Using Traceability to Enhance Mutation Analysis Dedicated to Model Transformation

Vincent Aranega
LIFL - UMR CNRS 8022, INRIA
University of Lille 1
Lille, France
Email: vincent.aranega@lifl.fr

Jean-Marie Mottu
LINA - UMR CNRS 6241
University of Nantes
Nantes, France
Email: jean-marie.mottu@univ-nantes.fr

Anne Etien, Jean-Luc Dekeyser
LIFL - UMR CNRS 8022, INRIA
University of Lille 1
Lille, France
Email: {anne.etien,
jean-luc.dekeyser}@lifl.fr

***Abstract*—Techniques initially used for programs require modifications to be properly used with to model transformation characteristics. Mutation analysis is one of these techniques. It aims to qualify a test data set by analyzing the execution results of intentionally faulty program versions. If the degree of qualification is not satisfactory, the test data set has to be improved. This step is currently relatively fastidious and manually performed.**

In this paper, we propose an approach based on traceability mechanisms to ease the test model set improvement in the mutation analysis process. A benchmark shows that the part of the input model to change is automatically and quickly identified. A new model is then created in order to raise the quality of the test data set.

***Index Terms*—test; model transformation; mutation analysis; traceability**

I. Introduction

When a program written in C has not the expected behavior or is erroneous, the programmers look for the faults in their program. Indeed, they trust in the compiler. This latter has been largely tested because a fault in a compiler may spread over lot of programs since a compiler is used many times to justify the efforts relative to its development. Similarly, model transformations that form the skeleton of model based system development and so enable to generate code from high level model specifications have to be largely tested and trustworthy.

Model transformations are usually considered as programs and may be tested as so. However, they rely on metamodels and manipulate models as input and output data. Using such data structures implies specific operations that do not occur in traditional programs such as navigating the input/output metamodels or filtering model elements in collections. Thus, classical but also specific faults may appear in model transformations. For instance, the programmer may have navigated a wrong association between two classes, thus manipulating incorrect class instances of the expected type. The emergence of the object paradigm has implied an evolution in the verification techniques [1]. Similarly, verification techniques have to be adapted to model transformation specificity to make profit of the model paradigm. New issues relative to the generation, the selection and the qualification of input model data are met.

There exist several test techniques. In this paper, we will only focus on mutation analysis. Mutation analysis relies on the following assumption: if a given test data set can reveal the fault in voluntarily faulty programs, then this set is able to detect involuntary faults. Mutation analysis [2] aims to qualify a test data set for detecting faults in a program under test. For this purpose, faulty versions of this program (called *mutants*) are systematically created by injecting one single fault by version. The efficiency of a given test data set to reveal the faults in these faulty programs is then evaluated. If the proportion of detected faulty programs [3] is considered too low, new and adapted data tests have to be introduced [4]. To apprehend the model transformation specificity, the mutation analysis process may be adapted. For each mutant, one fault is injected in a transformation rule [5]. The quality of the test model set is then evaluated and possibly enhanced.

Only the test data improvement step of the mutation analysis process is apprehended in this paper. The creation of new test models relies on a deep analysis of the existing test models and the execution of the undetected faulty transformations. Currently, this work is manually performed and fastidious; the tester deals with a large amount of information. In this paper, we propose an approach to fully automate the information collection. This automation relies on traceability mechanisms enhanced with mutation analysis characteristics. An algorithm is proposed to effectively collect the required and sufficient information. Then, the collected information is used to create new test models. Our enhanced traceability mechanisms helps to reduce the testers intervention to particular steps where their expertises are essential.

This paper is composed as follows. Section II presents mutation analysis to qualify test data set in model transformation testing. Section III describes our metamodels, foundations of our approach to improve test data set. Section IV validates our approach with transformations from the Gaspard framework. Section V introduces works related to the qualification and the improvement of the test data set. Section VI draws some conclusions and introduces future works.

978-0-7695-4384-0/10 $26.00 © 2010 IEEE
DOI 10.1109/MoDeVVa.2010.15

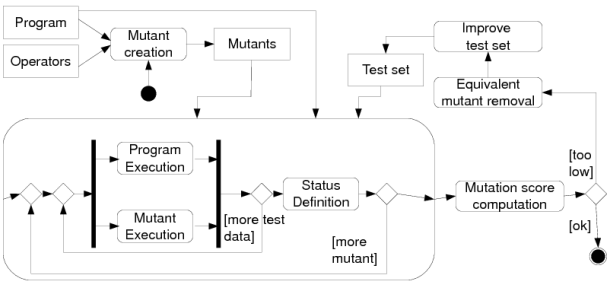

Fig. 1. Mutation analysis process

II. Mutation Analysis to Qualify Test Data Set

Assuring that a program is undoubtedly fault free is a difficult task requiring a lot of time and expertise. However, qualifying a test data set (*i.e.* estimate its pertinence and its effectiveness) is easier. If this estimation is considered too low, the test set must be improved. In the following subsections, we briefly describe the mutation analysis process [2], one way to qualify a given test data set. We then explain why that software testing method has to be adapted to the model paradigm.

In the following subsections, we briefly describe the mutation analysis process [2], one way to qualify a given test data set. We then explain why program testing approaches need to be adapted to the model paradigm.

A. Mutation Analysis Process

Preliminary to the mutation analysis process, variants (P_1, P_2,..., P_k) (called mutants) of the program P under test have to be created by injecting one atomic change. In practice, each change corresponds to the application of a single mutation operator on P. Then, P and all the mutants are successively executed with each test data of the set to be qualified. If the results returned by P differ from anyway from those returned by some P_i, these mutants are said to be *killed*. The faults introduced in those P_i were indeed highlighted by the test data. In the other case, if P returns the same results as some P_j, they are said to be *live* mutants. A mutant may be alive for two reasons: (1) P and P_j are actually equivalent programs and no test data will distinguish them (*e.g.* the fault has been inserted in dead code); or (2) the test data set is not sensitive enough to highlight that fault. In that latter case, the test data set has to be improved until it kills each mutant or it only leaves live mutants that are equivalent to P [2]. The mutation analysis process is stopped when the test data set is qualified *i.e.* when the ratio of killed mutants, also called the mutation score, reached 100 % or when it rises above a threshold beforehand fixed. Figure 1 sketches the mutation analysis process.

B. A largely manual process

Part of the mutation analysis process is automatic but work remains for the tester.

- The mutant creation can be automated. However, the operators are specific to the language used in the program to test. For each new language a new definition and implementation of the mutation operators have to be performed.
- The execution of P and its associated mutants with the test data is obviously automated as well as the comparison of the results.
- The analysis of a live mutant is manual up to now. Indeed, on the one hand, the automatic identification of equivalent mutants is an undecidable problem [2], [6]. On the other hand, the test data set improvement can be difficult.
- The improvement of the test data set is manually performed. Indeed, the unrevealed injected fault should be analyzed both statically and dynamically in order to create a new test data. Moreover, the new test data produced has to imply a behavior that is different from the original program, for at least one mutant.

The purpose of this paper is to help in the automation of the test data set improvement in case where test data are models and program is a model transformation. But us let explore in the next subsection the specificity of model transformation testing.

C. Adaptation to Model Transformation

Model transformations can be considered programs and therefore techniques previously explained can be used. However, the complexity and the specificity induced by the data structures (*i.e.* models conform to their metamodels) manipulated by the transformations imply modifications in the mutation analysis process described in the subsection II-A.

Each step of the mutation analysis process has to be adapted to model transformations. [7] deals with the generation of test models. In [5], dedicated mutation operators have been designed independently from any transformation language. They are based on three abstract operations linked to the basic treatments of a model transformation: the navigation of the models through the relations between the classes, the filtering of object collections, and the creation and the modification of the model elements. The execution of the transformation under test T and its mutants $T_1, T_2, \ldots, T_k$ differs from the execution of a program but remains common. The comparison of the output model produced by T and those produced by the T_i can be performed using adequate tools such as EMFCompare [8]. If a difference is raised by EMFCompare, the mutant is considered *killed*, otherwise new test models are built to kill the (non equivalent) live mutants.

The remainder of this paper focuses on this last part of the mutation analysis process where new test models are created

III. Traceability, a Means to Automatically Collect Information

Considering that creating a new test model from another one is easier than from scratch, the issue of the test set improvement raises three questions:

- Among all the existing couples (test model, mutant), which ones are relevant to be studied?

- What should the output model look like if the mutant was killed? *i.e.* what could be the difference we want to make appear in the output model?
- How to modify the (input) test model to produce the expected output model and thus kill the mutant?

To help the tester to answer these questions, we provide a method based on a traceability mechanism.

A. Traceability for Model Transformation

According to the IEEE Glossary, *Traceability allows one to establish degrees of relationship between products of a development process, especially products bound by a predecessor-successor or master-subordinate relationship* [9]. Regarding MDE and more specifically model transformations, the trace links elements of different models by specifying which ones are useful to generate others.

Our traceability approach [10], [11] relies on two metamodels: a local trace metamodel and a global trace metamodel. The first refers to the trace for model to model transformations whereas the second refers to the trace for transformation chains. In this paper, we focus on model to model transformation testing and do not consider transformation chain. Therefore, only the local trace metamodel is presented (cf. Figure 2).

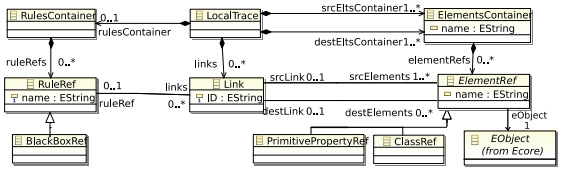

Fig. 2. Local Trace Metamodel

The local trace metamodel is built around two main concepts: *Link* and *ElementRef* expressing that one or more source elements are possibly bound to target elements. Furthermore, for each link, the transformation rule producing it is traced using the *RuleRef* concept. Finally, for implementation facilities, an *ElementRef* has a reference to the real object in the source or target model. As our environment is based on the Eclipse platform, models are implemented with EMF, the reference named *EObject* is an import of the *ECore* metamodel. The local trace metamodel and local trace models are independent of any transformation language. However, the generation of the local trace model strongly depends on the used transformation language.

For each *Link* instance, the involved elements of the input or output models are clearly identified thanks to the *ClassRef* directly referring the *EObject*. A continuity between the traceability and the transformation worlds is ensured. Furthermore, the transformation rule that has created a link is associated to it via the *ruleRef* reference. Each time a rule is called a unique new *Link* is created. Thus, from a rule, the *localTraceModel* enables the tester to identify, for each call (*i.e.* for each associated link), two sets of elements: those of the input model and those of the output model created by the rule. In the case of a faulty rule, these sets respectively correspond to the elements to modify and the elements that may be different if the mutant is killed.

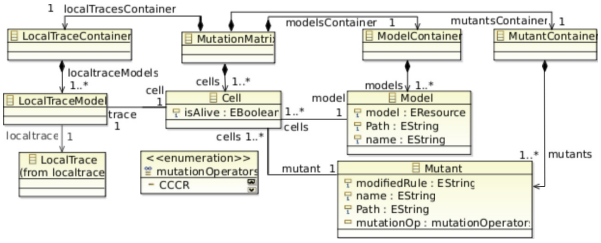

Fig. 3. Mutation Matrix Metamodel

B. Mutation Matrix Metamodel

Mutation analysis results and traces information are combined to automate a part of the test data set improvement process. Mutation analysis results are usually gathered in a matrix. Each cell indicates if an input model has killed a given mutant or not. A mutant is alive if none of its corresponding cells indicate a killing. From information contained in all the cells concerning a mutant, it can be deduced if it is alive or not.

Links between mutants, test models and their traces are managed using a dedicated matrix at a model level. The advantages are multiple. A cell corresponds to an abstraction of the execution of a mutant T_i for the test model D_k. By associating its trace to each cell, the matrix model becomes a pivot model. In this way a continuity is ensured between the traces, the test models and the information gathered in the mutation matrix. The navigation is eased between the different worlds. Moreover, the mutation matrix benefits from tools dedicated to models. Thus, the mutation matrix model is automatically produced from the results of the comparisons between the model produced by the original transformation and the one generated by T_i.

The mutation matrix metamodel, presented Figure 3, is organized around three main concepts:

- *Mutant* which refers to mutants created from the original transformation. The mutants have one rule ($modifiedRule$ attribute, it is a string to remain independent from any transformation language) modified thanks to one mutation operator ($mutationOp$ attribute).
- *Model* which refers to input test data.
- *Cell* which corresponds to an abstraction of the couple (mutant T_i, test model D_k). Its value ($false$ or $true$) of the property $isAlive$ specifies the state (killed or live respectively) of the *Mutant* T_i regarding to the specific *Model* D_k. The *LocalTraceModel* corresponding to the execution of T_i with D_k is thus associated to the *Cell*.

The matrix model is generated during the mutation analysis process. It is the foundation of the test model improvement process presented in the next subsection.

C. Data Improvement Process Assisted by Traces

The data improvement process is composed of three steps:

- step 1: identifying a live mutant
- step 2: finding an appropriate existing test model
- step 3: improving this test model in order it kills the mutant

A mutant is alive if no test model has killed it. Live mutants can thus be easily and automatically identified by exploring the matrix cells. Identifying a good candidate, among the test models, to kill a given live mutant is more difficult. Our approach relies on the principle that test models for which the faulty rule of the mutant has been called are better candidates. Indeed, the conditions to apply this rule were satisfied. Our traceability mechanism helps us to identify these models and for each of them to highlight the elements impacted by the faulty rule. The algorithm 1 implements this part of the improvement process (*i.e.* corresponding to step 2 and gathering information to perform step 3).

Algorithm 1 Information Recovering for a Live Mutant

```
1: trace ← null
2: rule ← null
3: modifiedRule ← mutant.modifiedRule
4: modelsHandled ← ∅
5: eltsHandledSrc ← ∅
6: eltsHandledDest ← ∅
7: for each mutant.cells do
8:     trace ← cell.trace
9:     rule ← trace.findRule(modifiedRule)
10:    if rule ≠ null then
11:        modelsHandled += cell.model
12:        tempEltsSrc ← ∅
13:        tempEltsDest ← ∅
14:        for each rule.links do
15:            tempEltsSrc += link.srcElements
16:            tempEltsDest += link.destElements
17:        end for
18:        eltsHandledSrc += tempEltsSrc
19:        eltsHandledDest += tempEltsDest
20:    end if
21: end for
```

The first five lines correspond to the initialization of the different variables:

- the $trace$ variable stores the trace associated to the execution of the mutant T_i for a given test model D_k
- $rule$ refers to a RuleRef in the trace model
- $modifiedRule$ is a String initialized with the name of the modified rule associated to T_i
- the $modelsHandled$ variable is a model list containing the test models for which the execution of the mutant requires the modified rule
- the $eltsHandledSrc$ and $eltsHandledDest$ variables are similar to the previous one. They contain lists of input (respectively output) elements (one list by test model) that are involved in the application of the faulty rule.

The algorithm then scans each cell relative to the studied mutant. The trace corresponding to the execution of T_i on one input model D_k is stored (line 8). The trace model is navigated to check if the modified rule has been called during the corresponding transformation. This search is performed through the $findRule$ method (not detailed in the algorithm). This method explores the $RulesContainer$ of the *LocalTraceModel* associated to the cell until it finds the *RuleRef* instance whose name corresponds to the one of the faulty rule (*i.e.* the assigned value of the modifiedRule property of the *Mutant*). This method returns a *RuleRef* instance or null if the rule doesn't appear in the trace. The result is stored in the $rule$ variable (line 9). If the content of the $rule$ variable is $null$, the analysis stops here for this cell and goes on with the next one. In the other hand, the model D_k is stored in the $modelHandled$ (line 10). For each link associated to the $rule$, the list of the input model elements ($srcElements$) is stored in the $eltsHandledSrc$ variable using the temporary variable $tempEltsSrc$. The management of the output model elements is performed from the same way. (line 12 to 17).

For a given live mutant T_i, this algorithm provides: (1) some test models ($modelsHandled$) (2) their elements ($eltsHandled$) involved in the application of the faulty rule and (3) the elements of the output models created by this rule. If the content of the $modelHandled$ variable is empty, the faulty rule has never been called, whatever the test model. A new model has to be created, possibly from scratch, containing elements satisfying the application of the faulty rule. On the other hand, if the $modelHandled$ variable is not empty, the faulty rule has been called at least once. However, since the mutant is alive, this rule has never produced a result different from the one generated by the original transformation T. A new test model is created by adapting the considered test model.

Our approach helps the tester to drastically reduces the field of the required analysis to create a new model. For a given live mutant, the set of models and the set of the model elements is reduced to only those impacted by the application of the faulty rule. Even if a part of the improvement process remains manually performed, our approach clearly eases the tester work.

IV. Example

This section aims to validate our approach on a real case study; the transformation from the UML metamodel enhanced with the MARTE profile to the MARTE metamodel. The MARTE profile is the OMG standard for the modeling and the analysis of real time embedded systems [12]. Both metamodels involved in the transformation are composed of around 200 metaclasses. This transformation is written in QVTO and corresponds to around 1500 lines of code. It is part of a larger framework Gaspard for the co-design of embedded system [13].

```
mapping UML::Port::port2flowPort() : GCM::FlowPort
  when { ... }
  {
    name := self.name;
    direction :=
        if self.getValue(stereotype, 'direction')
            .oclAsType(EnumerationLiteral)
              .name.equalsIgnoreCase('in')
          then GCM::DirectionKind::_in
          else if self.getValue(stereotype, 'direction')
                    .oclAsType(EnumerationLiteral)
                      .name.equalsIgnoreCase('out')
          then GCM::DirectionKind::_out
         -- else GCM::DirectionKind::_inout -- orig
          else GCM::DirectionKind::_out  -- mutant
         endif;
    ...
  }
```

Fig. 4. Mutate *port2flowPort* rule excerpt

TABLE I
TEST DATA ELEMENTS HANDLED BY THE MODIFIED RULE

Test Data	Src. Elements	Dest. Elements
simple_ reshape.uml	*p1:Port* *p2:Port*	*p1:FlowPort* *p2:FlowPort*
simple_tiler.uml	*p1:Port* *p2:Port* *p3:Port*	*p1:FlowPort* *p2:FlowPort* *p3:FlowPort*
simple_data_ allocation.uml	*p:Port*	*p:FlowPort*
ex4simple- malloc.uml	*host_variable* *kernel_variable*	*host_variable* *kernel_variable*
compound.uml	*p1:Port* *p2:Port*	*p1:FlowPort* *p2:FlowPort*

A. Application of our Approach

Due to the size of the involved metamodels and the large scope of the transformation, the amount of work is relatively huge. We thus decided to concentrate the faults injected in a limited part of the transformation. The mutants have been manually created. Indeed, even if the mutation operators are generic, their implementation and the automation of the mutation creation must be adapted for each transformation language. Such works have not yet been done for QVTO standard language. 35 mutants corresponding to the 10 mutation operators and initially 32 test models have been defined and the mutation matrix has been set up. An automatic exploration of the full matrix enables the tester to identify the killed mutants and the live ones. The remainder of the algorithm is applied in order to create a new test model that will kill this mutant.

Thanks to the mutation matrix, the rule *port2flowPort* of this mutant where a fault has been injected is easily identified. The original transformation sets the value of the produced port to *_inout*, whereas the mutant sets it to *_out*. In the Figure 4, the original piece of code is marked by the *orig* flag and the modified one by the *mutant* flag.

The algorithm 1 enables the identification of the 5 test models, for which the *port2flowPort* rule has been executed, as well as, for each one of them, their elements (the $eltsHandledSrc$ set) involved in this rule. Furthermore, the associated elements created in the output model are also highlighted and gathered in the $eltsHandledDest$ set. These results are presented in Table I. Each set being composed of only one element, their representations have been omitted in the table in order not to reduce the readability. Each line corresponds to a rule call.

Based on these information, the remainder of the improvement test set process is manually performed with one of the identified model, for example, the *simple_reshape.uml* test model. The *direction* properties of the *p1* and *p2 Flowport* are respectively set to *_in* and *_out*. The same occurs for the model produced by the original transformation from these input models. However, a static analysis of the faulty rule leads to the following observation: the original transformation may produce a *FlowPort* with a direction set to *_inout* but not the studied mutant. Thus, to kill this mutant, the tester must create a new test model whose the result of the transformation by the initial transformation contains at least one *FlowPort* with a direction set to *_inout*. A new test model is built by copying the *simple_reshape.uml* test model and modifying the *direction* property of the *p1:Port* or the *p2:Port* to *_inout*.

In order to check the efficiency of the new test, the mutation analysis process is performed once again. 33 test models are taken into account. The studied mutant is henceforth killed and the mutation analysis process goes on with another live mutant.

The modifications to perform on the test model to create a new one are not so easy than the one of the above example. However, this example illustrates how the information gathered thanks to our algorithm can be used to raise the quality of a test model set.

B. Quantitative Study

This section aims to show that our approach enables the tester to save a considerable amount of time and that the execution time remains largely acceptable whereas 1120 executions are performed and so many results analyzed. For this purpose, we perform different benchmarks corresponding to 5, 15 and 32 test models, respectively.

Identification of the live mutants. The number of mutants remains 35 in the three benchmarks, only the number of test models and thus also the number of cells to explore vary. The identification of the live mutants uses only the mutation matrix model that in our implementation is loaded once. The loading time is relatively short and approximates 1 second. The operations performed to identify the live mutants are only navigations that are quite instantaneous. The observed execution times are lower than 1 second for each benchmark.

Execution of the algorithm 1 for one live mutant. Once again three benchmarks with respectively 5, 15 and 32 test models are performed. The algorithm identifies the models, input elements and output elements impacted by the application of the faulty rule in respectively, 42 seconds, 1 minute 30 seconds and 3 minutes 45 seconds (Table II). These measures depend on the size of the models but reasonably sketches the variation of the time execution with the number of test models. With 32 test models, the observed time is acceptable and remains largely inferior to the time spent to manually collect the information. The execution time mostly

TABLE II
IDENTIFIED MODELS FOR EACH TEST MODELS SET

Test set size	Number of identified models	Execution time
5	1	42”
15	2	1’30”
32	5	3’45”

corresponds to the model loading time since other executions once again only concern quasi instantaneous navigations. This loading time increase with the complexity of the model and the metamodel. It is even more predominant for UML models enhanced with profiles than for classical EMF models. For each benchmark, Table II also indicates the number of test models for which the faulty rule is executed. Naturally, the number of identified models raises with the size of the test model set.

The execution time to collect the information associated to one mutant linearly raises with the number of test models. Our approach has to be tested with hundreds test models and the execution time measured. However, the quantitative analysis is really promising concerning the scalability of our algorithm.

V. RELATED WORK

There are different ways to obtain a qualified test data set. Since model transformation testing has only been briefly studied, few works consider test models qualification and improvement.

Fleurey et al. [14] propose to qualify a set of test models regarding its coverage of the input domain. The input domain is defined with metamodels and constraints. The qualification is static and only based on the input domain whereas the mutation analysis relies on a dynamic analysis of the transformation. In case of very localized transformations, the approach developed by Fleurey et al. produces more models than necessary.

However, in [15], they also propose an adaptation of bacteriologic algorithm to model transformation testing. The bacteriologic algorithm [16] is designed to automatically improve the quality of a test data set. It measures the mutation score of each data to (1) reject useless test data, (2) keep the best test data, (3) “combine” the latter to create new test data. Their adaptation consists in creating new test models by covering part of the input domain still not covered. The authors use the bacteriologic algorithm to select models whereas we propose the mutation analysis associated to trace mechanisms.

In [17], authors study how to use traceability in test driven development (TDD). TDD involves writing the tests prior to the development of the system. Here, traceability can be used to help the creation of new tests considering how the system covers the requirements. The trace links the requirement and the code, and helps the developer to choose the next features which should be tested, then coded. In that approach they do not consider the fault revealing power of the test data set, but the coverage of the requirements to assist the creation of test data.

VI. CONCLUSION

As any other program, it is important to test model transformations. For this purpose, test data set has to be qualified. Mutation analysis is an existing approach that has already been approved and adapted to model transformations. In this paper, we focus on the test model set improvement step and propose a traceability mechanism. This mechanism completely adopts the model paradigm and relies on a local trace metamodel and a matrix metamodel. An algorithm has been developed to assist in the analysis of the execution results for each couple (mutant, test model). This approach has been illustrated on a real case study; the UML to MARTE transformation of the Gaspard framework.

This approach clearly helps to statically analyze the execution results. However, modifying existing models to create a new one, that kills a live mutant, remains manually performed. We are currently designing operator metamodels and consequently adapting the creation of new test models. With these in progress works and the approach presented in this paper, we are going towards a full automation of the mutation analysis process.

REFERENCES

[1] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “Mujava: an automated class mutation system,” *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, 2005.

[2] R. DeMillo, R. Lipton, and F. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, 1978.

[3] J. M. Voas and K. W. Miller, “The revealing power of a test case,” *Softw. Test., Verif. Reliab.*, vol. 2, no. 1, pp. 25–42, 1992.

[4] T. Murmane, K. Reed, T. Assoc, and V. Carlton, “On the effectiveness of mutation analysis as a black box testing technique,” in *Software Engineering Conference*, 2001, pp. 12–20.

[5] J.-M. Mottu, B. Baudry, and Y. Le Traon, “Mutation analysis testing for model transformations,” in *ECMDA 06*, Spain, Jul. 2006.

[6] A. J. Offutt and J. Pan, “Detecting equivalent mutants and the feasible path problem,” *Software Testing, Verification and Reliability*, vol. 7, 1997.

[7] S. Sen, B. Baudry, and J.-M. Mottu, “On combining multi-formalism knowledge to select models for model transformation testing,” in *ICST.*, Norway, Apr. 2008.

[8] “EMFcompare,” www.eclipse.org/emft/projects/compare.

[9] IEEE, *IEEE standard computer dictionary : a compilation of IEEE standard computer glossaries*. New York, NY, USA: IEEE Computer Society Press, 1991.

[10] F. Glitia, A. Etien, and C. Dumoulin, “Traceability for an MDE Approach of Embedded System Conception,” in *ECMDA Traceability Workshop*, Germany, 2008.

[11] V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser, “Traceability mechanism for error localization in model transformation,” in *ICSOFT*, Bulgaria, July 2009.

[12] Object Management Group, “A UML profile for MARTE,” 2007, http://www.omgmarte.org.

[13] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J. Dekeyser, “A model driven design framework for massively parallel embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, 2010, accepted for publication.

[14] F. Fleurey, B. Baudry, P.-A. Muller, and Y. Le Traon, “Towards dependable model transformations: Qualifying input test data,” *SoSyM Journal*, 2007.

[15] F. Fleurey, J. Steel, and B. Baudry, “Validation in model-driven engineering: testing model transformations,” in *MoDeVVa*, 2004.

[16] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon, “From genetic to bacteriological algorithms for mutation-based testing,” *STVR Journal*, vol. 15, no. 2, pp. 73–96, Jun. 2005.

[17] J. H. Hayes, A. Dekhtyar, and D. S. Janzen, “Towards traceable test-driven development,” in *TEFSE Workshop*. USA: IEEE Computer Society, 2009, pp. 26–30.