Transactions on Information and Communications Technologies vol 4, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Software testing using an automatic generator of test data

C.J. Burgess
Department of Computer Science, University of Bristol, Bristol BS8 1TR, UK

ABSTRACT

The automatic generation of test data has been used as a tool for the black box testing of both compilers and electronic hardware, but only rarely has it been applied to the testing of other types of software. This paper reviews the the main techniques that have been employed in these generators, giving the advantages and disadvantages of this approach, and how it might be applied to a far wider range of software as a powerful software tool to complement other methods of testing.

INTRODUCTION

Testing is still a major part of software development and is fairly time-consuming. Despite the improvements in formal methods for specifying software, which can lead to a substantial reduction in the number of remaining errors, it is still not possible to develop error-free software for any reasonably sized programs. The situation is not improved by the fact that many software systems are getting progressively larger as new and often more complex applications are being tackled. Thus testing is likely to remain a very important part of software development and validation for many years to come. Adrion[1] has published a review of software validation, including the relative merits of both black box, i.e. functional testing, and white box, i.e. structural testing.

There have been a number of papers published on the automatic generation of test data for software. They can be split into two main types. The first type have concentrated on structural testing. These aim to generate the test data automatically, so as to achieve the maximum coverage of

the program code, as measured by a variety of metrics, such as the percentage of program paths executed at least once. The second type is based on functional testing and uses some functional specification of the program to guide the automatic generation of the test data. Here the aim is to produce data which exercises every function of the program at least once, and in many cases, exercises a variety of combinations of functions, without any reference to the actual code of the implementation. A general review of the automatic generation of test data for software was published by Ince[11].

Most of the papers based on structural testing have addressed software in general, and whilst structural testing has a certain value, it is often very expensive in terms of computer resources and also lacks any mechanism for the detection of the absence of any required code. In addition, although the coverage achieved can be measured in terms of certain metrics, it is not always clear how these metrics can then be related to the reliability of the actual software itself.

The majority of the papers which use functional testing as their basis, are concerned with the testing of compiler software. Other types of software are sometimes included as part of a paper addressing compilers as the main topic and a few papers have been published relating to other types of software, including the testing of software simulations of VLSI circuits. This paper concentrates on material published on the automatic generation of test data based on functional testing, and aims to show how the techniques used mainly for the testing of compilers and VLSI circuit simulations might be applied to a far wider range of software.

TERMINOLOGY FOR GRAMMARS

This section is only intended for those who are unfamiliar with the representation of grammars and their semantics.

1. Context-free Grammars

   The syntax of a language can be described by a context-free grammar. There are several ways of representing the grammar, but one of the most common ways is by using production rules. These production rules formally specify the order in which the basic symbols of the language can be combined to form syntactically valid sentences in the language. In the particular case of compiler testing, these sentences would normally be complete test programs.

   As an example, a typical part of the grammar for representing integer numbers of any length is:-

   Integer → Integer Digit | Digit

Digit → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

In these grammars, the symbols appearing on the left-hand sides of the production rules, e.g. Integer and Digit, are called non-terminal symbols, and the other symbols, which can only appear on the right-hand side of production rules, are called terminal symbols. The → symbol indicates a production rule and separates the subject, which is the non-terminal symbol on its' left-hand side, from its definition on the right-hand side. The | symbol represents an alternative definition of the subject and is a very convenient abbreviation to avoid repeating the subject when there are a number of alternative definitions. The grammar for a conventional programming language might have 100 to 300 production rules to cover the whole of the syntax of the language.

2. Representation of Semantic Information

There are a variety of ways of adding some semantic information to a context-free grammar. One of the earlier and better methods that is still used today, is by using an attribute grammar, which was first described by Knuth[12]. He associated any number of named attributes with any non-terminal in the grammar. These attributes were one of two types called inherited and synthesised attributes. The type determines the direction in which the semantic information is passed in the syntax tree when the grammar is used to parse sentences in the language. Synthesised attributes correspond to information that is passed up the syntax tree and inherited attributes correspond to information that is passed down the syntax tree. Every production rule has an associated semantic rule, which describes the semantic actions that are associated with that rule using these attributes. Parametric grammars, which are also referred to in some of the papers referenced, provide similar facilities using a different notation and are described by Camuffo[7]. Other authors use different extensions to the production rules to cope with the semantics, which often have less expressive power.

More details on grammars and programming language definition can be found in many textbooks on programming language design or compiler construction, e.g. Aho et al.[2].

GENERATION OF TEST DATA

One of the main uses for context-free grammars, besides providing a formal description of the syntax of a language, is for the syntax analysis of a string, to determine whether or not that string is in the language. However, the same grammar can also be used to generate strings from a given language. This is done by expanding any non-terminal in a string by the right-hand

side of one of the production rules, where it is defined, starting with the single non-terminal that represents the whole language. This process is repeated until the string produced contains only terminal symbols. This is the basis for the algorithm used in many of the generators that are described in the literature. The choice between the different alternative definitions to use to expand a given non-terminal can be done either systematically, so as to ensure that they are all used at least once, or at random, using a built-in pseudo-random generator. Some of the generators also allow the addition of weights to the production rules to control how frequently one alternative definition is chosen with respect to the others.

TEST DATA FOR COMPILERS

Many of the schemes described for the automatic generation of test data for the testing of compilers have used the syntax of the programming language as the main basis for their functional specification. One of the earliest was by Purdom[15], who used the syntax specification of the language as the input, and generated a series of tests to verify the correct construction of an LR parser, which is a commonly used algorithm for the syntax analysis stage of a compiler. He systematically selected the different definitions available for a given non-terminal, when expanding a given subject, so that all the right-hand sides of the productions rules were used at least once, and also so that only a minimal number and length of test programs were produced. He reported considerable success with this approach.

Several later papers have described schemes which produce test programs designed to compile successfully, but not necessarily to have a meaningful execution. This means that the semantics of the test programs has been controlled sufficiently to ensure successful compilation. For most languages, the major part of this semantics is controlling the type and use of the identifiers. One of the earlier papers in this group was by Celentano et al.[8], who used production rules, augmented by actions, to provide control over the semantics associated with identifiers. The actual productions were selected using a modification of Purdom's algorithm. Bazzichi and Spadafora[3] used a different extended form of the grammar notation and generated two different sets of test programs. In one set, all the programs should successfully compile, but in the other set, each program included a syntax or a semantic error preventing successful compilation. This second set was designed to check for the correct detection and identification of compile-time errors.

Murali and Shyamasunder[14] used a different representation of the grammar, called a syntax graph, which can be easily constructed from a set of productions, as the main basis for the coding of the actual generator. The

choice between the different definitions was made at random, with weights associated with each definition to control how frequently a given definition should be selected. They also generated both compilable test programs and those containing compile-time errors.

In all the previous papers, most of the test programs generated would not execute meaningfully, and there is no attempt to check for the correct execution. Thus the code generator section of the compiler is not properly tested. There were two early attempts at generating test programs that were meaningful to execute but both had severe limitations [9, 10]. The first comprehensive scheme was described by Bird and Munoz[4], where the generated test programs also included self-checking code to check automatically for their correct execution. Their method, although applicable to most programming languages, involved writing a specific generator for each different language. They tested a PL/I compiler and the basic structure of their generator was a loop which selected, at random, the next type of source statement to be generated, and then called the appropriate subroutine to generate an example of that type of source statement. The generated source statement was followed by self-checking code to verify the correct new values of any variables changed by any assignment statements and also to verify that the correct flow path had been executed. The syntax and semantic description of the language was used purely as a specification for the subroutine coding and not as a direct input. They report considerable success with their method, which they also applied to two other types of software which will be described in the next section. They did not attempt to generate programs with known compile-time or run-time errors and their emphasis was on being able to generate and execute a large number of tests with the minimum of human involvement.

A more recent paper by the author[5], describes a scheme for generating very similar test programs to Bird, using a grammar closely related to an attribute grammar for the representation of the syntax and some of the semantic information of the programming language. This enabled a fairly language independent generator to be built. The actual compiler tested was a Pascal compiler. Each of the production rules had a weight attached and some production rules also had guards, which prevented that rule from being used if the semantic context was incorrect. The productions to be used at each stage were chosen at random from the weighted productions. The resulting test programs compiled and executed, with self-checking code to verify the correct execution. These ideas have been further extended by the author and Saidi[6], in a generator of test programs for both normal and optimising Fortran Compilers. The test programs generated not only execute with self-checking code, but also include a number of features which optimising compilers are known frequently to exploit when trying to opti-

mise the object code produced, thus testing both the code generation and optimisation stages of the compiler.

TEST DATA FOR OTHER TYPES OF SOFTWARE

There are comparatively few papers describing the automatic generation of data for other types of software based on functional testing. One of the main papers is by Bird and Munoz[4], which has already been referenced. They show how a special purpose test generator can be built, not only for compiler testing, but also for the testing of a Graphical Data Display Manager and a Sort/Merge Program. A key feature of these generators is the prediction of the correct result of the test, followed by the verification of the test either automatically, or with the minimum of user involvement. The human involvement was unavoidable with the graphics software, since it was necessary to check for the correct visual output on the screen. Their work is specifically within the industrial context and they report a number of benefits of using this type of testing when compared with conventional testing methods.

A more recent paper is by Camuffo et al.[7], which describes the automatic generation of test data for software using a context-free parametric grammar. The paper reports work performed in an industrial context for the testing of arithmetic statements in a Cobol compiler; some operating system commands for their versions of DOS and Unix; and the validation of the communication protocols for performing file transfers. When generating the correct tests, they also computed the expected results, but when generating tests containing errors, they did not predict the effect of the error on the system under test.

An automatic generator of test data for testing software simulations of VLSI circuits is described by Maurer[13]. This uses a language called DGL, which is based on the concept of probabilistic context-free grammars[16]. The production rules can either be selected at random, or systematically, or some combination of the two. The generator also contains two extra features, called variables and chains. Variables provide a mechanism which enables the same generated sequence to appear more than once within a given test. Chains ensure that, for a limited set of productions, all possible combinations are generated as part of the test. An interesting feature of their approach is that they start with a test set which is designed to execute every path in the circuit at least one, more akin to the structural testing of software, and then vary the test in a random way. This is to ensure that every built-in micro-instruction is executed at least once. The user of their generator is responsible for computing the expected results of the tests, but there is the provision of a result field to record the expected

result. Although some of the features are designed very much with VLSI testing in mind, the author claims that, with the use of software simulation of VLSI circuits before fabrication, the gap between software testing and VLSI circuit testing is narrowing. The paper reports heavy and successful use of the generator for VLSI testing by their University Microelectronics Research Center, but very little use as yet on other types of software.

There seems to be little other published material on the use of the automatic generation of test data by industry based on functional testing. This could be due to a combination of factors, such as:-.

1. The technique is not used very widely at present.
2. The technique, where it is used, is very specific to the particular type of software under test.
3. It is not in their commercial interests to spend the time required to publish or to advertise the detailed techniques to others.

One industrial organisation, in a private communication, described how they have used similar techniques to some of those described, to test a large software system, which had a rich language as an interface and contained a large number of different commands. They constructed a generator which generated partially random data and also predicted the effect of the data so that this could be verified when the test data was executed. The effect of the software under test was simulated by a state transition table to assist in the prediction of the effect of the data generated. The same organisation has applied similar techniques to other pieces of software and found that it was good at finding problems at the levels of unit testing and functional verification with a single user, compared with other testing methods.

DESIGN CRITERIA FOR AN AUTOMATIC GENERATOR

On the basis of the literature reviewed it is possible to establish a basic set of criteria for assessing an automatic generator of test data which is based on functional testing.

1. Coverage
   There should be some measure of the testing coverage provided by the generator. This should mean, at the minimum, that all the functions in the specification should be used at least once, and in addition, some combinations of functions, where the functions are not completely independent.

2. Cost
   The development of the tool, if a suitable one does not already exist, and its subsequent use, should require the minimum of human involvement and not require an excessive amount of computer resources. This can be measured by comparison with the human and computer resources required to do the equivalent testing using manually constructed test data.

3. Prediction of results
   It should be possible to predict automatically the expected results of the test data, either directly within the generator as the test data is being produced, or by the use of some form of software simulation of the system under test.

4. Checking of results
   The results of the execution of the test data by the software under test should be automatically compared with the expected results. Any differences need to be reported with sufficient information so that both the test and the type of failure is reproducible. This should enable a very large number of tests to be executed with the minimum of human involvement.

5. Number of tests
   There should be some method of determining how many tests should be generated, since most generators are capable of generating many more different tests than needed. This probably has to be a statistical measure of the likelihood of failure of the software being tested, based on how many errors have been detected with a given number of tests, or some other criteria which may be harder to relate to the reliability of the final product.

6. Range of values used
   Within the test data, for any given data item, there is likely to be a wide range of possible values. The test data, whether automatically generated or manually constructed, must include the following three types of values:-
   (a) Extremal values These are values which are known to be at, or near, the ends of the range of valid input values, or input values known to produce output values near to the end of their range.
   (b) Special values These are valid input values which are known to have special effects on the software in use. A very common example is zero for a numerical value or its equivalent for a different type.
   (c) Non-extremal values These are the values that are not in either of the previous two categories and correspond to the majority of the values that are used in automatically generated tests.

7. Testing error detection
   A generator, as a user option, should be able to generate test data with given types of errors and be able to predict the form of error response that should be obtained from the software under test. There are a number of forms these tests could take and many of them will be very dependent upon the type of software being tested. Some fairly general possibilities are:-

   (a) Values of input data just beyond the valid range.

   (b) Incorrect formats within a data field.

   (c) Incorrect number or order of fields.

   (d) Incorrect spellings of keywords.

   (e) Common mistypings.

   (f) Invalid combinations of otherwise valid data.

   It is not clear how many of these can usefully be built into an automatic generator, as it might be quite difficult to simulate the correct error behaviour of the software and hence predict the error response. However a set of these tests, without the prediction of the error response, would also be a useful diagnostic tool.

8. Limited exhaustive testing
   For some software, a useful feature would be to generate exhaustively all the possible combinations of some group of features, within a larger test framework. With a grammar based generator, this probably implies an ability to specify either strictly sequential selection of productions, or random selection, and also to provide a linking mechanism between related productions. The only published paper that seems to provide such a mechanism is the one by Maurer[13].

9. Ease of use
   Like most other software, a successful generator has got to be easy to use if anyone other than its author is going to want to use it.

BUILDING A COMPLETE TEST SYSTEM

The complete test system should have four tasks.

1. The generation of the test data.

2. The prediction of the results of the test data.

3. The actual execution of the tests by the software under test.

4. The comparison of the predicted and actual results of the tests.

The design and implementation of these tasks will depend both on the type of software being tested and the method used to predict the test results, if this is possible. There are at least five basic approaches.

1. Design the generator so that it predicts the expected results of the test data whilst the data is being generated, and generates self-checking data at the same time. This method has been used for compiler testing [4, 5, 6], but it is only likely to be applicable for a limited range of software, where the provision of self-checking data is feasible.

2. Design the generator so that it predicts the results, as for 1 above, but instead of generating self-checking tests, the results of the execution of the test data is compared with the predicted effect, either manually, or preferably by an automatic comparison program.

3. When the required test data is such that its effect cannot be predicted during its generation, then the test data is applied to both the software being tested, and a software simulation of the software, and their results compared, once again, preferably automatically. The software simulation only has to simulate those parts of the software being tested, and unless the software is time-dependent, it need not be particularly efficient. One possibility would be to use an executable formal specification of the software to provide the simulation, but this probably requires more research and development into executable formal specifications.

The next two designs are less desirable, but are still worthwhile.

4. When the test data is being generated, provide a special field so that the user can fill in the expected result. This method has been used in the testing of software simulations of VLSI circuits[13].

5. When there are two different versions of the software, both expected to achieve the same results, then useful results can be obtained without prediction, by comparing the results of the two versions with the same test data. This is really closely related to case 3 above. This has been used for compiler testing, but would also seem to be a viable way of testing new versions of software against previous versions, where only new features or corrected bugs should cause any differences in the results produced by the test data.

DESIGNING THE GENERATOR

The first major decision is which of the following approaches to adopt for the design of the generator.

1. Special purpose
   A generator is designed specifically for the piece of software being tested. All aspects of the syntax and the semantics of the test programs are hand-coded into the generator. The process is repeated for each new piece of software. This method was adopted by Bird[4], and when allowing for the cost of large software, this is not necessarily a bad approach, if it makes the construction of the generator a lot easier.

2. General purpose
   The generator is designed as a more general purpose piece of software and uses a representation of the grammar, or some other formal specification, as one of its main inputs. The generator then either actually produces the test data directly, or generates as output, the code of a special purpose generator similar to the previous case. This is the most common approach adopted by the testers of compilers, as the input required has a fairly complex grammatical structure and the grammar of the appropriate language is already available. This approach has also been used for the testing of simulations of VLSI circuits and some other pieces of software [7, 13].

The next fundamental decision is whether to include the prediction of the results of the test data, and if so, how this is to be implemented. Although obviously desirable, the difficulty of doing this will be very dependent upon the type of software under test. The two main options are:

1. Build the prediction mechanism into the generator itself. If the generator is a special purpose one anyway, then this involves just extending the code. If the generator is more general purpose, then the same philosophy can be extended by the use of a grammar allowing some semantic information to be expressed, such as an attribute or parametric grammar. It is still unlikely that all the information can be easily expressed formally but some mechanism of calling specially coded subroutines would allow for the inclusion of those features that cannot be expressed in any other way.

2. Use the generator to produce only the syntax of the tests, and write a software simulation of the software under test to predict the results, with just sufficient complexity to cope with the area being tested. This would seem a good approach where the cost of writing such a simulator is comparatively small, or where the result of any particular test is not completely independent but depends on the current state of the test software. These states may possibly be simulated using a state transition table in the software simulation or by some other technique.

The third decision concerns the checking process and whether this can be automated. This will obviously reduce the human involvement if there are a very large number of tests, and may be feasible for a variety of software when the test data generated can be executed directly by the software under test.

In addition, there are a number of fairly independent decisions.

1. Should the automation extend to the generation of incorrect data and the prediction of the correct error response from the software under test? This will probably make it harder to always predict the results accurately, and to automate the checking of the results.

2. Should the generated data be restricted to non-extremal values? If not, are the extremal and special values obvious or easily calculated? Is it possible to deduce these from either an informal or formal specification of the software?

3. Are there some parts of the possible test data spectrum that require exhaustive testing and not just random selection?

THE TYPES OF SUITABLE SOFTWARE

The approach described in this paper is likely to be more suitable for some types of software than others. The following are some characteristics, which either individually or together, could be used to identify suitable software.

1. The input for the software is a fairly rich language, with a well-defined syntax.

2. There is a fairly formal way of predicting the expected results of any piece of test data or it is easy to construct a simulation of the software under test.

3. The different parts of a sample data set are not completely independent and a large number of different combinations are possible.

   e.g. a command language where the effect of any particular command is dependent upon the previous commands.

4. By the nature of the application, the software is quite complex and therefore the normal testing process would be quite complex and involve a large number of tests.

5. It is very important that the software should have a high reliability.

There are other characteristics of software that might make it difficult or inappropriate to apply this approach. Some of these might be :-

1. Software that requires a large amount of interaction with the user. This could cause problems but even this can sometimes be overcome as illustrated by the testing of the Graphical Data Display Manager, described by Bird[4].

2. There is a severe limit on the time and resources available for testing.

3. The software is comparatively simple, with typically very small data sets that are completely independent of one another.

ADVANTAGES AND DISADVANTAGES OF THE APPROACH

The main advantages of the approach are:-

1. It is easy, once the test system has been constructed, to generate a very large number of tests. If the prediction and the checking of the results are fully automated, this should enable a large number of tests to be carried out in a comparatively short time compared with other testing techniques.

2. It is very good at producing a large number of different combinations, either within one set of test data or between sets. This should provide a good test where different parts of the test data interact with one another.

3. Since, at the heart of the generator, there is most likely a pseudo-random number generator, then there is a random element within the selection of the test data. There is evidence from the literature that this helps to produce more effective testing, e.g.[11]. In addition, by recording the seeds used to start the random number generator, any set of test data need not be stored, but can be reproduced at a later date, if required.

4. The production and use of the generator can be more enjoyable and rewarding for the person doing the testing , than having to use a large number of manually constructed test cases.

5. There is evidence from the published literature that where this method has been employed, they have been pleased with the results.

The main problems or disadvantages with the approach are:-

1. There is no obvious method of knowing how much data to generate. The only possible help is if a statistical record is built into the system, to record the number of tests, and the number and types of failure detected. This provides the possibility of some measure of reliability as a guide to how much more testing is required. This situation is no worse than that which occurs when using manually constructed test data.

2. For some software, it is important to be able to have detected errors corrected as quickly as possible, as one error may cause a large percentage of the failures and cause other errors to be obscured. For the same reason, until the software is fairly robust, it will probably not be worthwhile generating a very large number of tests.

3. It is not clear how best to generate incorrect data and predict error responses.

4. Very few systems have a mechanism built in to allow for the testing of extremal and special values, leaving the testing of these cases to hand-coded tests.

FUTURE WORK

There are no generators known to the author that meet all the design criteria proposed, even for a limited range of software, e.g. compilers, but most of the criteria are met by at least one system, and most of the systems described meet many of them. Thus there are a number of open research questions still to be resolved.

1. Can one design a suitable language for an automatic generator which is both easy to use and meets many or all the proposed design criteria?

2. Should there be different generators for each particular type of software or can one scheme be designed that is flexible and adaptable enough to be tailored towards a particular piece of software?

3. With the slowly increasing use of formal methods for functional specification, can these be used to help in either the generating of the test data, or, by the simulation of the formal specification, the production of the predicted results for a given set of test data?

4. Is there a useful systematic method of producing incorrect data for testing error responses of software?

5. What is the right balance between hand-coded tests and automatically generated tests for a given piece of software?

SUMMARY

This paper has reviewed the main techniques that have been used in the construction of automatic generators of test data for compilers and some other types of software, based on functional testing. It then gave some of the desired characteristics that such a generator should have and shown possible ways in which a test system could be generated which would be applicable to a wide range of software. It also identified those characteristics that software

might have that was suitable for this type of testing. Finally it summarised the advantages and disadvantages of this approach as a complement to other methods for the testing of software.

REFERENCES

1. Adrion,W.R., Branstad,M.A. and Cherniavsky,J.C. 'Validation, Verification and testing of Computer Software', Computing Surveys, Vol. 14, No. 1, pp. 159-192, 1982.

2. Aho,A.V., Sethi,R. and Ullman,J.D. 'Compilers - Principles, Techniques, and Tools', Addison-Wesley, 1986.

3. Bazzichi,F. and Spadafora,I. 'An automatic generator for compiler testing', IEEE transactions on Software Engineering, Vol. SE-8, pp. 343-353, 1982.

4. Bird,D.L. and Munoz,C.U. 'Automatic generation of random self-checking test cases', IBM Systems Journal, Vol. 22, pp. 229-245, 1983.

5. Burgess,C.J. 'Towards the automatic generation of executable programs to test a Pascal compiler', Software Engineering '86, Ed. D.Barnes and P.Brown, IEE Computing Series Vol. 6, Peter Peregrinus Ltd., pp. 304-316, 1986.

6. Burgess,C.J. and Saidi,M. 'The Automatic Generation of Test Programs for Normal and Optimising Fortran Compilers', Department of Computer Science Technical Report, No. CSTR-92-18, University of Bristol, 1992.

7. Camuffo,M., Maiocchi,M. and Morselli,M. 'Automatic software test generation', Information and Software Technology, Vol. 32, No. 5, pp. 337-346, 1990.

8. Celentano,A., Crespi-Reghizzo,S., Della-Vigna,P. and Ghezzi,C. 'Compiler testing using a sentence generator', Software Practice and Experience, Vol. 10, pp. 897-918, 1980.

9. Hanford,K.V. 'Automatic generation of test cases', IBM Systems Journal, Vol. 9, pp. 242-257, 1970.

10. Houssais,B. 'Verification of an Algol68 implementation', ACM Sigplan Notices, Vol. 12, pp. 117-128, 1977.

11. Ince,D.C. 'The Automatic Generation of Test Data', Computer Journal, Vol. 30, No. 1, pp. 63-69, 1987.

12. Knuth, D.E. 'Semantics of Context-Free Languages', Mathematical Systems Theory, Vol. 2, No. 2, pp. 127-145, 1968.

13. Maurer,P.M. 'The Design and Implementation of a Grammar-based Data Generator', Software Practice and Experience, Vol. 22, No. 3, pp. 223-244, 1992.

14. Murali,V. and Shyamasunder,R.K. 'A sentence generator for a compiler for PT, a Pascal subset', Software Practice and Experience, Vol. 13, pp. 857-869, 1983.

15. Purdom,P 'A sentence generator for testing parsers', BIT, Vol. 12, pp. 366-375, 1972.

16. Wetherell,C.S. 'Probabilistic Languages: A Review and some Open Questions', ACM Computing Surveys, Vol. 12, pp. 361-379, 1980.