Repositorio Institucional de la Universidad Autónoma de Madrid

https://repositorio.uam.es

Esta es la versión de autor del artículo publicado en:
This is an author produced version of a paper published in:


DOI:  http://doi.org/10.1016/j.scico.2018.01.008

Copyright: © 2018 Elsevier B.V.

El acceso a la versión del editor puede requerir la suscripción del recurso
Access to the published version may require subscription
A Tool for Domain-Independent Model Mutation

Pablo Gómez-Abajo, Esther Guerra, Juan de Lara
Modelling & Software Engineering Research Group
http://miso.es
Computer Science Department
Universidad Autónoma de Madrid (Spain)

Mercedes G. Merayo
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid (Spain)

Abstract

Mutation is a systematic technique to create variants of a seed artefact by means of mutation operators. It has many applications in computer science, like software testing, automatic exercise generation and design space exploration. Typically, mutation frameworks are developed ad-hoc by implementing mutation operators and their application strategies from scratch, using general-purpose programming languages. However, this is costly and error-prone.

To improve this situation, we propose Wodel: a domain-specific language and tool for model-based mutation that is independent of the domain meta-model. Wodel enables the rapid development and application of model mutations. It provides built-in advanced functionalities like automatic generation of seed models, and static and dynamic metrics of operator coverage and applicability. It offers extension points, e.g., to post-process mutants and describe domain-specific equivalence criteria. As an example, we illustrate the usage of Wodel for the mutation of security policies, and present an empirical evaluation of its expressiveness.

Keywords: Model-driven engineering, domain-specific languages, model mutation, mutation footprint, model synthesis

1. Introduction

This paper presents Wodel, an extensible software tool for model mutation that consists of: (1) an editor to define mutation operators and their application policies, featuring code completion and code validation; (2) a compiler of
Wodel programs into executable Java code implementing the defined operators; (3) metrics for mutation footprints which provide information about the static coverage of a meta-model by a mutation program, as well as about the effects of the dynamic execution of a mutation program on a given set of models; (4) a seed model synthesizer to automatically generate seed models ensuring the application of the defined operators; and (5) an extensibility mechanism that allows pipelining external applications to Wodel programs.

The rest of this paper is organized as follows. First, Section 2 identifies some limitations of existing mutation frameworks to define mutation-based applications. Then, Section 3 describes the architecture and main functionalities of Wodel, and Section 4 gives details of its implementation. Section 5 illustrates Wodel using an example in the domain of security policies \[1\]. Finally, Section 6 evaluates the expressiveness of our proposal, and Section 7 ends with the conclusions and lines of future work.

2. Motivation and background

Model mutation is the process of generating variants (i.e., mutants) of a seed model by the application of a set of mutation operators. Mutation is essential to applications like mutation testing \[2, 3\], where a program is modified and then used to assess the quality of a test suite; automated generation of modelling exercises \[4\], where variations of a correct solution model are presented to students, who must identify the injected errors; model-driven design space exploration \[5, 6\], where the goal is to heuristically find a model optimizing some property, and the model population is generated by mutation; and synthesis of a diversity of test models which are generated by mutating some given seeds \[7\].

Although there are some frameworks for model mutation, their scope is limited to either a particular language (e.g., logic formulae \[8\]) or application domain (e.g., testing \[9, 10\]). Moreover, mutation operators must be defined using general-purpose programming languages not tailored to the definition and creation of mutants. Hence, developing domain-specific applications that rely on mutation (like the abovementioned ones) becomes costly and error-prone, because existing tools do not facilitate the creation and analysis of mutation operators for arbitrary languages and application domains.

In order to fill this gap, we propose a model-based mutation approach where the artefact to be mutated is represented as a model conformant to a domain meta-model, and the mutation operators are defined using a domain-specific language called Wodel. Wodel provides high-level primitives to simplify the definition of mutation operators and their application strategies, and it supplies some built-in services to facilitate the generation of mutants, like a registry of applied mutations, the ability to detect duplicated or malformed mutants, debugging support, and synthesis of seed models that exercise the operators. The framework provides handy integration with external applications through a compilation into a general-purpose programming language. Moreover, it is extensible with post-processing actions that can use the generated mutant models, like mutation testing or exercise generation \[4\].
3. Software framework

Figure 1 shows the modular, component-based architecture of our mutation framework. The typical workflow is as follows. First, since our approach is domain-independent, the user needs to describe the domain concepts by means of a domain meta-model (label 1 in the figure). For example, in order to mutate automata, the user should provide a meta-model including the concepts of state, transition and alphabet symbol, as well as how they relate to each other.

Next, the user defines the desired mutation operators and their execution details, like how many times each mutation operator should be applied, or the mutation execution order. We call this specification a $\textit{WODEL}$ program (label 2). $\textit{WODEL}$ is meta-model independent, which means that it can be used to define mutation operators for arbitrary meta-models. Nonetheless, $\textit{WODEL}$ programs must refer to a domain meta-model to allow type checking – to assure the program only refers to valid meta-model types and features – and to ensure the resulting mutants are valid.

Once created, a $\textit{WODEL}$ program can be applied to seed models conforming to the given domain meta-model (label 3). For convenience (e.g., to exercise the defined operators and assess their correct implementation), we provide a seed models synthesizer that is able to generate seed models to which all mutation operators in the program are applicable.

The execution of a $\textit{WODEL}$ program produces mutant models from the seed
models (label 4), as well as a mutation registry keeping trace of the modified elements and applied mutation operators (label 5). In addition, the tool offers two extension points (label 6) to further customize Wodel for a domain or application. The first extension point permits specifying domain-specific equivalence criteria to avoid the generation of duplicate mutants. The second extension point allows registering application-specific post-processors for the generated mutants, e.g., for mutation-based testing or automated exercise generation [4].

Next, we describe the functionalities that Wodel offers to facilitate the definition and generation of mutants:

- Wodel provides nine mutation primitives for object creation and deletion, reference redirection, attribute modification, object retyping, and shallow and deep object cloning, among others. The primitives can define a range \( m..M \) to indicate they must be applied a random number of times between \( m \) and \( M \), and can be combined into composite mutations with transactional semantics. The mutation candidate objects can be selected according to different strategies, like randomly, based on some property value, or to all objects satisfying certain property.

- The engine verifies that each generated mutant is a valid model (i.e., it conforms to the domain meta-model and satisfies its integrity constraints). Non-conformant mutants are discarded, in which case, the engine attempts to generate another mutant up to a configurable maximum number of retries. Wodel programs can include OCL invariants [11] that any produced mutant is enforced to satisfy.

- When generating the mutants, the engine takes care of assigning an appropriate value to any mandatory attribute or reference not initialized by the Wodel program. Similarly, new objects are automatically placed in a suitable container, if no explicit container is stated. This way, programs become more compact, and the likelihood to obtain valid mutants is higher.

- Wodel includes a mechanism to identify and avoid duplicate mutants. Ensuring uniqueness of mutants is useful in some applications, like the automated generation of exercises [4] or mutation testing. By default, mutant equivalence is syntactic, but users can provide their own equivalence criteria (e.g., behavioural) through an extension point.

- The execution of a Wodel program produces a registry of applied mutations and objects affected by them (label 5 in Figure [1]). This registry can be used to replicate the mutation process, or for application-specific purposes. For example, we have used it to synthesize a natural language description of the mutations, in order to include it in automatically generated exercises [4]. The registry is stored as a model, and can be compacted to eliminate mutations that cancel each other (e.g., a mutation that creates an object, and another that deletes it).
• An optional post-processing step can be used to generate domain-specific artefacts tailored to particular applications (label 6 in Figure 1), like mutation testing or exercise generation [4].

• **Wodel** provides some metrics that help in analysing the behaviour of mutation programs. On the one hand, the static footprint of a **Wodel** program identifies the meta-model classes and features touched by the program, and the kind of changes it performs (creation, deletion or modification). This information is computed statically, and is useful to identify immutable types or undesired mutation side effects. On the other hand, **Wodel** programs are stochastic, as each mutation operator can be configured to be applied a random number of times at random locations. To ascertain the actual operators applied to build a mutant, the dynamic footprint of the program execution can be inspected. There are two types of dynamic footprints: net (i.e., net effect of the program execution calculated by differencing the seed model and the mutant) and debugging (i.e., detailed enumeration of applied operators). This information can help to locate program errors by identifying parts of the seed model that were not mutated as expected. The footprints are available via dedicated Eclipse views with drill-down tables showing the information organized either by meta-model element or by mutation operator, and where cells use different colours to easily distinguish between creation, modification and deletion actions.

• To facilitate the evaluation of **Wodel** programs and exercise their operators, the IDE permits the automatic synthesis of seed models ensuring that all instructions in the program will be applicable to the models (if any such models exist within a given search bound).

### 4. Implementation

**Wodel** is available as an Eclipse plugin. It includes an Xtext editor for **Wodel** programs which features syntax highlighting, automatic code completion, and type-checking of programs against the specified domain meta-model to ensure only valid meta-model types and features are used. The underlying modelling technology is the Eclipse Modelling Framework (EMF) [12], the de-facto standard for modelling within Eclipse nowadays.

**Wodel** programs are compiled into Java using an Xtend code generator. The produced Java code, which is in charge of creating the mutants from the seed models, can be transparently executed from the IDE, or in a separate standalone application. Being able to execute **Wodel** programs outside the IDE may be needed by some applications, and is the reason why we opted for a compiled approach.

The seed model synthesizer relies on model finding, a technique based on
constraint solving over models [13]. In particular, the synthesizer produces a
description of the domain meta-model and its OCL integrity constraints, en-
riched with additional OCL invariants derived from the Wodel program. The
invariants express the requirements that a seed model must fulfil to enable the
application of each mutation operator in the program. For example, the Wodel
instruction remove one ElementType requires the seed model to contain an instance
of ElementType to ensure the instruction is applicable, which is encoded as the
OCL invariant ElementType.allInstances().size() > 0. Then, the enriched meta-model
is fed into the USE Validator [14] model finder, which searches for models that
are valid instances of the domain meta-model and satisfy the invariants. Users
can customize the search by providing a search scope (minimum and maximum
number of objects in the seed models), additional model requirements expressed
with OCL, or a seed EMF model for the search.

Figure 2 is a screenshot of Wodel which shows its editor (label 1), the Java
code generated from a Wodel program (label 2), a domain meta-model and some
seed models (label 3), several mutants generated from the seed models (label 4),
and the static and dynamic footprints of the program (label 5). The different
artefacts in the screenshot correspond to an example in the domain of security
policies, which we will develop in the next section.

As the figure shows, the static footprint view counts the number of explicit
and implicit creations (C, IC), modifications (M, IM) and deletions (D, ID) of each
meta-model class and feature (upper-left view), or performed by each mutation
operator (lower-left view). The cells corresponding to classes aggregate the
actions performed on the class and its features. For instance, the cell for the
explicit creation of class Rule contains 1c 2f because the program contains one
explicit creation of Rule objects and two explicit creations of its features. The
first row of the tables displays the average class coverage for each type of ac-
tion. For example, the explicit creation percentage is 14% because the program
explicitly creates 1 out of the 7 classes in the domain meta-model.

The dynamic footprints to the right (net and debug) have columns stating
the number of elements actually created (C), modified (M) and deleted (D) by
the execution of the Wodel program over two seed models (LibraryOrBAC and
LibraryRBAC), and the effects of each mutation operator on the models.

5. Example

Next, we illustrate Wodel with an example in the context of mutation testing
for security policies. Here, the goal is measuring the quality of a set of test
cases by injecting errors in the artefact under test (a security policy), and then
checking whether the test cases detect the injected errors. We have chosen this
application scenario because it is concise, but still, it will allow demonstrating
all features of our tool.

We base the example in the work of Mouelhi and collaborators [11]. They pro-
pose a meta-model to represent both access control languages – like RBAC\(^3\) and OrBAC\(^4\) – and security policies expressed with them. Hence, this meta-model contains classes like RuleType or PolicyType to represent access control languages, and classes like Rule or Policy to specify security policies. In addition, the authors define five mutation operators for security policies expressed with this meta-model, so that the operators are independent of the concrete access control language (i.e., RBAC or OrBAC). Table 1 shows their proposed generic mutation operators.

We can use Wodel to define the operators in this table. As an example, Listing 1 shows the Wodel program that implements the PPD mutation operator. This mutation replaces one rule parameter with one of the parameter descendants. Line 1 states that mutants are to be generated in folder out, from the seed models in folder models. The number of mutants generated from each

\(^3\)Role-Based Access Control
\(^4\)Organization-Based Access Control
Table 1: Mutation operators for security policies (from [1]).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTT:</td>
<td>Selects two rule types with the same parameter types. Then modifies the type of a rule whose type is the first selected rule type with the second rule type.</td>
</tr>
<tr>
<td>PPR:</td>
<td>Replaces a rule parameter with a different one (of same type).</td>
</tr>
<tr>
<td>ANR:</td>
<td>Creates a new rule using a selected rule type.</td>
</tr>
<tr>
<td>RER:</td>
<td>Removes a rule.</td>
</tr>
<tr>
<td>PPD:</td>
<td>Replaces a parameter on a rule with one of its descending parameters (using the children reference).</td>
</tr>
</tbody>
</table>

The seed is configurable. Line 2 indicates the meta-model the seed models conform to. Lines 5–10 define the PPD mutation operator. In particular, line 6 selects one Rule having at least one parameter with descendants, and stores the rule in variable r. Since Rule.parameters is a collection, Rule.parameters.children collects the children of each Parameter object in reference parameters, and flattens the result in a single collection. Then, line 7 selects one parameter of rule r with non-empty children, and line 8 selects one descendant of the selected parameter. By using the function closure, which iteratively collects all reachable elements through a reference, we consider both direct and indirect descendants of the parameter. Line 9 removes the parameter selected in line 7 from rule r, and adds the parameter descendant selected in line 8 to r. In line 10, the range [1..2] allows applying the operator once or twice at random.

```
1 generate mutants in "out/" from "models/"
2 metamodel "http://SecurityPolicy.com"
3 with commands {
4   PPD = [
5       r = select one Rule where { parameters.children <> null }
6       p = select one Parameter in r.parameters where { children <> null }
7       c = select one Parameter in closure(p.children)
8       modify r with { parameters -= p, parameters += c }
9       ] [1..2]
10  ]
11 }
```

Listing 1: Model program encoding the PPD mutation operator.

Figure 3 shows an application of this operator to a seed model. The operator selects one rule having a parameter with children (Personnel), and replaces such a parameter by one of its descendants (Secretary). The objects selected in lines 6–8 of the listing (r, p, c) are marked in the figure.

The screenshot in Figure 2 corresponds to this example. The editor (label 1) contains the definition of all operators in Table 1, although only RTT is visible. The static footprint (label 5) shows that the operators do not mutate classes PolicyType, RuleType and ElementType, which is sensible as those classes are used to model the access control language. However, the footprint also uncovers that no object of type Parameter is created, modified or deleted, and its feature Parameter.children is neither mutated. Since this class is used to define security policies, this suggests the need for further mutation operators dealing with Parameters and their hierarchies.

In the same screenshot, the dynamic footprint shows that the RTT operator
could not be applied on the RBAC seed model (LibraryRBAC). If we inspect the RBAC language definition, we realise that RBAC only defines two rule types with different parameter types. Since RTT requires a rule pair with same parameters, it cannot be applied to any security model expressed with RBAC. This can be confirmed by generating additional seed models for RBAC using the seed model synthesizer.

Altogether, this example illustrates how Wodel simplifies the definition of mutation operators (e.g., the definition of the PPD operator using Kermeta requires three times more LOCs [1]), and how it permits analysing the defined operators using footprints and seed generation.

6. Empirical results

To evaluate the expressiveness and usefulness of Wodel, we have used it to define and analyse sets of mutation operators proposed in the literature. We have implemented mutations for automata [4, 15] with the purpose of automated exercise generation; and mutations for probabilistic automata [16], class diagrams [17], BPEL [18] and security policies [1] for mutation testing. The purpose of this experiment is to assess the expressiveness of Wodel to deal with realistic mutation operators, and to show the usefulness of its metrics to evaluate sets of mutation operators.

Table 2 summarises the results. The columns show the number of implemented mutation operators (we show in parenthesis the number of operators proposed in the original publication); the meta-model size in classes; the mutation percentage including both explicit and implicit Creation, Modification, and Deletion actions; and the percentage of non-mutated concrete classes (column unmodified). The encoding of the operators is available at http://gomezabajo.
Table 2: Evaluating sets of mutation operators.

<table>
<thead>
<tr>
<th>mutation operators</th>
<th>#mm classes</th>
<th>static footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>Automata [4]</td>
<td>10 (10)</td>
<td>50%</td>
</tr>
<tr>
<td>Automata [15]</td>
<td>4 (4)</td>
<td>25%</td>
</tr>
<tr>
<td>Prob. automata [16]</td>
<td>4 (4)</td>
<td>25%</td>
</tr>
<tr>
<td>Class diagrams [17]</td>
<td>50 (50)</td>
<td>47%</td>
</tr>
<tr>
<td>BPEL [18]</td>
<td>18 (26)</td>
<td>13%</td>
</tr>
<tr>
<td>Sec. policies [1]</td>
<td>5 (5)</td>
<td>14%</td>
</tr>
</tbody>
</table>

The metrics shown in the table have been statically computed by Wodel.

The column unmodified reveals none of these works provide full mutation coverage with respect to the domain meta-model, i.e., they do not mutate all meta-model classes and features. The first two sets of automata mutations [4, 15], which are used to automate exercise generation, do not mutate the class Symbol. Hence, the alphabet of the automata cannot get changed, although this would be useful to generate more variety of exercises.

The mutations for probabilistic automata in [16] provide even less coverage: they do not change the alphabet either, there are no deletion mutations, and there are creation mutations only for Transitions. Hence, these mutations yield mutants with same alphabet as the seed, same states, and emulate faults by changing transition targets, their probabilities, the initial state, or adding extra transitions. However, for mutation testing, it would be interesting to add new states, or to delete transitions re-adjusting the siblings’ probabilities.

The class diagram mutations in [17] are quite complete, as indicated by the low percentage of unmodified elements (16%).

The BPEL mutations in [18] have low meta-model coverage, as they do not mutate 50% of the meta-model concrete classes. The reason for this low coverage is that the mutations aim at modelling programming mistakes when implementing WSBPEL 2.0 compositions using graphical tools. From the proposed 26 mutations, we were able to encode 18. The remaining 8 were related to expressions, and we could not define them because the meta-model represents expressions as external objects.

The mutations for security policies [1] have low coverage, but this is because they do not mutate the classes to specify access control languages, but only those to define security policies, which is sensible. Anyhow, we miss being able to mutate Parameters and their hierarchical organization, which would be useful for testing.

Altogether, we could specify most mutations (91/99) in the analysed works using Wodel, demonstrating its expressiveness. The footprints Wodel provides helped in identifying omissions for automata, security policies and BPEL, but confirmed a reasonable coverage for class diagrams.
7. Conclusions and future work

Mutation has many applications in computer science, but there is currently a lack of general approaches to define mutation operators. Wodel fills this gap using a model-based approach to mutation. It has the advantage of being domain-independent, enabling the rapid development of mutation operators using a dedicated domain-specific language. The tool offers advanced functionality for the automatic generation of seed models, and to calculate the static and dynamic footprints of Wodel programs. We have used our tool to define collections of mutation operators defined in the literature, identifying limitations in those sets and showing the usefulness of footprints in practice.

In the future, we plan to extend Wodel with OCL helpers, as well as with smart synthesis of mutation operators that maximize the coverage of the static footprint. We plan to work on further static analysis techniques, e.g., to detect operator conflicts and dependencies. Finally, we are working on a dedicated Wodel post-processor for mutation testing.

Acknowledgements

Work partially funded by project FLEXOR (Spanish MINECO, TIN2014-52129-R), project DArDOS (Spanish MINECO/FEDER TIN2015-65845-C3-1-R) and the R&D programme of the Madrid Region (S2013/ICE-3006).

References


URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928


### Current executable software version

<table>
<thead>
<tr>
<th>Nr.</th>
<th>(executable) Software metadata description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Current software version 1.1</td>
</tr>
<tr>
<td>S2</td>
<td>Permanent link to executables of this version</td>
</tr>
<tr>
<td>S3</td>
<td>Legal Software License EPL-1.0 License</td>
</tr>
<tr>
<td>S4</td>
<td>Computing platform/Operating System</td>
</tr>
<tr>
<td>S5</td>
<td>Installation requirements &amp; dependencies</td>
</tr>
<tr>
<td>S6</td>
<td>If available, link to user manual - if formally published include a reference to the publication in the reference list</td>
</tr>
<tr>
<td>S7</td>
<td>Support email for questions</td>
</tr>
</tbody>
</table>

Table 3: Software metadata

### Current code version

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Code metadata description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Current code version 1.1</td>
</tr>
<tr>
<td>C2</td>
<td>Permanent link to code/repository used of this code version</td>
</tr>
<tr>
<td>C3</td>
<td>Legal Code License EPL-1.0 License</td>
</tr>
<tr>
<td>C4</td>
<td>Code versioning system used</td>
</tr>
<tr>
<td>C5</td>
<td>Software code languages, tools, and services used</td>
</tr>
<tr>
<td>C6</td>
<td>Compilation requirements, operating environments &amp; dependencies</td>
</tr>
<tr>
<td>C7</td>
<td>If available Link to developer documentation/manual</td>
</tr>
<tr>
<td>C8</td>
<td>Support email for questions</td>
</tr>
</tbody>
</table>

Table 4: Code metadata