

# New Ideas: Automated Engineering of Metamorphic Testing Environments for Domain-Specific Languages

Pablo C. Cañizares  
Pablo.Cerro@uam.es  
Universidad Autónoma de Madrid  
Madrid, Spain

Pablo Gómez-Abajo  
Pablo.GomezA@uam.es  
Universidad Autónoma de Madrid  
Madrid, Spain

Alberto Núñez  
Alberto.Nunez@pdi.ucm.es  
Universidad Complutense de Madrid  
Madrid, Spain

Esther Guerra  
Esther.Guerra@uam.es  
Universidad Autónoma de Madrid  
Madrid, Spain

Juan de Lara  
Juan.deLara@uam.es  
Universidad Autónoma de Madrid  
Madrid, Spain

## Abstract

Two crucial aspects for the trustworthy utilization of domain-specific languages (DSLs) are their semantic correctness, and proper testing support for their users. Testing is frequently used to verify correctness, but is often done informally – which may yield unreliable results – and requires substantial effort for creating suitable test cases and oracles.

To alleviate this situation, we propose an automated technique for building metamorphic testing environments for DSLs. Metamorphic testing identifies expected relationships between the outputs of two consecutive tests, reducing the effort in specifying oracles and creating test cases manually. This new ideas paper presents the overarching concepts, the architecture and a prototype implementation. We illustrate our proposal using a DSL to model and simulate data centres.

**CCS Concepts:** • Software and its engineering → Software verification and validation.

**Keywords:** Metamorphic Testing, Model-Driven Engineering, Software Language Engineering, DSLs

## ACM Reference Format:

Pablo C. Cañizares, Pablo Gómez-Abajo, Alberto Núñez, Esther Guerra, and Juan de Lara. 2021. New Ideas: Automated Engineering of Metamorphic Testing Environments for Domain-Specific Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21)*, October 17–18, 2021, Chicago, IL, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3486608.3486904>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SLE '21, October 17–18, 2021, Chicago, IL, USA

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-9111-5/21/10...\$15.00

<https://doi.org/10.1145/3486608.3486904>

## 1 Introduction

Model-driven engineering (MDE) promotes models as the main artefacts of the software development process [4]. Models can be created with general-purpose modelling languages, like UML [30], or using dedicated domain-specific languages (DSLs) tailored for a domain [21, 37]. Modelling languages encompass abstract syntax, concrete syntax and semantics [16]. The latter is typically defined through code generators, simulators, or a combination of both [11].

Ensuring the correctness of DSLs – and the programs developed with them – is crucial for the success of MDE solutions. Testing can be used for this purpose. At the language level, there are proposals for test-driven DSL development processes [20] and automating the testing of DSL implementations [31]. At the DSL usage level, there are proposals for (unit) testing [22, 40] and debugging [3, 27]. However, specifying test oracles for some DSLs is challenging, and creating test cases manually requires a high effort.

Metamorphic testing (MeT) [9, 34] is a technique for testing systems in cases where there is no oracle or it is too expensive to compute [38]. This kind of testing identifies relations that describe expected variations in the output of two subsequent test cases, when the input of the first test case is changed according to some criteria. MeT has been successfully applied to difficult-to-test systems in many disciplines, solving the oracle issue and facilitating the automated generation of test cases [10, 32].

We claim that using MeT for both engineering and using DSLs may help to increase the trustworthiness of MDE solutions. However, building one MeT environment by hand, for each employed DSL, is costly. This is so as it involves developing tools for defining and executing MeT relations based on features and processes that depend on each DSL. Hence, in this new ideas paper, we propose to automate the construction of MeT environments, based on the use of a DSL for their specification. Our proposal is applicable both at the DSL definition level (e.g., to test the DSL semantics) and at the DSL usage level (e.g., to test DSL programs). We illustrate its use on a DSL for data centres.

## 2 Background and Running Example

This section provides background on MeT (Sec. 2.1) and describes a DSL that we use as a running example (Sec. 2.2).

### 2.1 Metamorphic testing (MeT)

Testing complex systems entails two main challenges. The first one is the *oracle problem*, which refers to the availability of mechanisms to assess if a test case passes or fails. However, some systems – like the cloud [28], scientific computation software [25], or machine learning applications [41] – may not have an oracle available, or it can be computationally too expensive to apply. Additionally, since the amount of potential test cases for a complex system may be computationally unaffordable, it is desirable to select just a subset of them that determines the system correctness effectively. Unfortunately, selecting an optimal subset is challenging in most cases. This is known as the *reliable test set problem*.

MeT is a testing technique that aims at alleviating these two problems. It has been used in very different domains, such as web services [33], machine learning [41], compilers [24] and cloud systems [8, 28]. MeT uses metamorphic relations (MRs) to determine if the execution of the test cases is correct. In contrast to conventional testing, where the result provided by each individual test case is compared to the one provided by the oracle, MeT studies the relations between different test inputs and the resulting outputs.

MRs model the behaviour of the system under test. An MR can be seen as a property of the system that involves multiple inputs and their outputs. It can be represented as a logical implication  $MR_i \Rightarrow MR_o$ , where  $MR_i$  is a relation between the inputs of two test cases, and  $MR_o$  a relation between their outputs. Hence, if the relation between the inputs is satisfied, so must be the relation between the outputs.

The *reliable test set problem* is faced by generating so called *follow-up* test cases using the MRs and a small set of initial test cases, generally created by the tester. A follow-up test case  $t'$  can be generated from a test case  $t$  and an MR by performing a modification on  $t$ 's input so that  $MR_i$  is satisfied. Then, the MeT process checks whether  $MR_o$  is also satisfied, for  $t$  and each synthesized follow-up test case. This way, by using the MRs, the *oracle problem* is alleviated as well.

### 2.2 Running example: MeT for data centres

To illustrate our MeT proposal, we will use a DSL to describe data centres. Our goal is to simulate the data centre models against workloads (i.e., applications to be executed) to obtain expected processing times. Fig. 1 shows a highly simplified meta-model for the DSL, where a DataCentre is made of a Network and any number of Racks. Each Rack contains Boards, which are connected via Switches and have computing nodes with characteristics described by NodeTypes.

To test if a simulator  $S$  for this DSL performs as expected, it is difficult to establish an oracle, but we can use MeT instead.

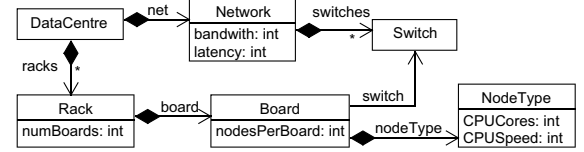


Figure 1. Meta-model excerpt of a DSL for data centres.

Hence, we can define a test case as a pair  $(m_1, \omega)$ , where  $m_1$  is a data centre model,  $\omega$  is a workload (also a model, whose meta-model is omitted due to space constraints), and  $S_t(m_1, \omega)$  is the simulation time of the data centre  $m_1$  when processing the workload  $\omega$  using simulator  $S$ .

Then, we can exploit expert knowledge about data centres to define MRs. For example, as a rule of thumb, decreasing the number of computing nodes (leaving the other components the same) may increase the processing time of a workload. This can be expressed as the MR:  $NNodes(m_1) > NNodes(m_2) \Rightarrow S_t(m_1, \omega) \leq S_t(m_2, \omega)$ , where  $NNodes$  is a function counting the number of nodes of a data centre. The function  $NNodes$  in the MR pre-condition is to be evaluated on the test cases, while  $S_t$  in the post-condition is to be evaluated on the simulation results. We call the functions over test cases (like  $NNodes$ ) input features, while those on the outputs (like  $S_t$ ) are called output features.

Overall, the MR can be used as an oracle (to check that decreasing nodes increases the simulation time) and to generate follow-up test cases (e.g., by reducing the nodes of an initial test model  $m_1$  decreasing the values of nodesPerBoard). Instead of one, a thorough MeT process would have a catalogue of MRs, modelling different aspects of the DSL semantics (e.g., network, bandwidth, memory, energy).

## 3 Overview of the Approach

We propose a method to synthesize MeT environments for DSLs, which can be used to test the DSL definition, or to provide testing support to DSL users. Fig. 2 overviews the approach, assuming the former scenario. It involves three phases and three roles (MeT expert, DSL semantics expert and tester) who may (or may not) be the same person.

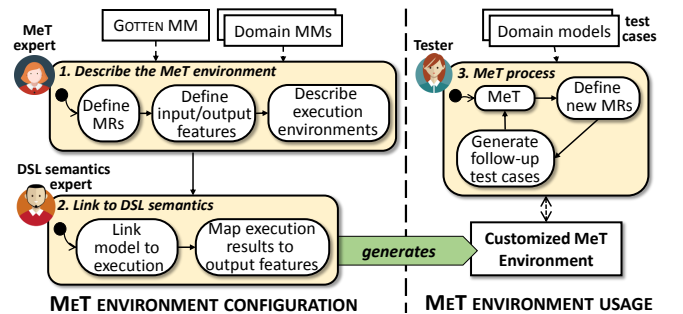


Figure 2. Overview of our approach.

In the first phase, the MeT expert describes the MeT environment. For this purpose, we provide a DSL called GOTTEN (for Generic MDE framework fOr meTamorphic TEStiNg), which facilitates the definition of MRs, input features that the MRs can use (e.g.,  $NNodes$ ), output features (e.g.,  $S_t$ ), and meta-data of the DSL execution or simulation environments. The value of the input features is extracted from the test models using OCL [29]. Sec. 4 will detail this DSL.

In the second phase, the expert in the DSL semantics defines how to invoke the DSL execution, and how to extract the value of the output features used by the MRs from the execution results. For example, for the data centre DSL, the expert would map the DSL models into the input of the CloudSim simulator [7], and would specify how to extract the simulation time from the simulator results. Sec. 5 will explain this phase.

Next, our approach generates the MeT environment out of the previous specifications. In this environment, the tester can provide manually created test models, and the environment performs the following tasks:

- *MeT*. Every pair of test models is checked against the pre-condition of each MR. If a pair of models satisfies the pre-condition of an MR, then, the post-condition of the MR is evaluated to determine the success or failure of the test case. At the end of the process, the environment reports the set of passed/failed test cases.
- *Define new MRs*. To complement the MRs defined in phase 1, the tester can define new ones by means of GOTTEN.
- *Generate follow-up test cases*. Since the MRs use input features specified in OCL, it is possible to use a model finder to obtain follow-up test cases of a given test case, which together satisfy the pre-condition of an MR. A model finder is a constraint solver over models. It receives a meta-model and a set of OCL constraints, and outputs a model that conforms to the meta-model and satisfies the constraints [18, 23]. Our current prototype does not support this feature yet, but it is on our future work agenda.

As mentioned above, the MeT environment can be used to test both the DSL semantics (as in our example, to test a simulator for data centres) and the utilization of the DSL by its users (e.g., a MeT environment to test ATL model transformations [36], to be used by transformation developers).

## 4 The GOTTEN DSL

GOTTEN supports the configuration of the MeT environment for a given DSL. Fig. 3 shows an excerpt of its meta-model. A GOTTEN program (represented by class `GottenEnvironment`) comprises five parts:

- A declaration of the (Ecore) meta-models of the target DSLs (class `Domain`), and the folder containing the input test models – instances of the given meta-models – for the MeT process (class `TestModelsRepository`).

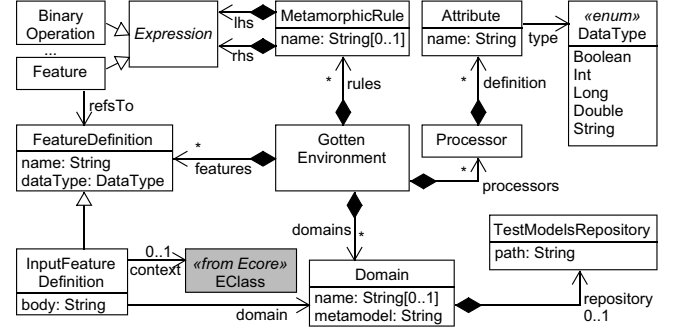


Figure 3. Excerpt of the GOTTEN DSL meta-model.

- A definition of the input and output features that the MRs can use (reference features). Output features (class `FeatureDefinition`) have a name and a type (integer, long, double, boolean, string). Input features (class `InputFeatureDefinition`), in addition, declare an OCL expression to calculate their value from the input test models, and may be defined in the context of a class (an `EClass`, since we assume Ecore domain meta-models).
- Meta-data (classes `Processor` and `Attribute`) to be informed by each DSL execution environment, or processor, that the MeT process will use.
- The declaration of MRs (class `MetamorphicRule`).

Listing 1 shows a simple GOTTEN program. Lines 1–2 declare the datacentre meta-model and the folder containing the test models. Here, one could declare additional meta-models, e.g., for the workload. Next, lines 4–7 define the input feature `NNodes` of this meta-model, its type `Int` and the OCL expression to compute its value. The context of an input feature may be empty, meaning that the OCL expression has a global scope. Otherwise, the expression is evaluated on any object of the given type. Since `DataCentre` is the root class of the datacentre meta-model, we can assume that datacentre models only have one object of this type.

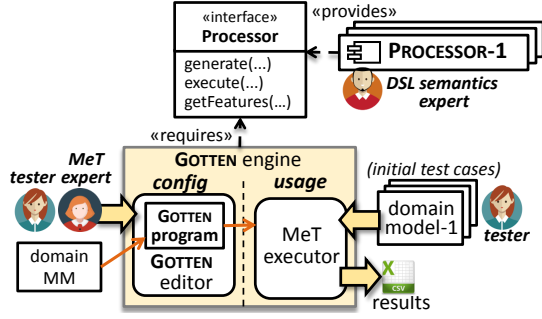
```

1 metamodel datacentre "/sample.gotten/model/datac.ecore"
2 models "/sample.gotten/model/dcmmodels"
3
4 datacentre input Features {
5   context DataCentre def : NNodes: Int =
6     "racks→collect(numBoards*board.nodesPerBoard)→sum()"
7 }
8 output Features {
9   Time: Long
10 }
11 Processor {
12   Name: String
13   Version: String
14 }
15 MetamorphicRelations {
16   MR1 = [ (NNodes(m1) > NNodes(m2)) => (Time(m1) <= Time(m2)) ]
17 }

```

Listing 1. GOTTEN program for the running example.

Lines 8–9 declare the output feature `Time` with type `Long`. The values of output features need to be retrieved from the



**Figure 4.** Architecture of the GOTTEN engine.

execution output. To specify how to retrieve these values, we provide an extension point, which is explained in Sec. 5.1. Lines 11–14 detail the processor meta-data, in this case, Name and Version. At run-time, the GOTTEN engine supports instantiating multiple processors to enable the comparison of alternative execution semantics of a DSL. In our running example, this is useful to compare the accuracy of different cloud simulators w.r.t. the aspects modelled by each MR. Finally, line 16 defines the MR explained in Sec. 2.2, which uses the previously defined input and output features.

## 5 Architecture and Tool Support

In this section, we overview the architecture of our solution (Sec. 5.1) and describe our prototype tool support (Sec. 5.2).

### 5.1 Architecture

Fig. 4 shows the architecture of the GOTTEN engine, which is designed as an Eclipse plugin. It provides an editor where the MeT experts can specify the MRs and their features using the GOTTEN DSL. In order to register execution and simulation engines for the DSL, and map their results to output features of the MRs, the DSL semantics expert needs to instantiate an extension point. This requires implementing a Java interface called Processor for each considered execution/simulation engine. The interface demands the implementation of three methods: generate, to transform the DSL models into the input format of the processor; execute, to run the processor over the given input test models; and getFeatures, to extract the value of the output features from the execution results.

At this point, the MeT environment for the DSL is ready, and the tester can use it by providing the set of domain test models, and optionally adding MRs. Then, the *MeT executor* takes care of executing the MeT process, displaying the results in an interactive view and storing them as a CSV file.

### 5.2 Tool support

The GOTTEN environment is an Eclipse plugin<sup>1</sup>, developed using EMF [35] for handling the (meta-)models, and Xtext [42] for creating the editor. Fig. 5 shows it in action.

<sup>1</sup><https://g0tten.github.io/gotten/>

The environment provides an editor for GOTTEN (label 1), featuring code completion, syntax highlighting, and validation of the features within each MR. GOTTEN programs are stored within GOTTEN projects (label 2), which typically also host the meta-models and the test models. After defining a GOTTEN program, it is possible to invoke a wizard to generate a plugin project template instantiating the Processor extension point. Label 3 shows a project example that implements the methods of the Processor interface (cf. Fig. 4) for the execution of datacentre models using CloudSim [7].

The environment has several views for the MeT process. The model view (label 4) classifies the test models by each active processor. This view allows to double-click on each model to launch the generate method of the instantiated Processor interface. The processors view (label 5) enables executing the MeT process for the selected processors by double-clicking on each of them, and shows the values of the output features. A wizard (not shown) permits launching the MeT process and selecting the MRs to apply. The results view (label 6) displays the evaluation of the MRs for each pair of models, distinguishing between those whose pre-condition ( $MR_i$ ) is not satisfied (displayed in brown), those whose pre-condition and post-condition ( $MR_o$ ) are satisfied (i.e., passing tests, in green), and those whose pre-condition holds but not the post-condition (i.e., failed tests, in red). In addition, this view supports filtering the results by each one of these three kinds. Just for illustration, the figure shows the MR MR2 with the reverse post-condition as MR1, which is reflected in reversed pass/fail tests in the results view.

## 6 Related Work

Next, we review works on MeT frameworks, on the use of MeT to verify MDE artefacts, on testing DSL specifications, and providing test facilities to their users.

*MeT frameworks.* Existing MeT frameworks have been built ad-hoc for specific domains. Sun et al. [5, 6] proposed a MeT framework for testing web services. Hadiwijaya and Liem [15] proposed a language for defining MRs and generating test cases for program competition environments in Java. DeepRoad [43] is a framework for testing DNN-based autonomous driving systems, able to synthesize realistic driving scenes for detecting inconsistent driving behaviours. AMT [14] is a MeT framework for checking MRs, built atop Inka for generating test cases by means of constraint programming. The scope of AMT is limited to programs supported by Inka (a limited subset of C and C++).

These works evince the interest of using MeT for hard-to-test programs, but they are limited to a field of application. Only [6, 15] provide dedicated languages to define MRs, but they are heavily oriented to the application domain, handling simple data types and targeting a specific programming language. Hence, none of them provide mechanisms to design MRs independently of the domain; execute different systems



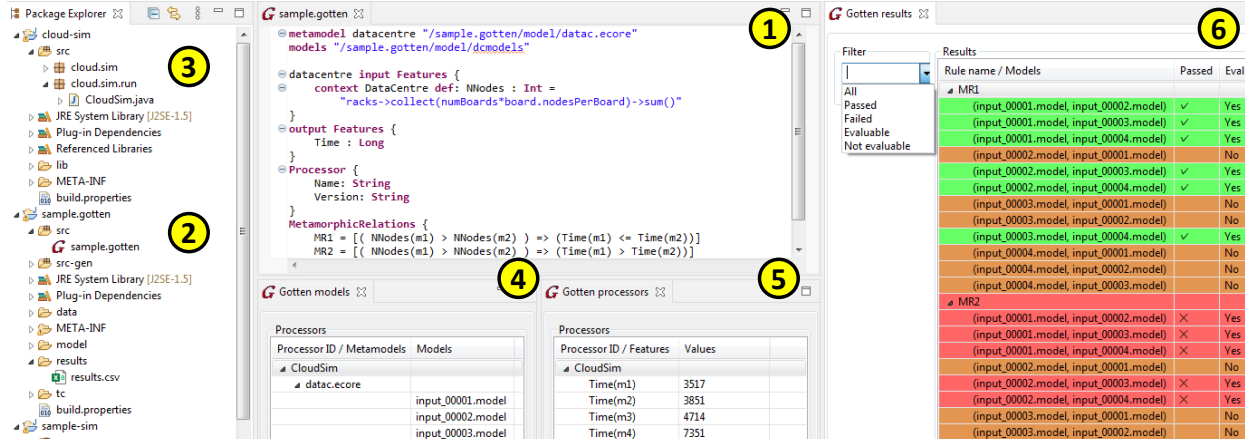


Figure 5. GOTTEN development environment in action.

and applications; and evaluate the MRs on the results. For this purpose, our DSL GOTTEN permits defining input/output features for their use in MRs; and our solution defines extension points to link with the execution environments.

*Use of MeT within MDE.* Some researchers have proposed using MeT to verify transformations and code generators. Boussaa et al. [1] detect inconsistencies in code generators using non-functional MRs for resource usage and performance. MeT has also been applied to test model transformations [12, 17, 19]. Troya et al. [36] propose an approach for the automatic inference of MRs in ATL transformations based on the execution traces. They define a catalogue of 24 domain-independent MRs, which are used as a basis for instantiating new MRs for transformation programs.

Similar to the previous case, these MeT approaches are specific to one domain. A framework like ours could facilitate building MeT environments for all sorts of MDE artefacts.

*Systematic testing of DSLs.* There are proposals to automate the testing of DSLs. Kats et al. [20] define a language parametric testing language within Spoofax, which can be used to test different aspects of a DSL definition and implementation, including its syntax and semantic services. Ratiu et al. [31] present some features of the MPS workbench to help systematic testing of DSLs, which they employed to develop *mbedder*, a DSL for embedded systems. The testing helpers include a language to test the type system, a model checker, and generators of input test programs. Interestingly, one of their biggest challenges was finding good oracles. Our MeT framework is directed to help in this aspect.

*Testing facilities for DSLs.* Wu et al. [40] present a testing framework that can be customized for specific DSLs by a mapping into JUnit. Khorram et al. [22] leverage on the Test Description Language [26] to create test cases for executable DSLs [2]. WODEL-test [13] can generate mutation testing environments for DSLs. It provides a DSL to specify mutation operators, and extension points to connect the test engine

with the DSL execution engine. Finally, several approaches support debugging for DSLs [3, 27, 39].

Overall, our approach is useful for both DSL designers and users. In our running example, we tested DSL semantics, but we could use our approach to, e.g., generate a MeT framework for ATL model transformations (useful for the users of ATL). GOTTEN is complementary to the works in this category, providing an additional testing technique that facilitates the creation of oracles and test cases.

## 7 Conclusions and Future Work

In this new ideas paper, we have presented the initial steps towards a model-driven solution to automate the construction of MeT environments for DSLs. We have illustrated its feasibility using a DSL for data centres. Our proposal aims at facilitating a more systematic verification of the different artefacts of MDE solutions.

In the future, our first objective is to improve the expression language for MRs, to make the specification of recurring patterns easier. For example, with a predefined predicate to specify the parts of the models that can differ. This would be useful in our running example to specify in MR1's precondition that, in addition to the condition on *NNodes*, everything else should be equal in *m1* and *m2*, except for the attribute *nodesPerBoard* in *Board* objects. We also plan to work on the generation of follow-up test cases using model finding, as well as to improve the presentation of results of the MeT process. Since our approach permits working with several processors (e.g., several cloud simulators) we will add facilities for comparing them w.r.t. the MeT results. Finally, we would like to apply our framework to different domains.

## Acknowledgements

Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00), the Madrid region (P2018/TCS-4314), and the Madrid-UCM young doctors' program (PR65/19-22452).

## References

- [1] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry. 2020. Leveraging metamorphic testing to automatically detect inconsistencies in code generator families. *Softw. Test. Verification Reliab.* 30, 1 (2020).
- [2] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. DeAntoni, and B. Combemale. 2016. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 84–89.
- [3] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry. 2018. Omniscient debugging for executable DSLs. *J. Syst. Softw.* 137 (2018), 261–288.
- [4] M. Brambilla, J. Cabot, and M. Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers.
- [5] C.-ai Sun, G. Wang, B. Mu, H. Liu, Z.S. Wang, and T. Y. Chen. 2011. Metamorphic testing for web services: Framework and a case study. In *ICWS*. IEEE, 283–290.
- [6] C.-ai Sun, G. Wang, Q. Wen, D. Towey, and T. Y. Chen. 2016. MT4WS: An automated metamorphic testing system for web services. *International Journal of High Performance Computing and Networking* 9, 1/2 (2016), 104–115.
- [7] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. 2011. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software - Practice and Experience* 41, 1 (2011), 23–50.
- [8] P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. 2020. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *J. Syst. Softw.* 163 (2020), 110522.
- [9] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report. HKUST-CS98-01.
- [10] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27.
- [11] B. Combemale, R. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek. 2017. *Engineering modeling languages. Turning domain knowledge into tools*. Chapman and Hall/CRC.
- [12] K. Du, M. Jiang, Z. Ding, H. Huang, and T. Shu. 2019. Metamorphic testing in fault localization of model transformations. In *SOFL+MSVL*. Springer, 299–314.
- [13] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. 2021. Wodel-Test: A model-based framework for language independent mutation testing. *Software and Systems Modelling* 20 (2021), 767–793.
- [14] A. Gotlieb and B. Botella. 2003. Automated metamorphic testing. In *COMPAC*. IEEE, 34–40.
- [15] R. I. Hadiwijaya and M. M. Liem. 2015. Metamorphic testing and DSL for test cases & checker generators. *Olympiads in Informatics* 9 (2015), 75–88.
- [16] D. Harel and B. Rumpe. 2000. *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff*. Technical Report. ISR.
- [17] X. He, X. Chen, S. Cai, Y. Zhang, and G. Huang. 2018. Testing bidirectional model transformation using metamorphic testing. *Information and Software Technology* 104 (2018), 109–129.
- [18] D. Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, London, England. See also <http://alloy.mit.edu/>.
- [19] M. Jiang, T. Y. Chen, F.-C. Kuo, Z. Zhou, and Z. Ding. 2014. Testing model transformation programs using metamorphic testing. In *SEKE*. Knowledge Systems Institute Graduate School, 94–99.
- [20] L. C. L. Kats, R. Vermaas, and E. Visser. 2011. Integrated language definition testing: enabling test-driven language development. In *OOPSLA*. ACM, 139–154.
- [21] S. Kelly and J.-P. Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley.
- [22] F. Khorram, E. Bousse, J.-M. Mottu, and G. Sunyé. 2021. Adapting TDL to Provide Testing Support for Executable DSLs. *Journal of Object Technology* 20, 3 (June 2021), 6:1–15. The 17th European Conference on Modelling Foundations and Applications (ECMFA 2021).
- [23] M. Kuhlmann and M. Gogolla. 2012. From UML and OCL to relational logic and back. In *MoDELS (LNCS)*, Vol. 7590. Springer, 415–431.
- [24] V. Le, M. Afshari, and Z. Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. ACM, 216–226.
- [25] X. Lin, M. Simon, and N. Niu. 2020. Exploratory metamorphic testing for scientific software. *Computing in Science Engineering* 22, 2 (2020), 78–87.
- [26] Philip Makedonski, Gusztáv Adamis, Martti Käärik, Finn Kristoffersen, Michele Carignani, Andreas Ulrich, and Jens Grabowski. 2019. Test descriptions with ETSI TDL. *Softw. Qual. J.* 27, 2 (2019), 885–917.
- [27] S. Van Mierlo, Y. Van Tendeloo, and H. Vangheluwe. 2018. A generalized stepping semantics for model debugging. In *MoDELS Workshops (CEUR Workshop Proceedings)*, Vol. 2245. CEUR-WS.org, 541–546.
- [28] A. Núñez, P. C. Cañizares, M. Núñez, and R. M. Hierons. 2021. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability* 70, 1 (2021), 261–284.
- [29] Object Management Group. 2014. UML 2.4 OCL Specification. <http://www.omg.org/spec/OCL/>.
- [30] Object Management Group. 2017. UML 2.5.1 Specification. <https://www.omg.org/spec/UML/About-UML/>.
- [31] D. Ratiu, M. Voelter, and D. Pavletic. 2018. Automated testing of DSL implementations - experiences from building mbeddr. *Softw. Qual. J.* 26, 4 (2018), 1483–1518.
- [32] S. Segura, G. Fraser, A. B. Sánchez, and A. Ruiz Cortés. 2016. A survey on metamorphic testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824.
- [33] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. *IEEE Trans. Software Eng.* 44, 11 (2018), 1083–1099.
- [34] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen. 2020. Metamorphic testing: Testing the untestable. *IEEE Softw.* 37, 3 (2020), 46–53.
- [35] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional.
- [36] J. Troya, S. Segura, and A. Ruiz Cortés. 2018. Automated inference of likely metamorphic relations for model transformations. *J. Syst. Softw.* 136 (2018), 188–208.
- [37] M. Voelter. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org. <http://www.dslbook.org>
- [38] E. J. Weyuker. 1982. On testing non-testable programs. *Comput. J.* 25, 4 (1982), 465–470.
- [39] H. Wu, J. Gray, and M. Mernik. 2008. Grammar-driven generation of domain-specific language debuggers. *Softw. Pract. Exp.* 38, 10 (2008), 1073–1103.
- [40] H. Wu, J. G. Gray, and M. Mernik. 2009. Unit testing for domain-specific languages. In *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL (LNCS)*, Vol. 5658. Springer, 125–147.
- [41] X. Xie, J. W. K. Ho, C. Murphy, G. E. Kaiser, B. Xu, and T. Y. Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *J. Syst. Softw.* 84, 4 (2011), 544–558.
- [42] Xtext 2021. Xtext. <http://www.eclipse.org/Xtext/>. (last accessed in July 2021).
- [43] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. 2018. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*. ACM, 132–142.