Improving Testing of Complex Software Models through Evolutionary Test Generation

Michaela Newell
Department of Computer Science and Creative Technologies
University of the West of England
Bristol BS16 1QY United Kingdom
michaela2.newell@live.uwe.ac.uk

ABSTRACT
Considerable cognitive effort is required to write test cases for complex software and fix any defects found. As the generation of test cases using evolutionary computation has a long established track record, this paper explores whether this pedigree can be exploited to improve efficiencies in larger testing suites that typically address complex software models. A genetic algorithm has been designed and implemented with complex software models in mind, and then trialled against five real world programs that vary in scale and complexity. Results show that test case generation using an evolutionary algorithm on average can improve the number of coverage goals met by 75.83%. Therefore we conclude that even with complex models that have thousands of objects improvements can be made.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools (e.g., data generators, coverage testing).*

I.2.8 [**Computing Methodologies**]: Problem Solving, Control Methods, and Search – *Heuristic methods.*

General Terms
Algorithms, Design, Human Factors

Keywords
Automatic Test Generation, Evolutionary Algorithm, Metaheuristic Search, Unified Modelling Language, Complex Data Models

1. INTRODUCTION
Manual software testing of complex software is a cognitively demanding task and so can be expensive, time consuming and occasionally unreliable [1]. Figures released for the USA suggest that approximately $20 billion each year could be saved if more efficient and effective software testing was performed prior to deployment and release. [2]. The need to improve on this situation is significant as in addition to the loss of considerable amount of money, failures of complex safety-critical software systems potentially put human life in jeopardy. As a spectacular example of a complex software systems failure, the European Space Agency estimates losses of $500,000,000 caused by the launch failure of the Arianne 5 rocket in 1996; the root cause of the failure is thought to be inadequate testing coverage [3].

Evolutionary Test Generation (ETG) has attracted significant research interest and shows great promise in reducing the development costs and improving the quality (and hence the level on confidence) in the software under test [4]. Within the field of Search Based Software Engineering (SBSE) [5], many meta-heuristic search techniques have been applied which treat the generation of test cases as a search problem. Meta-heuristic search approaches encode candidate solutions using a problem specific representation, and fitness functions and operations to preserve solution diversity. The technique is typically measured on how expensive, effective and scalable the algorithm is at reaching the test objectives [6]. A widely applied objective fitness function used in meta-heuristic search for test cases is branch coverage [7], where the goal is to arrive at a restricted number of test cases that achieve the maximum degree of branch coverage. However, generating a set of test cases for software systems of realistic complexity presents a challenge not only due to the size of the search space expanding rapidly, but furthermore, many researchers acknowledge solving a complex search problem means there is no optimal or exact solution [8]. This paper therefore sets out to design and implement a genetic algorithm that exploits models of complex software, specifically object-oriented class models, as a basis for generating test cases.

2. BACKGROUND
Evolutionary Algorithms (EA) typically use a population of individuals, rather than one individual candidate. The algorithm then uses optimisation techniques inspired by the biological evolution processes; reproduction, mutation and selection. EAs consist of a number of varying techniques including: genetic algorithms, genetic programming and evolutionary programming. Genetic Algorithms (GA) are arguably the most well known form of EA [9]. Genetic algorithms require three components in order to achieve effective search: a solution representation, a measure of solution fitness and a mechanism for diversity preservation. A representation can typically take the form of real numbers, binary digits or floating point numbers. Examples of GAs using more complex data structures have attempted to address some challenging problems such as scalability, predictability and robustness [10].

There are many methods of generating test cases, including search based [8][10], model based [11][12][13][14] and specification based [15]. Model based test generation is very different to the search techniques discussed. The tests are generated from modelling languages including the most widely used [11], Unified Modelling Language (UML) [16]. The benefit of this method is that in many cases, designs in the form of UML have already been completed; consequently less additional effort is required. Data can be gathered from various UML diagrams including: use cases [14], interaction diagrams [15] or a combination of diagrams [16].

Figure 1 summarises the different techniques that can be used and the level of testing that they are suitable for:

	Unit Testing	Integration Testing	Functional Testing
Search Based Technique		McMinn [8] Harman [10]	McMinn [8] Harman [10]
Model Based Technique	Prasanna *et al.* [11] Nebut *et al.* [12] Swain *et al.* [14]	Prasanna *et al.* [11] Tonella and Potrich. [13] Swain *et al.* [14]	Swain *et al.* [14]
Specification Based Technique			Liu and Nakajima [15]

Figure 1. Techniques suitability to different stages of testing.

Figure 1 summarises all of the previously mentioned frameworks by their technique and how the authors assess their suitability in the various stages of software testing. The figure shows that the only framework that is suitable for all levels is a model based technique proposed by Swain *et al.* [16]. Additionally Figure 1 highlights that various techniques and frameworks can be adapted, one most appropriate to the problem.

3. PROPOSED APPROACH

The proposed approach uses a combination of a model and search based technique to generate test cases and to address the challenges of a complex data model. Other authors have also proposed combining these two techniques and their methods show promising results [17]. A description of how this approach is implemented is included in the following sections.

3.1 Problem Encoding

A prerequisite of the system is that the user must input a UML Class Diagram. As the tool is an initial prototype only one structure of UML is currently supported. The structure supported is the automatically generated structure of StarUML [18]. An example of the structure can be seen from Figure 2.

```
<XPD:OBJ name="OwnedElements[0]" type="UMLClass"
guid="95FFK+ln1kOVRD4GJYRJhQAA">
<XPD:ATTR name="Name" type="string">Class1</XPD:ATTR>
<XPD:REF name="Namespace">YMt6yt/Y90qiBdvoCPhIcwAA</XPD:REF>
<XPD:ATTR name="#Views" type="integer">4</XPD:ATTR>
<XPD:REF name="Views[0]">0tQXfFFEhkaabBimWRCnpAAA</XPD:REF>
<XPD:REF name="Views[1]">Phczm+38hEqdvBhZpBDf3AAA</XPD:REF>
<XPD:REF name="Views[2]">s/+vlwh5EEC9dhihI0tb5gAA</XPD:REF>
<XPD:REF name="Views[3]">zYgdJGmGL0uBTrgKDz2F3wAA</XPD:REF>
<XPD:ATTR name="#Operations" type="integer">2</XPD:ATTR>
<XPD:OBJ name="Operations[0]" type="UMLOperation"
guid="zOXxGE8kH0CZy0upBp5RnAAA">
<XPD:ATTR name="Name" type="string">method1</XPD:ATTR>
<XPD:REF name="Owner">95FFK+ln1kOVRD4GJYRJhQAA</XPD:REF>
</XPD:OBJ>
<XPD:OBJ name="Attributes[0]" type="UMLAttribute"
guid="hPKc2Pm4k0arRtSG5BMetgAA">
<XPD:ATTR name="Name" type="string">var1</XPD:ATTR>
<XPD:ATTR name="Visibility"
type="UMLVisibilityKind">vkPrivate</XPD:ATTR>
<XPD:REF name="Owner">95FFK+ln1kOVRD4GJYRJhQAA</XPD:REF>
</XPD:OBJ>
</XPD:OBJ>
```

Figure 2. StarUML structure: class, attribute and operation.

However, not all models in the case study are as simple as the one that can be seen in Figure 2. One of the models used to validate how the program manages complexity can be seen in Figure 3.

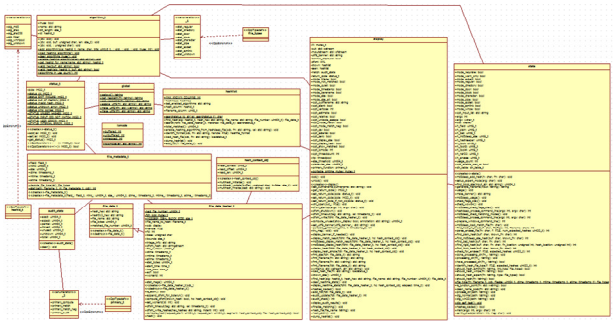

Figure 3. Class diagram for md5deep and hashdeep.

Figure 3 illustrates a diagram used to check the frameworks ability to handle model complexity, with 18 classes and hundreds of attributes and operations. The framework begins by parsing the UML to gain information. In order to identify relevant information within the UML, the required information must first be identified by its XML tag. The tag '<XPD:OBJ>' will differentiate between the type of object (Class, Variable or Method). Inside the objects the parsing looks for the tag '<XPD:ATTR>', which will identify information relating to the object such as the object name.

3.2 Solution Representation

The tool splits by each object narrowing the search space. There is then a separate method that gets specific information from the object. For example, the method can be called to return all method names. Both of these methods are generic to allow for future expansion to obtain more information from the UML.

There are four methods that request information from the generic methods. These four methods are split into types of objects. The logic of these methods is: for each class diagram check for all classes and for each class check for all variables and methods. Currently these four classes return the object names and in the structure specified, add them to a vector. This initial structure includes all the objects and is used later in the fitness function. The benefit of obtaining this information from a design diagram as opposed to the program's code is that we are assuming the design diagrams are correct and we cannot assume this for the program. An example of an initial vector can be seen in Figure 4.

[ClassDiagram1, Class1, var1, var2, method1, method2, Class2, var3, var4, method3, method4]

Figure 4. Example of an initial vector.

Figure 4 shows a genotype representation, Figure 5 the phenotype.

[ClassDiagram1, Class1, method1, Class2, var3, var4]

Figure 5. Example of an individual.

3.3 Diversity Preservation

The standard selection schemes include: tournament, ranking and proportional truncation selection [19]. This framework uses three techniques: tournament selection, one point crossover and single point mutation. The tournaments run by pairing each individual in the population together at random. The winner of the tournament is determined using a fitness proportional selection, as opposed to the absolute fitness value that is determined at the end of each generation. The winner of each tournament is selected for crossover. One point crossover is where one point is selected at

random in the individual and all the data beyond that point is swapped between the two parents, resulting in offspring.

3.4 Fitness Operation

The fitness function assesses the population for coverage goals. Each individual in the population is assessed for its' coverage. A higher coverage can be achieved by including the testing of each object. The individual's assessments are then used for diversity preservation. Once complete, the population's fitness is assessed. The population can increase its fitness score by increasing the number of objects that will be tested. The score is awarded by giving one point if an individual contains an element that exists in the initial vector.

3.5 Genetic Algorithm

The general scheme in psuedocode [20] can be seen in Figure 6.

BEGIN

INITALISE population with random candidate solutions;

EVALUATE population;

REPEAT UNTIL (TERMINATION CONDITION is satisfied) DO

1. *SELECT* two candidates at random;
2. *EVALUATE* each candidate;
3. *SELECT* two candidates at random;
4. *SELECT* crossover point for each candidate;
5. *SELECT* parent for each candidate;
6. *SELECT* child for each candidate;
7. *RECOMBINE* parent of candidate with child of other candidate;
8. *SELECT* two candidates at random;
9. *EVALUATE* each candidate;
10. *SELECT* highest candidate for the next generation;
11. *MUTATE* the resulting offspring;

OD

END

Figure 6. Scheme of an evolutionary algorithm in psuedocode.

4. EXPERIMENTAL METHODOLOGY

Each test will be structured in the same way in order to guarantee the fairness of a comparison. Each test will have a main variable, the program. The test vehicles are five real world programs. Each program varies in the number of objects and the complexity of the model.

The algorithm parameters are: 20 runs for each program, similar to the works of Forrest *et al.* [21]. The number of generations will be 100, similar to the works of Arcuri *et al.* [22]. The population size will be 40 similar to sizes in previous literature [21][22]. 100% of individuals are paired together in a tournament selection which is a common practise in GA's [20]. The crossover rate is 75% which is considered to be ideal [23]. The mutation rate is 5% as this typically shows the best performance [23].

5. RESULTS

The fitness curve for Program 2 is shown in Figure 7.

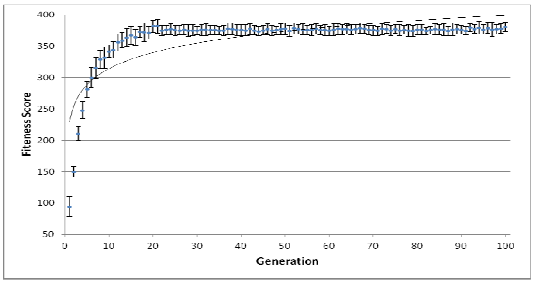

Figure 7. Mean fitness score for 20 runs of Program 2 over 100 generations on a logarithmic scale.

As it can be seen in Figure 7, the fitness score peaks before the 20th generation, which is why every further run used 20 generations, instead of 100. The highest standard deviation point is 17, which is observed at the 7th and 9th generations. As it can be seen, the later generations have a lower standard deviation this is expected as at the beginning the diversity between individuals is larger due to the size of the exploration space is larger. As the candidate solution begins to form the search space decreases, lowering the deviation.

In order to test the results by model complexity and the frameworks ability to scale, Figure 8 shows the total number of objects for each program used in testing. The number of objects directly influences the complexity of the model.

Program Number	Program Name	Length (Total Number of Possible Objects)
1	Bouncy Castles – Open PGP	1478
2	Autopsy – Keyword Search	15
3	md5deep and hashdeep	1872
4	pmd	117
5	TrueCrypt	504

Figure 8. The total number of objects per program.

All of the programs chosen were open source, written in either Java or C++. Source code, test cases and design diagrams were freely available. However, the source code is not used by the framework. The design diagrams are used in the test generation which is then compared to the manual test cases to assess improvement. Figure 9 shows the fitness curve relative to the total number of possible objects listed in Figure 8.

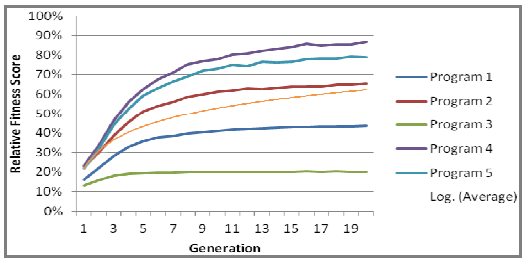

Figure 9. Fitness scores relative to the total number of objects.

Figure 9 shows that every program has a similar fitness curve. The small decreases in the fitness curve are due to the program using a generational model, as opposed to steady state. Figure 9 also shows that the scalability affects the curve. Program 3 has the highest number of objects and the lowest relative fitness score. Program 4 has the second lowest number of objects and the highest fitness score. This indicates that the programs complexity

affects the fitness score. The percentage improvement when comparing automatically generated test cases to manual test development shows that there is a larger room for improvement when the programs complexity is greater. For example, Program 3 achieved a percentage improvement of 95% whilst; Program 1 achieved an improvement of 77%. This could arguably be due to the difficulty of developing tests cases manually for larger programs.

6. CONCLUSION

Results show that significant improvement can be made when using automatic test generation as opposed to manual development. The improvement can be made on software that already exists and is used in the public domain. The complexity affects the final fitness and based on the limited testing, complex programs with more objects, score slightly lower fitness scores. In order to achieve a reasonable relative score (50% or higher) the program size has to be around 1000 objects or lower. However, using the final code base, the average improvement of the programs used was 75.83%. This is strong evidence to suggest that the proposed approach addresses the challenges of a complex model.

Limitations include: the user must have an accurate class diagram written in StarUML. However if desired the 'UMLParser' can be modified to accept various structures chosen by the UML tools. Another limitation is that class diagrams do not typically contain information such as boundary conditions and run time information. The class diagram is currently used as a pre validation to make sure all the objects in design are included. Future work would include obtaining information such as run time information from a sequence diagram or other suitable method. Currently boundary conditions are taken from the user via a GUI, in order to achieve complete automation; this to could be taken from a UML diagram or alternative method. Lastly a multi-objective fitness function would be preferable. After viewing the test cases generated, whilst they make a significant improvement in terms of object coverage, it would be beneficial to improve test cases based on variables such as: length of test, diversity in test set, execution time *et cetera*.

The challenges of a complex model have been addressed. Each of the five programs tested against, were variant in complexity. The improvement in efficiency appears not to be based on the size of the program and/or testing suite but on the quality of the current tests. Reinforcing that irrespective of the problem model complexity, automatic test case generation can be an improvement on manual test development.

7. ACKNOWLEDGMENTS

Thanks to Dr Chris Simons for his continued support.

8. REFERENCES

[1] Katanić, N., Nenadić, T., Dečić, S., Skorin-Kapov, L. 2010. *Automated generation of TTCN-3 test scripts for SIP-based calls.* MIPRO, 33rd International Convention.

[2] Tassay, G. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing.* Final Report.

[3] Kosindrdecha, N. and Daengdej, J. 2010. *A Test Case Generation Process and Technique.* Journal of Software Engineering. 265-287.

[4] Clark, J., Mander, K., Mcdermid, J., Tracey, N. 2002. *A Search Based Automation Test-Data Generation Framework for Safety-Critical Systems*. 1-41.

[5] Harman, M. 2010. *Why the Virtual Nature of Software Makes it Ideal for Search Based Optimization.*

[6] Shaukat, A., Lionel C. B., Hadi, H., Rajwinder K. 2010. *A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation.* IEEE Transactions on Software Engineering. 742-762.

[7] Fraser, G., Arcuri, A. 2011. *Whole Test Suite Generation.* Software Engineering, IEEE Transactions.

[8] Pavlov, Y., Fraser, G. 2012. *Semi-automatic Search-Based Test Generation.* IEEE Fifth International Conference on Software Testing, Verification and Validation.

[9] McMinn, P. 2004. *Search-based Software Test Data Generation: A Survey.* 105-156.

[10] Harman, M. 2007. *The Current State and Future of Search Based Software Engineering.* Future of Software Engineering, 2007. 342-357.

[11] Prasanna, M., Sivanandam, S., Sundarrajan, R., Venkatesan, R. 2005. *A survey on Automatic Test Case Generation*. Academic Open Internet Journal.

[12] Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J. 2006. *Automatic test generation: a use case driven approach.* IEEE Software Engineering. 140-155.

[13] Tonella, P., Potrich, A. 2003. *Reverse Engineering of the Interaction Diagrams from C++ Code.* IEEE International Conference on Software Maintenance, 159-168.

[14] Swain, A. K., Mohapatra, D. P., Mall, R. 2010. *Test Case Generation Based on Use case and Sequence Diagram.* Int. J. of Software Engineering. 21-52.

[15] Liu, S., Nakajima, S. 2010. *A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications.* Fourth International Conference on Secure Software Integration and Reliability Improvement. 147-155.

[16] Object Management Group. 2013. http://www.uml.org/ [2 June, 2013].

[17] Neto, A., de Freitas Rodrigues, R., Travassos, G. 2011. *Porantim-Opt: Optimizing the Combined Selection of Model-Based Testing Techniques.* ICSTW. 174-183.

[18] StarUML. 2005. http://staruml.sourceforge.net/en/ [26 January, 2012].

[19] Legg, S., Hutter, M., Kumar, A. 2004. *Tournament versus fitness uniform selection.* 2144-2151.

[20] Eiben, A.E., Smith, J.E. 2003. *Introduction to Evolutionary Computing.* 2nd edn.

[21] Forrest, S., Nguyen, T., Le Goues, C., Weimer, W. 2009. *A Genetic Programming Approach to Automated Software Repair*. 947-954.

[22] Arcuri. A., Yao, X. 2007. *Coevolving Programs and Unit Tests from their Specification*.

[23] Andrade, V.A., Errico, L., Aquino, A.L.L., Assis, L.P., Barbosa, C.H.N.R. 2008. *Analysis of Selection and Crossover Methods used by Genetic Algorithm-based Heuristic to solve the LSP Allocation.*