Efficient Probabilistic Testing of Model Transformations using Search

Louis M. Rose and Simon Poulding
Department of Computer Science
University of York
Deramore Lane, Heslington, York, YO10 5GH, UK
{louis.rose,simon.poulding}@york.ac.uk

***Abstract*—Checking the output of a test case for correctness—applying a test oracle—is challenging for many types of software, including model transformations. Decreasing the number of test cases that are executed during testing will therefore reduce the costs involved in testing a model transformation. However, there is a trade-off: to be confident that the transformation fulfils its specification requires the execution of sufficient test cases to fully exercise the transformation. In this paper, we demonstrate a process that derives a method of sampling random models which enables the test engineer to balance testing cost and testing efficacy. The output of the process is an optimised probability distribution over the models on which the transformation acts; test sets that efficiently exercise the transformation may then be derived by sampling models from the optimised distribution. Furthermore, we describe benefits and challenges of combining model-driven engineering and search-based software engineering tools and techniques, which include conflating metamodels with grammars to enable grammar-based search techniques over a set of models, and the need to increase the scalability of model-driven engineering tools to make them more amenable to search.**

I. Introduction

There are two major advantages in testing software with random input data. Firstly, if the software is repeatedly tested against the same set of test inputs, the software can become 'overfitted' to the test set: only a limited range of potential faults—those exercised by the fixed inputs specified in the test set—could ever be exposed during testing. By using a randomly-generated set of inputs each time the software is tested, a substantially wider range faults could be detected. Secondly, it is straightforward to generate test sets of any chosen size through random sampling of the inputs. This flexibility enables the amount of testing to be adapted to the time and budget available.

However, random sampling from an *arbitrary* input profile (a probability distribution over the set of the possible inputs to the software) is unlikely to be as effective as deterministic testing strategies such as structural testing. Structural testing uses the software's source code to guide the selection of test data: the test inputs are chosen so that every member of a chosen set of coverage elements—e.g. the set of statements, conditional branches, or paths—is exercised by at least one test case in the test set. This objective, called the adequacy criterion, is designed to efficiently distribute the testing effort throughout the software's code in a way that random testing from an *arbitrary* input profile cannot.

Thévenod-Fosse and Waeselynck [1], [2] describe a form of probabilistic structural testing, which they call *statistical testing*, which combines the advantages of random sampling with the efficiency of structural testing. Instead of sampling test inputs from an arbitrary input profile, the input profile is chosen so as to satisfy a probabilistic adequacy criterion: every coverage element must have a sufficiently high chance of being exercised by any one test input sampled from the profile. Thévenod-Fosse and Waeselynck show empirically that statistical testing is more effective at detecting faults than both deterministic structural testing and random testing using a uniform input profile.

The probabilistic adequacy criterion of statistical testing ensures that a *small* randomly-selected test set is sufficient to exercise every coverage element. The size of the test set is particularly important when the process of applying an *oracle*—checking that the observed output from the software for each test case is correct—is costly. This is often the case when the oracle is manual, e.g. a test engineer interpreting specification documents in order to verify the observed output.

It is, however, a complex task to derive input profiles that satisfy the adequacy profile in order to ensure a sufficiently small test set. A manual technique described by Thévenod-Fosse and Waeselynck assesses potential profiles by sampling inputs from them, executing an instrumented version of the software under test with the inputs so as to determine which coverage elements are exercised, and using the results to refine the profile. An analytical technique proposed by Denise et al. [3] specifies how to sample paths through the software according to a distribution that satisfies the criterion, but requires the application of constraint solving to locate test inputs that exercise the sampled path. Both of these techniques are limited in their scalability: the former dynamic approach in terms of the effort and cost involved, and the latter analytical approach by the computation required to solve constraints and the large proportion of infeasible path that are sampled.

In response to these limitations, Poulding demonstrated an automated search-based algorithm for deriving input profiles [4], [5]. By automating the process, Poulding's technique is potentially cheaper than the equivalent manual technique; the use of search facilitates the application of parallel computation for scalability; and, in contrast to Denise's technique that is specific to path coverage, the search-based algorithm can be applied to any chosen sets of coverage elements.

In this paper, we extend Poulding's technique to software for which the test inputs are concrete models. In this context, the input profile is a probability distribution over the set of models that conform to the metamodel which defines inputs to the software, and the software under test is the sequence of model transformation constituting a model-driven engineering (MDE) toolchain.

The objective is to derive a specific input profile (i.e. a probability distribution over the set of conforming models) so that a set of models sampled from the profile exercise all the features of the toolchain transformations. As Mottu et al. discuss in [6], when the output of the software under test is a model—as it is for an MDE toolchain—applying an oracle can be particularly difficult: not only is the output itself complex in nature, but the observed and predicted models may be syntatically different even when they are semantically equivalent. There is, therefore, a particular motivation to derive an *optimised* input profile that minimises the size of the test set required to exercise the MDE toolchain and hence the costs involved in applying the oracle.

To ensure the generality of our approach, we apply Poulding's search-based algorithm to the random generation of models using MDE languages and mechanisms that are independent of the problem domain. The models sampled from input profiles are described using the Human-Usable Textual Notation (HUTN) [7], [8]—a compact, metamodel-independent and human-readable model representation—and the search algorithm uses a grammar-based representation derived from the metamodel in a domain-agnostic manner.

This paper makes the following contributions:

- In terms of application: a novel search-based process for deriving, over the set of models conforming to a metamodel, a probability distribution that has desirable properties (in this case, that the distribution satisfies the adequacy criterion of statistical testing).
- In terms of implementation: an innovative use of Poulding's algorithm in conjunction with a grammar, derived from the metamodel, that emits models expressed using HUTN.
- Also in terms of implementation: the use of a service-oriented architecture to integrate the MDE toolchain (software under test) and search-based algorithm.
- An illustration and evaluation of the process on a realistic case study: the testing of a model transformation used in specifying the behaviour of a robot.
- Observations on the unexpected benefits of applying the process to the case study.
- Recommendations as to features required of MDE toolsets to facilitate the dynamic generation of test data by metaheuristic search in particular, and the application of search-based software engineering in general.

II. Background and Related Work

A. Statistical Testing

In this section, we expand on the adequacy criterion for statistical testing that was introduced above.

The adequacy criterion is expressed in terms of a set of coverage elements, $\mathcal{C}$. If the software's source code is a traditional language, the set of elements may be statements, conditional branches, conditions (atomic Boolean predicates in conditional or assignment statements), or unique paths through the code. For MDE toolchains, there is the potential to utilise additional coverage elements specific to the model transformation languages, where elements also include transformation rules. The choice of coverage elements often depends on the available testing budget and the desired 'thoroughness' of the testing. For example, exercising every statement will require fewer test cases than exercising every path through the code.

For a given input profile, each coverage element $c_i \in \mathcal{C}$ has a coverage probability, p_i, that the element is exercised (or 'covered') by a single input sampled at random from the input profile. Let p be the *minimum coverage probability*: the lowest value of the p_i across the set of elements. Then the adequacy criterion may be interpreted in two ways: either that p must be above a chosen target value, or that the input profile should be optimised so p is as high as possible given the resources available. In this paper we apply the latter interpretation.

Since the sampling of the inputs from the profile is probabilistic, it is not possible to *guarantee* that all coverage elements are exercised in a test set of a given size. Instead, we may specify that a confidence, Q, that all elements are exercised. Normally Q close to 1 is desirable: a value of 0.9 would indicate that 9 times out of a 10, all coverage elements are exercised.

For an input profile with a minimum coverage probability of p, the test set size, N, necessary to exercise all the coverage elements in the set with a confidence, Q, is [2]:

$$N = \frac{\log(1-Q)}{\log(1-p)} \qquad (1)$$

This equation quantifies the motivation in optimising the minimum coverage probability: the higher the value of p, the smaller the test set size, N.

B. Search-Based Algorithm

The search-based algorithm for deriving input profiles is initially described by Poulding in [4]. In [5], the algorithm is enhanced by the use of stochastic context-free grammars to represent input profiles, and it is this form of the algorithm that we utilise in this paper. We summarise relevant features of the algorithm here, and refer readers to the cited papers for further details of algorithm and the parameter settings.

The representation used by algorithm considers three aspects of the input profile. Firstly, the grammar defines the structural constraints on the inputs sample from the profile in terms of production rules. In the case of models, these constraints arise from conformity to the metamodel.

Secondly, the weights assigned to production rules to create a stochastic context-free grammar specify a probability distribution over the language defined by the grammar, and thus an input profile over the inputs of the software under test. An innovative aspect of the representation is that the weights are permitted to be conditional on recently applied production rules of a another grammar variable. While the grammar itself is context-free, the use of conditional weights introduces a limited form of context-sensitivity.

The third, and again innovative, aspect is the representation of ranges of scalar values, e.g. an attribute value in the integer range $[0, 32767]$. Representing each value as a separate grammar terminal would result in an impractically large search space; instead the scalar values are aggregated into a much smaller number of bins that partition the range such as $[0, 4209]$, $[4210, 23177]$, and $[23178, 32767]$. Each bin is represented by a terminal in the grammar, and in combination with weights assigned to production rules, this enables the representation of a probability distribution over the scalar range.

The search method used to optimise the input profile is random-mutation hill climbing; it modifies on the latter two aspects of the representation while leaving the grammar production rules unchanged.

The search starts from an input profile with random weights and bin partitions. At each iteration a small number of near-neighbours to the current profile are created by making minor changes to rule weights, the conditional dependence of weights, the number of bins, and the length of bins of one variable in the grammar.

The fitness metric used to guide the search is the minimum coverage probability of the neighbours. A number of inputs are sampled from the neighbour profile, and the inputs are used to execute the software under test. The software is instrumented to provide information as to the coverage elements exercised by each input (without changing its other functionality), and this information enables an estimate to be made of the minimum coverage probability induced by the neighbour profile.

If the best of the neighbours (the input profile with the highest estimated minimum coverage probability) is fitter than the current profile, the former replaces the latter in the next iteration of the hill climb. In a practical application of the algorithm, the process will terminate after a chosen number of iterations set according to the time the test engineer has available to run the algorithm.

C. Generating Data for Testing Model Transformations

Testing a model transformation involves constructing a (set of) models that conform to the metamodel that describes the set of inputs to the transformation. Test models can be generated manually or randomly. Lin et al. [9] and García-Domínguez et. al. [10] describe testing frameworks in which test models can be specified manually with a textual notation or with a modelling tool. Compared to other areas of MDE, there is relatively little work on randomly generating models. Our previous work [11] demonstrates the use of a form of grammatical evolution to generate models of video game characters that exhibit desirable characteristics. The purpose of the work described in this paper is rather different: we seek to increase the efficiency of testing MDE toolchains by deriving a probability distribution over a set of models conforming to a metamodel; and not to generate individual models with particular characteristics. Ehrig et al. [12] and Mougenot et al. [13] demonstrate the use of graph grammars to generate random models for testing model transformations. Neither approach considers metamodel constraints [14]. Additionally, neither approach seeks to increase the efficiency of testing a model transformation: Ehrig et al. approach adequacy by seeking to generate a large set of test models, while Mougenot et al. apply a uniform random sampling to generate very large models with the goal of stress testing transformations.

D. The Human-Usable Textual Notation

In this paper, we derive a probability distribution over a stochastic context-free grammar that emits models represented with the Human-Usable Textual Notation (HUTN) [7] [8]. HUTN is intended to be human-readable and is otherwise equivalent to other model representations used in contemporary MDE tools, such as XML Metadata Interchange (XMI).

III. Envisaged Process

We now present our envisaged process for producing an optimised input profile for testing an MDE toolchain (figure 1). The remainder of this section discusses each task in the process by reflecting on a case study in which we have derived an optimal input profile for a model transformation between two domain-specific languages for programming a Lego Mindstorms[1] robot.

Our envisaged process comprises the following four tasks:

a) *Developing the MDE toolchain*: the software under test is developed manually. This task involves typical MDE development activities (e.g., defining or selecting a metamodel; writing model management programs such as model transformations), selecting coverage elements for statistical testing of the MDE toolchain, and deploying the toolchain in a manner that increases throughput to facilitate the large amount of sampling required by the optimisation task.
b) *Optimising the input profile*: a probability distribution over the set of models conforming to the input metamodel is automatically derived. The scale and complexity of optimising the input profile necessitates a large amount of sampling, cannot be completed manually and consequently we have developed and applied structures that automate most of this task.
c) *Validation of the input profile*: a potential solution (i.e., an optimised input profile) is analysed manually. The analysis involves assessing the choice of coverage elements. The aim of this task is to increase confidence in the input profile before using it for testing. As discussed below, we have found that validating the input profile led to the

[1] http://mindstorms.lego.com/

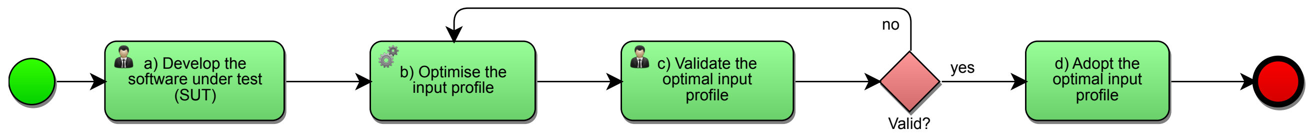
a) Develop the software under test (SUT)
b) Optimise the input profile
c) Validate the optimal input profile
no
yes
Valid?
d) Adopt the optimal input profile

Fig. 1. Our envisaged process for producing an optimised input profile, specified in BPMN 2.0.

identification of defects in the software under test, which is an unexpected benefit of our process.

d) *Adoption of the input profile*: the optimised input profile is incorporated into the test suite of the software under test. This task involves altering the structures (and perhaps processes) used for testing by, for example, producing a set of test models by sampling from the optimised input profile. Once an optimal input profile has been derived, it can be re-used for different types and instances of testing.

A. Developing the MDE Toolchain

This task involves developing the MDE toolchain. Two important outputs from this task are a deployment of the MDE toolchain (which is used in the optimisation task to assess the fitness of candidate input profiles) and an input metamodel (which is used in the optimisation task to construct a stochastic context-free grammar for the search-based algorithm). To allow optimisation, the MDE toolchain might need to be adapted to provide additional information about the extent to which the MDE toolchain has been exercised by an input (i.e., code coverage data), and tuned to increase the rate at which models can be processed by the toolchain (i.e., increase throughput).

For the Lego Mindstorms case study, our MDE toolchain comprised a single model-to-model transformation that consumes a platform-independent representation of our robot's behaviour, and produces a model containing details specific to the LEJOS[2] Java API for Mindstorms robots. The transformation was encoded in the Epsilon Transformation Language (ETL) [15], comprised 128 lines of code and 10 transformation rules. We investigated three problem-independent coverage elements: transformation rule coverage (whether or not a transformation rule was fired whilst transforming an input model), branch coverage, and condition coverage. We also investigated a problem-specific coverage element that indicated whether or not the input model defined a sufficient amount of behaviour for the robot. ETL provides built-in support for determining rule coverage, and the other types of coverage were determined by manually editing the model transformation code.

To allow optimisation, we adopted principles from service-oriented architecture to integrate the MDE toolchain (software under test) and search-based algorithm. In particular, we created a web service interface for the MDE toolchain and deployed it to the Heroku[3] cloud platform. Consequently, the integration between the software under test and search-based algorithm was language-independent (because inter-process communication was achieved via HTTP), and throughput could be increased via horizontal scaling of the MDE toolchain (i.e., provisioning of additional instances of the MDE toolchain to facilitate parallelisation of parts of the optimisation task).

[2]http://lejos.sourceforge.net

[3]http://www.heroku.com

B. Optimising the Input Profile

This task involves deriving an input profile – a probability distribution over the set of input models for the MDE toolchain – that can used to produce input models for testing. As discussed in section I, manually deriving an input profile is difficult and costly. Instead, we have adapted the search-based algorithm (discussed above in section II-B) to automatically derive an optimised input profile from a metamodel. To enable search, we have developed a mechanistic, though currently manual approach for translating a metamodel into the stochastic context-free grammar which is optimised by the search-based algorithm. When sampled, the resulting grammar produces a string representation of a model conforming to the metamodel. As discussed below, the translation between metamodel and context-free grammar is objective: it does not use metamodel-specific information and can be performed for any MDE metamodel.

For the Lego Mindstorms case study, we have manually translated the input metamodel to a context-free grammar that emits HUTN when sampled. Essentially, translation involved locating patterns in the metamodel and constructing one or more production rules in the context-free grammar. For example, one pattern involves constructing for every class in the metamodel a new production in the grammar. The production emits a HUTN string that instantiates the class, assigns values for mandatory features and might also assign values for optional features. A further pattern involves constructing for every non-containment association in the metamodel a production that emits a HUTN association block (i.e., specifies a connection between two model elements). The bins terminals in the grammar (discussed above in section II-B) enables a search-amenable representation of scalar ranges, such as integer attribute values.

Epsilon, the MDE environment used to construct the software under test, has built-in support for parsing HUTN and hence it was straightforward to adapt the software under test to consume models represented with HUTN. Very few MDE environments can parse HUTN out-of-the-box and hence an important piece of future work is to explore the extent to which our approach generalises to other model interchange formats, such as XMI.

C. Validation of the Input Profile

This task involves investigating the validity of the optimal input profile. Models constructed by sampling the input profile are inspected manually to understand the way in which coverage elements are being exercised. The primary purpose of analysing the input profile in this manner is to validate the choice of coverage elements used for optimisation. In applying the process to the Lego Mindstorms case study, we have found a secondary and unexpected benefit of the validation task: models sampled from an optimised input profile can be used to validate part of the specification for the software under test (i.e., the metamodel) and hence to identify defects.

For the Lego Mindstorms case study, we have used the four types of coverage element discussed above (rule, branch and condition coverage, as well as a problem-specific coverage element, proportion coverage). Our initial work on the case study involved a single coverage element (rule coverage). Reasoning about sample models from the initial optimal input profile highlighted that additional coverage elements were needed to increase confidence in the correctness of the MDE toolchain. (In practice, the choice of coverage elements may be based on experience with similar software or determined by the policy of the organisation.)

A further benefit of the validation task was that sampling from an optimised input profile highlighted defects in the software under test. The first kind of defects that we identified were due to the input metamodel containing type information that was not sufficiently restrictive (e.g., “integer” rather than “positive integer”; “zero-to-many” rather than “one-to-many”). The second kind of defects were due to missing domain-specific constraints on the metamodel (e.g., “every zone must be wholly contained in the world”). Arguably, both kinds of defect would have been difficult to detect with deterministic testing, particularly if the test engineer has been involved in designing the metamodel.

D. Adoption of the Input Profile

This task involves integrating the optimal input profile into the testing structures and processes for the software under test. The same optimal input profile can be used to support different types of testing. When generating test data is costly, we might use the optimal input profile to derive a minimal set of input models that cover each coverage element (e.g., transformation rules, statements, branches) as frequently as possible, and hence seek to reduce the overall cost of testing. Alternatively, we might use a testing process that involves sampling from the optimal input profile each time the software is tested in order to obtain new random test data.

For the Lego Mindstorms case study, generating test data involves manual observations of robot behaviour (i.e., a manual oracle) and each test may take several minutes to complete. Therefore, we sought to minimise the number of test cases required in order to maximise the frequency with which each coverage element was covered. The empirical evaluation of the search-based algorithm, below, demonstrates the extent to which we were able to achieve this goal.

IV. Evaluation

A. Method

In this section, we evaluate the potential efficiency of the input profiles derived by the process described above and applied to the Lego Mindstorms case study. The objective was coverage of the ‘strongest’ set of coverage elements in the MDE toolchain: a combination of conditions in the transformation code, conditions in the problem-specific constraints on the metamodel, each of all ten transformation rules, and of both outcomes of any transformation rule guard—a total of 42 coverage elements.

Our assessment metric was the estimated minimum coverage probability (i.e., the fitness) of the best profile found by the hill-climbing search in 80 iterations. At each search iteration 10 candidate input profiles are evaluated, and each fitness evaluation samples over 300 inputs from the candidate profile. A single algorithm trial of 80 iterations therefore involves approximately 240,000 executions of the MDE toolchain; on the hardware used to run the experiments this takes approximately 17 hours. This number of executions is much smaller than in previous application’s of Poulding’s algorithm, but is limited by the much longer time required to execute the software under test and return the coverage information.

The search algorithm is stochastic, and therefore in order to estimate the average performance, 8 independent algorithm trials were run, each with a different seed to the pseudo-random number generator. The algorithm parameter settings used were those described in [5]; space precludes their description here. No attempt was made to tune the parameter settings to this specific case study as tuning would not occur in a real-world application of the process.

In order to assess the difficulty of the problem for search, a further 8 independent trials were run in which random search was used in place of the hill-climbing search method. This was implemented by instead of creating neighbours at each iteration of the hill climb, generating entirely random input profile that were unrelated to the current profile. The number of executions of the MDE toolchain were therefore the same, and the minimum coverage probability was recorded was the best found at any point during the algorithm.

Finally, a third set of trials were performed to assess the minimum coverage probability of a uniform input profile. This is a natural choice for the type of *arbitrary* input profile that is used for traditional random testing, and is constructed by setting all weights in the grammar to the same value.

For all three algorithm variants (hill climbing, random search, and the uniform profile), a limit was placed on the size of the model sampled by the grammar. If the model contained more than 64 objects (including the three pseudo-objects in the HUTN header), then it was not used to execute the MDE toolchain, and the model was assumed to cover none of the elements. This is designed to encourage parsimony in the models sampled from the input profile, on the basis that it would be particularly expensive to apply a manual oracle to such large models.

TABLE I
THE COVERAGE ATTAINED BY EACH ALGORITHM VARIANT.

Algorithm Variant	Minimum Coverage Probability	Resultant Test Size
Hill-Climbing Search	0.0762	29
Random Search	0.0447	50
Uniform Profile	0.00828	277

B. Results

Table I summarises the results[4]. The column 'Minimum Coverage Probability' lists the median minimum coverage probability attained across the 8 trials for each algorithm variant. The value in the column 'Resultant Test Size' is the value, N, calculated from equation 1 using the median minimum coverage probability attained for each method and a confidence of 0.9. A single test set of this size consisting of inputs sampled from the input profile optimised by the algorithm would therefore—9 times out of 10—exercise all 42 coverage elements at least once.

The difference in minimum coverage probabilities between the optimised input profile resulting from the hill climbing algorithm and the uniform profile is very large. Moreover, the difference is statistically significant: a p-value of much less than 1% using the Mann-Whitney-Wilcoxon rank-sum test. The difference is particular clear when comparing the equivalent test sizes: the optimised input profile permits a test set of less than a ninth of the size of one that would be required if the uniform profile were used to sample the input data. Therefore, the testing process using the optimised input profile would be much cheaper.

The difference between hill climbing and random search is not so large. Although the hill climbing algorithm attains a better minimum coverage probability, the difference is much less statistically significant: the p-value is 6.8%. One possible interpretation of this result is that the search problem is not particularly hard. However, in previous work by Poulding, the hill-climbing algorithm ran for many more iterations before attaining the best input profile, and we speculate that a greater difference between the hill-climbing and random search methods would be observed if the algorithm were run for more iterations.

V. Conclusion

We have demonstrated the viability of a process for efficient random testing of MDE toolchains, and of applying a search-based algorithm to derive input profiles for MDE metamodels. By applying our process to a realistic case study, we discovered an unexpected benefit: defects in the software under test were identified before the final optimal input profile was derived, which highlights an advantage of making explicit (part of the) specification for the software under test as a metamodel. A challenge of combining MDE tools with search techniques has been balancing the large degree of sampling required by the search-based algorithm with the difficulty of increasing the throughput of model transformations.

A major objective of future work is to achieve greater automation and scalability. We will investigate the use of a model transformation to automate the translation of metamodel to stochastic context-free grammar, and the feasibility of replacing the grammar with a metamodel annotated with the probabilistic weights. Additionally, we will further develop our service-oriented approach to parallelising the sampling of the software under test to achieve greater scalability and reliability. In particular, we anticipate creating a more robust framework for deploying ETL transformations to cloud platforms. We will use the automated and scalable structures to allow us to conduct an objective evaluation of statistical testing for MDE.

Acknowledgement

This work was funded in part by EPSRC grant EP/J017515/1, DAASE: Dynamic Adaptive Automated Software Engineering.

References

[1] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *J. Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 5–26, 1991.
[2] ——, "Statemate applied to statistical testing," in *Proc. Int'l Symp. Software Testing and Analysis (ISSTA'93)*, 1993, pp. 99–109.
[3] A. Denise, M.-C. Gaudel, and S.-D. Gouraud, "A generic method for statistical testing," Laboratoire de Recherche en Informatique (LRI), CNRS - Univ. de Paris Sud, Tech. Rep. 1386, April 2004.
[4] S. Poulding and J. A. Clark, "Efficient software verification: Statistical testing using automated search," *IEEE Trans. Software Eng.*, vol. 36, no. 6, pp. 763–777, 2010.
[5] S. Poulding, "The use of automated search in deriving software testing strategies," Ph.D. dissertation, University of York, 2013, (manuscript in preparation).
[6] J.-M. Mottu, B. Baudry, and Y. Traon, "Model transformation testing: oracle issue," in *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, april 2008, pp. 105 –112.
[7] OMG, "Human-Usable Textual Notation 1.0 Specification," Available at: http://www.omg.org/technology/documents/formal/hutn.htm, 2004.
[8] L. Rose, R. Paige, D. Kolovos, and F. Polack, "Constructing models with the Human-Usable Textual Notation," in *Proc. MoDELS*, ser. LNCS, vol. 5301. Springer, 2008, pp. 249–263.
[9] Y. Lin., J. Zhang, and J. Gray, "A testing framework for model transformations," *Model-Driven Software Development*, pp. 219–236, 2005.
[10] A. García-Domínguez, D. Kolovos, L. Rose, R. Paige, and I. Medina-Bulo, "Eunit: A unit testing framework for model management tasks," in *Proc. MoDELS*, ser. LNCS, vol. 6981. Springer, 2011, pp. 395–409.
[11] J. Williams, S. Poulding, L. Rose, R. Paige, and F. Polack, "Identifying desirable game character behaviours through the application of evolutionary algorithms to model-driven engineering metamodels," in *Proc. SSBSE*, ser. LNCS, vol. 6956. Springer, 2011, pp. 112–126.
[12] K. Ehrig and T. G. Küster, J.M., "Generating instance models from meta models," *Software and System Modeling*, vol. 8, no. 4, pp. 479–500, 2009.
[13] A. Mougenot, A. Darrasse, X. Blanc, and M. Soria, "Uniform random generation of huge metamodel instances," in *Proc. ECMDA-FA*, ser. LNCS, vol. 5562. Springer, 2009, pp. 130–145.
[14] J. Williams and S. Poulding, "Generating models using metaheuristic search, in Proc. York Doctoral Symposium," University of York, Tech. Rep. YCS-2011-468, October 2011.
[15] D. Kolovos, R. Paige, and F. Polack, "The Epsilon Transformation Language," in *Proc. ICMT*, ser. LNCS, vol. 5063. Springer, 2008, pp. 46–60.

[4]Unsummarised data from the evaluation is provided at: http://www.cs.york.ac.uk/~smp/supplemental/