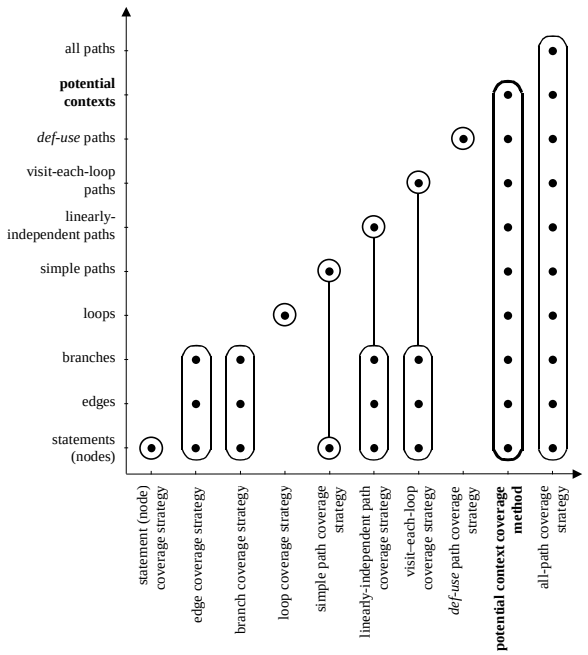
$$TER = \frac{\left| \text{Set of covered contexts} \right|}{\left| \text{Set of potential contexts} \right|} = 2:3 = 0{,}66$$

6 COMPARISON OF THE NEW STRATEGY WITH WELL-KNOWN COVERAGE STRATEGIES

Virtues and shortcomings of this new strategy are analysed in [13]. Here we just note the following.

1. When the set of potential contexts of a program unit is built, the analyser finds 'non-context' paths (and statements) that include usages of variables before their values have been defined. It allows finding semantic defects 'non-initialised variables' before actual program unit testing. As a matter of fact, no test cases are needed to find these defects. Since it is one of the most common and difficult-to-locate type of defects [1-3], automatic finding of those defects can allow decreasing the total time of program implementation and testing.
2. If the potential coverage strategy is applied alone, it allows testing of a program unit equally or better than application of all other traditional structure coverage strategies together with the exception of all-path coverage (see Figure 3: mainly different strategies supplement each other. None can have the same effect as the potential coverage can except the all-path coverage; two last strategies cover all possible components of a program unit or

Figure 3. Test coverage strategies for program units and program unit (or its control-flow model) components covered when the strategies are applied.



Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

its structural model; each other conventional strategy covers one or few program or model elements/ components only).

3. Efforts to develop static and dynamic components of tools for implementing and measuring the potential context coverage are similar to efforts to create similar tools for each conventional strategy. Since the potential coverage strategy is more powerful, it is reasonable to develop and use those tools for this strategy only.
4. In general, efforts to develop the complete set of test cases to cover all potential contexts in a program unit according to the new strategy are higher than for any conventional test coverage strategy (except all-paths coverage) because the number of paths tested is bigger. However the program unit is tested better because the number of paths tested is bigger.

CONCLUSIONS

The paper presents a new structure coverage strategy for program units of imperative programming languages, the potential context coverage.

This strategy allows the same or better testing of a program unit than all conventional coverage strategies applied together except all-paths coverage. The strategy does not have apparent disadvantages.

All notions used to define this strategy are formally defined. It allows rather simple and unambiguous implementation of the strategy. Efforts to develop the corresponding tools are comparable with ones to create similar tools for conventional strategies.

Note: LOA considers programs as being constructed from *simple statements* instead of statements. The simple statement is [7–10]: (1) a procedure/function declaration (heading), (2) a dynamic variable declaration, (3) a 'functional' statement of an algorithmic programming language (in particular, an assignment and data input/output statement), (4) the predicate part of a 'complex' control statement **(***if*-part of *if*-statement, *for*/*while*/*until*-part of a loop statement, *case*-part of a *case*-statement), (5) *goto*-statement, (6) *call*-statement and some other statements. The measuring model of the construction *Simple statement* is a component of the *measuring language model* of an algorithmic language.

REFERENCES

1. Pressman R.S. Software Engineering: Practitioner's Approach. Fifth edition. McGraw-Hill Inc., 2000.

2. Sommerville I. Software Engineering. Sixth ed. Addison Wesley Longman Ltd. 2001.

3. Fenton N.E., Pfleeger S.L. Software metrics: A rigorous & practical approach. Second edition. Revised printing. International Thompson Computer Press, 1997.

4. Collins electronic dictionaries.

5. Cogan B.I., Hunter R.B. Language-oriented approach to software measurement. In SMS-96 (eds J.C. Munson and W.F. Tichy). *Proc. Software Metrics Symposium*. IEEE Computer Society Press, 1996, pp 3–8.

6. Cogan B.I., Matveeva T.O. A relational approach to software measurement and quality assessment. In ESCM'96 (ed A.J. Couderoy). *Proc. 7th European Software Control and Metrics Conference*. UK, 1996, Wilmslow, pp 280–291.

7. Cogan B.I., Hunter R.B. Definition and collection of metrics for comprehensive software measurement. *Software Quality Journal*, 1996, 5:211-220.

8. Cogan B.I., Matveeva T.O. Automatic software evaluation based on domain knowledge and formal evaluation specification. In *Quality Improvement Issues*, SQM VI (eds. C.Hawkins, M.Ross, and G.Staples). *Proc. 6th Int. Conf. on Software Quality Management*, Springer-Verlag Berlin Heidelberg New York, 1998, pp 173-184.

9. Cogan B.I., Shalfeeva E.A. A generalised structural model of structured programs for software metrics definition. *Software Quality Journal*, 2002, 10: 147-165.

10. Cogan B.I, Matveeva T.O., Nikiforova N.Yu. Formal definitions of language-based coverage metrics in the context of the language-oriented approach. Proc. *The 10th International Conference on Software, Telecommunications and Computer Networks SoftCOM 2002*, University of Split, Croatia, 2002. P. 28-34.

11. Cogan B.I., Nikiforova N.Yu. A definition of the notion 'context in a program': Research Report 4–2001. Vladivostok, IACP FEB RAS, 2001 (in Russian).

12. Cogan B.I., Nikiforova N.Yu. A model of a 'context in a program': Research Report 5–2001. Vladivostok, IACP FEB RAS, 2001 (in Russian).

13. Cogan B.I., Nikiforova N.Yu. The potential context coverage strategy for program units: Research Report 5–2003. Vladivostok, IACP FEB RAS, 2003 (in Russian).

Copyright © 2004, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

0 more pages are available in the full version of this document, which may be purchased using the "Add to Cart" button on the publisher's webpage:
www.igi-global.com/proceeding-paper/potential-context-coverage/32408

Related Content

Missing Part of Halal Supply Chain Management
Ratih Hendayaniand Yudi Fernando (2018). *Encyclopedia of Information Science and Technology, Fourth Edition (pp. 5456-5464).*
www.irma-international.org/chapter/missing-part-of-halal-supply-chain-management/184248

Analyzing the IS 2010 Model Curriculum for Evidence of the Systems Approach
George Schelland Richard Mathieu (2016). *International Journal of Information Technologies and Systems Approach (pp. 54-66).*
www.irma-international.org/article/analyzing-the-is-2010-model-curriculum-for-evidence-of-the-systems-approach/144307

Application of Automatic Completion Algorithm of Power Professional Knowledge Graphs in View of Convolutional Neural Network
Guangqian Lu, Hui Liand Mei Zhang (2023). *International Journal of Information Technologies and Systems Approach (pp. 1-14).*
www.irma-international.org/article/application-of-automatic-completion-algorithm-of-power-professional-knowledge-graphs-in-view-of-convolutional-neural-network/323648

Methodology for ISO/IEC 29110 Profile Implementation in EPF Composer
Alena Buchalcevova (2017). *International Journal of Information Technologies and Systems Approach (pp. 61-74).*
www.irma-international.org/article/methodology-for-isoiec-29110-profile-implementation-in-epf-composer/169768

A Utility Theory of Privacy and Information Sharing
Julia Puaschunder (2021). *Encyclopedia of Information Science and Technology, Fifth Edition (pp. 428-448).*
www.irma-international.org/chapter/a-utility-theory-of-privacy-and-information-sharing/260204