



Wodel-Test: a model-based framework for language-independent mutation testing

Pablo Gómez-Abajo¹ · Esther Guerra¹ · Juan de Lara¹ · Mercedes G. Merayo²

Received: 12 December 2019 / Revised: 6 June 2020 / Accepted: 17 August 2020
© Springer-Verlag GmbH Germany, part of Springer Nature 2020

Abstract

Mutation testing (MT) targets the assessment of test cases by measuring their efficiency to detect faults. This technique involves modifying the program under test to emulate programming faults, and assessing whether the existing test cases detect such mutations. MT has been extensively studied since the 70's, and many tools have been proposed for widely used languages like C, Java, Fortran, Ada and SQL; and for notations like Petri-nets. However, building MT tools is costly and error-prone, which may prevent their development for new programming and domain-specific (modelling) languages. In this paper, we propose a framework called WODEL- TEST to reduce the effort to create MT tools. For this purpose, it follows a model-driven approach by which MT tools are synthesized from a high-level description. This description makes use of the domain-specific language WODEL to define and execute model mutations. WODEL is language-independent, as it allows the creation of mutation operators for any language defined by a meta-model. Starting from the definition of the mutation operators, WODEL- TEST generates a MT environment which parses the program under test into a model, applies the mutation operators, and evaluates the test-suite against the generated mutants, offering a rich collection of MT metrics. We report on an evaluation of the approach based on the creation of MT tools for Java and the Atlas transformation language.

Keywords Mutation testing · Model mutation · Model-driven engineering · Domain-specific languages · Java · Model transformation

1 Introduction

Mutation testing (MT) is a software testing technique consisting in injecting artificial faults to the program under test to evaluate an existing test suite [21,43]. The hypothesis is that if a test suite is good at distinguishing a program from its mutants, then it is probably good at discovering faults.

Communicated by Benoit Baudry.

✉ Pablo Gómez-Abajo
pablo.gomeza@uam.es

Esther Guerra
Esther.Guerra@uam.es

Juan de Lara
Juan.deLara@uam.es

Mercedes G. Merayo
mgmerayo@fdi.ucm.es

¹ Modelling and Software Engineering Research Group,
Universidad Autónoma de Madrid, Madrid, Spain

² Design and Testing of Reliable Systems Research Group,
Universidad Complutense de Madrid, Madrid, Spain

Hence, MT is an alternative to other techniques to evaluate the strength of a test suite, e.g. based on coverage [6].

MT is based on the *competent programmer hypothesis*, which states that most software faults introduced by experienced programmers are caused by small syntactic errors [21]. MT emulates these errors in the form of mutation operators. These are specifications of small changes (e.g. swapping plus by minus in an arithmetical expression) which are systematically applied to the program under test to produce a set of mutants. Then, the test suite is applied to each mutant. If the original program and the mutant produce different outputs when they are executed against a test case, then the test suite has detected the fault. The mutation score, which is the percentage of mutants whose faults are detected, provides a measure of the test suite quality.

MT has been extensively studied [49,78], and MT tools exist for widely used programming languages like Fortran [55], Java [12,54,65], C [2,47] or SQL [91]. It has also been applied to modelling notations with executable semantics, like Petri nets [27] and state machines [26,46]. Most of the times, MT tools are created manually with no extensi-

bility in mind, and so defining customized sets of mutation operators is not possible. Moreover, building a MT tool is a high investment in terms of effort. This may hamper using MT in emerging programming or domain-specific languages (DSLs) [92] with a low number of users, as without automation, building MT tools for them may not be cost-effective. Finally, there may be further complexity in development if the MT tool needs to tackle heterogeneous technology (e.g. a MT tool for JavaScript may need to deal with HTML pages and CSS documents as well).

To improve this situation, we propose a novel approach for the automated construction of MT tools. Our approach is language-independent, and so it is applicable to any programming or modelling language defined by a meta-model. We rely on a DSL to define mutation operators for the target language [33,34] and use model-driven engineering (MDE) [13] to generate a customized MT tool for the language. To show the usefulness of our proposal, we present two case studies on the development of MT tools for two very different languages: Java and the Atlas model transformation language (ATL) [50].

In [34], we presented the WODEL DSL for model mutation and its extensible development environment that permits incorporating post-processors for different applications, like the automated generation of exercises with auto-correction [33]. In this paper, we report on another post-processing extension—shortly exposed in a previous vision paper [35] and a tool demo [36]—devoted to the automated construction of MT tools for any language. This way, this paper makes the following contributions: (i) an automated approach to construct MT tools for arbitrary languages; (ii) its realization in a tool called WODEL- TEST, which synthesizes Eclipse plug-ins for MT; and (iii) its evaluation by means of the creation of MT tools for Java and ATL. This evaluation aims at answering two research questions (RQs):

- RQ1:** Does WODEL- TEST allow creating MT tools with similar capabilities to existing MT tools developed by hand?
- RQ2:** How effective is WODEL- TEST to specify MT tools?

Overall, the benefits of WODEL- TEST are: (i) reduced effort in creating MT tools; (ii) extensibility of the MT tool with new mutation operators and mutant equivalence criteria; and (iii) simplicity to create MT tools that need to access heterogeneous languages.

The rest of this paper is organized as follows. First, Sect. 2 provides background on MT and MDE and introduces a running example. Next, Sect. 3 presents our approach to specify MT tools, and Sect. 4 shows the proposed architecture and tool support. Section 5 reports on two case studies consisting

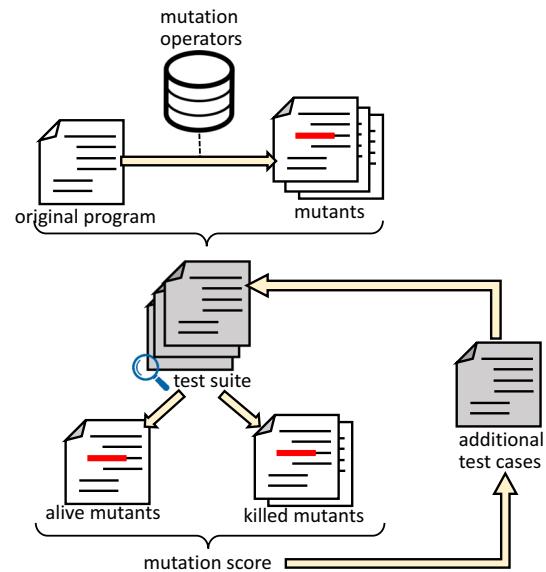


Fig. 1 MT-process

on the development of MT tools for Java and ATL, and compares the former tool with state-of-the-art MT tools for Java. Finally, Sect. 6 analyses related research and Sect. 7 finishes with the conclusions and future work.

2 Background

In this section, we first present the basic concepts of MT (Sect. 2.1) and then introduce the main elements of MDE using a running example (Sect. 2.2).

2.1 Mutation testing

The aim of MT is to assess the quality of a test suite, that is, to measure its effectiveness with respect to its ability to detect faults. Figure 1 describes its process.

First, given a program and a test suite, MT introduces small syntactic changes in the program to generate faulty programs called *mutants*. Mutants are produced by means of *mutation operators* that simulate common faults made by competent programmers. We distinguish between *first-order* mutants if they are obtained by applying a mutation operator once, and *higher-order* mutants if they are generated by applying mutation operators more than once.

Next, the test suite is applied to both the original program and the mutants. If the output produced by a mutant is different from the output produced by the original program for some test in the suite, then the mutant is said to be *killed*; otherwise, it is *alive*. Some mutants may also be *equivalent* to the original program, in the sense that no possible test case can kill them. For example, an operator that adds “+ 0” to an

arithmetical expression yields an equivalent mutant, i.e. the mutant is syntactically different from the original program, but the mutation cannot be detected semantically.

The usefulness of MT is based on the *coupling effect*. This states that the tests able to distinguish all programs differing from a correct one by simple errors are so sensitive that they can also distinguish more complex errors [21]. The adequacy of a test suite is given by the *mutation score*, which corresponds to the ratio of killed mutants over the total number of non-equivalent mutants. If the mutation score is deemed low, then the test suite should be improved with new test cases able to kill the live mutants and consequently improve the score.

2.2 Model-driven engineering by example

Model-driven engineering [13] is a software engineering paradigm where models lead the development process. Hence, models are not merely passive documentation, but are used to describe, analyse, simulate, verify and generate code for the final application, among other activities. Models can be described using general-purpose modelling languages, like the Unified Modelling Language (UML) [74], or by means of DSLs tailored to an area of interest.

In MDE, the abstract syntax of languages is defined by a meta-model. This is itself a model—typically a class diagram—that describes the language primitives, their properties and relations. Additionally, it may contain integrity constraints, often expressed with the Object Constraint Language (OCL) [73]. Valid models are said to *conform* to their meta-model if they contain objects typed by the meta-model classes and obey the meta-model integrity constraints. In addition to an abstract syntax, languages are represented using a notation—a concrete syntax—which can be textual [92] (more typical of programming languages) or graphical [52] (more frequent in modelling languages).

As a running example, we will be using a small subset of Java, for which we will create a MT tool using our approach. As we use MDE, we require a meta-model for the Java language. Figure 2 shows a small excerpt of this meta-model, taken from the model-based application modernization tool MoDISCO [15]. This meta-model contains elements to represent Java assignments, variables, binary expressions, some literals (number, string and null) and constructor invocations.

Figure 3 shows an assignment in Java represented as a model conformant to the previous meta-model (left), and the equivalent representation in textual concrete syntax (right). The object with type Assignment has assign (“=”) as selected operator. Its lhs reference is an object of type Variable with name “myObject”, and its rhs reference is an ObjectCreation object that refers to the Constructor of a class named “MyClass”.

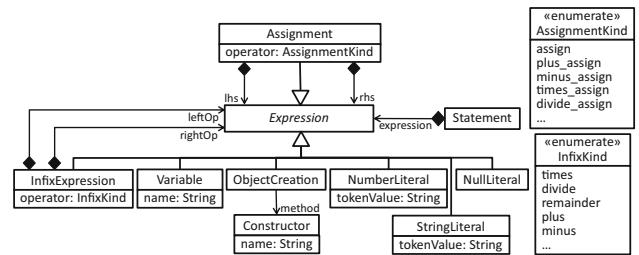


Fig. 2 Java meta-model excerpt (from MoDISCO [15])

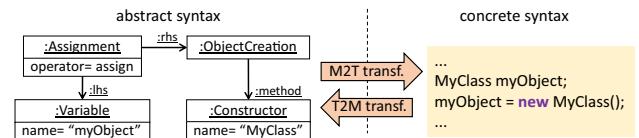


Fig. 3 A Java model fragment (left) and its textual representation (right)

Models are not isolated entities. As part of MDE processes, models can be *merged* to homogenize different versions of a system, *aligned* to create a global system from different views, *refactored* to improve their internal structure without changing their behaviour, *refined* into lower-level models, or *transformed* into other languages, e.g. for code generation or analysis [13].

These model operations are implemented as transformations which can be either model-to-model (M2M), text-to-model (T2M) or model-to-text (M2T), being their difference the nature of their input and output artefacts (models or text). In the context of this work, we are interested in T2M and M2T transformations. T2M transformations can be used to realize reverse-engineering processes that transform an input text into an output model. In Fig. 3, the generation of the abstract syntax model from the textual representation of a Java program is a T2M transformation. M2T transformations are used to automate software engineering tasks like the generation of documentation, configuration files or program code. In Fig. 3, the synthesis of the textual representation of a Java program from its abstract syntax model is a M2T transformation.

3 Specifying mutation testing tools

In this section, we detail our method to automate the synthesis of MT tools for arbitrary languages. Section 3.1 starts with an overview of our approach, and Sects. 3.2–3.4 provide the details.

3.1 Overview

Nowadays, most MT tools are created manually. However, this is costly as it implies: defining a set of proper mutation operators for the language; transforming the program

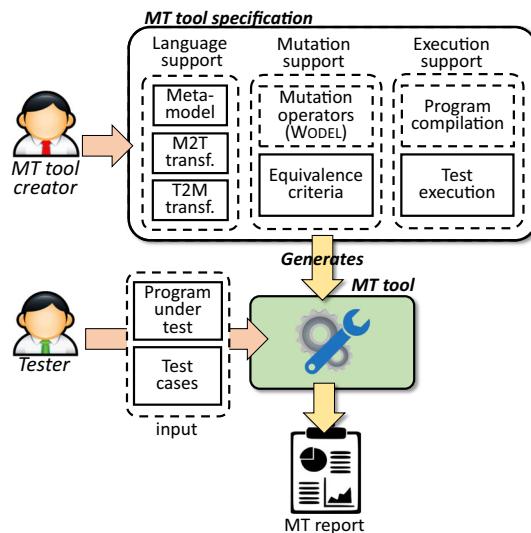


Fig. 4 Creating and using a MT tool with WODEL- TEST

under test into a format (e.g. bytecode, abstract syntax tree) that enables the application of the operators; applying the operators to the program exhaustively to create mutants of it; executing the test cases on the mutants; and reporting the results of the MT process. In addition, MT tools can incorporate advanced mutation features like detection of equivalent mutants; mutant filtering [51], reduction [37], sampling [96] and prioritization [51] to optimize the execution time; automated generation of test cases to improve the mutation score [30]; and generation and testing using higher-order mutants [48]. Implementing these features is challenging because they are frequently language-specific.

Overall, all these different features that a MT tool need to implement can be classified into *language support* (i.e. transforming the program under test into a suitable internal format and vice versa), *mutation support* (i.e. definition of mutation operators and other advanced mutation features), and *execution support* (i.e. compiling the program, running the test cases and calculating metrics). The objective of our approach is to facilitate the construction of MT tools by providing support to configure these three types of activities. For this purpose, we rely on models as the internal representation of programs; on a DSL to define mutation operators over the model-representation of programs; on extension points to configure additional mutation features (currently, only the definition of mutant equivalence criteria); and on MDE to synthesize the MT tools.

Figure 4 shows a scheme of our MDE-based approach to specify MT tools. It distinguishes two roles: the *MT tool creator* and the *tester*. The former provides a MT tool specification from which a MT tool is synthesized. The latter uses the generated tool to perform MT. Next, we detail the activities performed by each role.

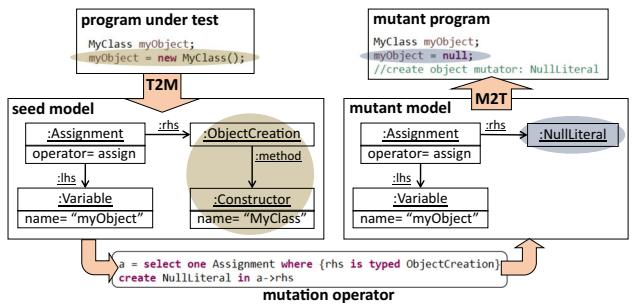


Fig. 5 Working scheme of the MT tool for Java

First, the MT tool creation takes place. For this purpose, the MT tool creator needs to provide the following three groups of artefacts:

- specification of *language support* in the MT tool. This consists of the meta-model of the target (programming or modelling) language, a T2M transformation to parse the textual syntax of programs into models, and a M2T transformation to serialize the models into text. Section 3.2 provides details on these artefacts, and how our approach covers both graphical and textual programming and modelling languages.
- specification of *mutation support* in the MT tool. This corresponds to the definition of the mutation operators of interest and syntactic/semantic mutant equivalence criteria. We provide a DSL called WODEL to facilitate the definition of mutation operators [33]. Section 3.3 will describe WODEL and how to specify equivalence criteria.
- specification of *execution support* in the MT tool, that is, a description of how to compile programs of the target language and execute test cases. Section 3.4 will explain how to specify this part.

Our system takes this specification as input and automatically synthesizes a MT tool tailored to the language. The tester can use this tool by providing a program under test and a set of test cases, and the tool outputs different metrics including the mutation score. Section 4 will present the functionality of the generated MT tools.

As an example, Fig. 5 illustrates the internal working scheme of the MT tool generated for our running example. First, the Java program under test is parsed into a model conformant to the meta-model of Fig. 2 using the specified T2M transformation. Then, the model is mutated using the defined mutation operators, and the mutant is serialized back to textual format using the M2T transformation. Finally, the original and mutant programs in text format are compiled and the test cases are executed against them.

The working scheme so defined is akin to MT approaches working at the abstract syntax tree (AST) level. However,

models are not restricted to have tree shape (models are graphs), and their concrete syntax does not need to be textual.

3.2 Specification of language support

As mentioned above, the first set of artefacts of a MT tool specification concerns the language support. Specifically, since our approach relies on MDE, the MT tool creator needs to provide the meta-model of the target language. For the running example, this is the meta-model shown in Fig. 2.

In addition, if the programs of the language are textual source code files, the tool creator needs to provide a T2M transformation that creates a model out of the text files, and a M2T transformation to serialize the model into text after applying the mutation operators. If the language is not textual but graphical, there are two possibilities: if the language has separate representations for its concrete and abstract syntax (as in Sirius- or GMF-based graphical languages [32,85]), then there is no need to provide any transformation as the mapping between both syntaxes can be reused instead; however, if the language only has concrete syntax, then transformations to obtain the abstract syntax model out of the graphical syntax, and vice versa, should be provided. Such transformations can be either M2M or M2T/T2M depending on the technology the language is built in, but in any case, this does not have any impact in the design of our architecture. Finally, if we are working with models without concrete syntax (e.g. pure models defined with the Eclipse Modelling Framework (EMF) [87], which only contain abstract syntax information), then these transformations are not needed.

The three required language-related artefacts (meta-model and T2M/M2T transformations) may exist or else they need to be created. For modelling languages (e.g. UML, BPMN) and some custom-made DSLs, these artefacts already exist and can be reused. For programming languages, the situation varies. Since MDE is a technology used for re-engineering software projects [1], meta-models and M2T/T2M transformations for languages like languages like Java, COBOL, C++, C# or SmallTalk are available [15,29,64]. However, if these artefacts do not exist for a given programming language, their construction may require a substantial amount of work. This effort can be alleviated by the use of frameworks like EMFText [44] and Xtext [92], which automate the creation of the meta-model and M2T/T2M transformations from an EBNF grammar of the language. For example, in the JaMoPP re-engineering framework [45], a Java meta-model with 237 classes was created by hand, while the MT2/T2M transformations were synthesized from an EBNF grammar written in EMFText, consisting of 154 rules and 665 LOC.

```

1 generate exhaustive mutants in "out/" from "model/"
2 metamodel "mm/java.ecore"
3 description "Simple Wodel program"
4
5 with blocks {
6   CIR 'Replace a call to a constructor by null' {
7     a = select one Assignment where {rhs is typed ObjectCreation}
8     create NullLiteral in a->rhs
9   }
10 }
```

Listing 1 WODEL program defining the CIR mutation

3.3 Specification of mutation support

In order to specify the mutation operators, the MT tool creator must use the WODEL DSL [34]. As an example, Listing 1 shows a simple WODEL program that defines a mutation operator replacing a call to a constructor by null (CIR) [60]. Figure 5 illustrates an application of this operator to a model.

A WODEL program has two parts: a header declaring the language the mutations are defined for and configuring the mutation process, and a body containing the definition of the mutation operators. In Listing 1, the header goes from lines 1 to 3. Line 1 defines the mutant generation mode (*exhaustive*), the output folder that will contain the generated mutants (*out*) and the input folder with the seed models (*model*). WODEL supports two modes to generate mutants. The first one (illustrated in the listing) is enabled by the keyword *exhaustive* and generates all possible mutants of the seed models. This mode is adequate for MT. The second mode consists in defining a maximum number of mutants, which are generated randomly among all possible ones. This mode can be appropriate for other mutation applications like the automated generation of exercises [33]. This maximum number is a default value that can be modified in the resulting MT tool without the need to regenerate the tool.

Line 2 indicates the meta-model of the language the seed models conform to, and line 3 provides a description of the program, which is optional. Although not illustrated in the listing, we have extended WODEL to allow including the meta-model of auxiliary resources that can be queried in the mutation process but are not mutated. For example, the mutation operators for a JavaScript project may need to query HTML or CSS documents, in addition to the JavaScript program. Section 5.2 will illustrate the use of resources.

Lines 5–10 of Listing 1 contain the body of the program with the mutation operators. The operators can be organized in blocks (line 5) and use the mutants generated by other blocks as seed models, e.g. to define higher-order mutants. Lines 6–9 declare a mutation operator named CIR composed of two statements. The first one (line 7) stores in variable *a* one random *Assignment* object whose reference *rhs* contains an object with type *ObjectCreation*. Note that *Assignment* and *ObjectCreation* refer to types from the Java meta-model

(cf. Fig. 2), as the mutation is applied to Java programs. The second statement (line 8) creates a new NullLiteral object in the rhs reference of the variable a. Overall, this program generates mutants replacing an object creation in the right-hand side of a Java assignment by `null`.

WODEL provides primitives to *select*, *create*, *clone*, *modify*, *retype* and *delete* information from models. Table 1 illustrates these primitives using the running example. In each case, we show a sample WODEL program and its application to an excerpt of Java code. In particular, WODEL supports the following mutation primitives:

- *Object selection*. Primitive `select` selects an object with the given type and field values. The table shows two examples in rows 1 and 2. In the first case, the WODEL program selects an object of the type `Assignment` whose `rhs` reference is of the type `ObjectCreation`. In the second case, the program selects two objects of the type `InfixExpression` having the value “+” and “%” in their field `operator`.
- *Object/reference creation*. Primitive `create` creates an object of the given type and initializes its fields. The new object is placed inside the specified composition reference of an existing object, or if none is given, WODEL selects one compatible container object at random. This primitive is used in the first row of Table 1 to create an object with type `NullLiteral` and store it on the `rhs` reference of a previously selected `Assignment` object. Primitive `create reference` creates a reference of the given type between two selected objects.
- *Object cloning*. Cloning permits creating an object using another one as a template. It can be used to emulate duplication errors (e.g. redundant variable definitions in a Java program) or to build nearly-exact copies of complex objects (e.g. cloning a Java method, and then changing its name). As some authors have observed, it is frequent that a program with an error includes other similar cases handled correctly [62].
- *WODEL provides the primitive `clone` to make a shallow copy of an object. In the deep variant, the operator clones in addition all other reachable objects via composition references. As in the primitive for object creation, it is possible to specify a container reference for the clone; otherwise, the clone is stored in a random compatible container reference. The primitive is illustrated in row 2 of the table, where an `InfixExpression` with operator “%” is deep-cloned and stored in the `rightOp` reference of a previously selected `InfixExpression` object.*
- *Reference/attribute modification*. Primitive `modify` is used to change the value of the fields of an object. The table shows two examples modifying a reference (row 3) and an attribute (row 4). In the first case, the WODEL pro-

gram modifies the `rightOp` reference of an expression by assigning to it a different `NumberLiteral` object. In the second case, the program changes the attribute `operator` of an `InfixExpression` from “+” or “-” to “*”.

This primitive can also be used with modifiers `source` and `target` to redirect the source or target of a reference to another object. The table shows an example in row 5, where the target of a reference called `method` is changed from one `Constructor` to a different one.

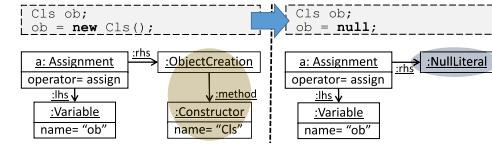
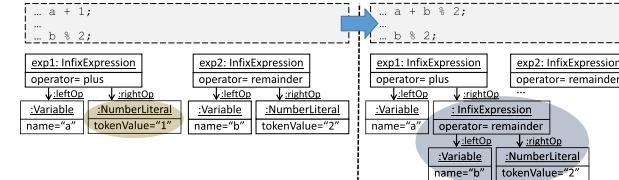
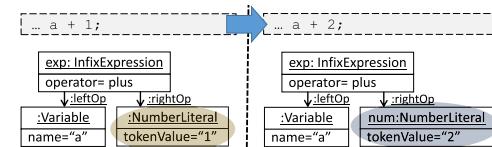
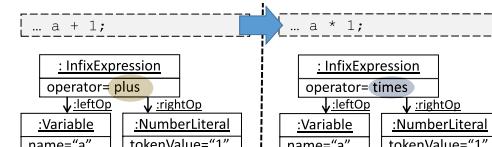
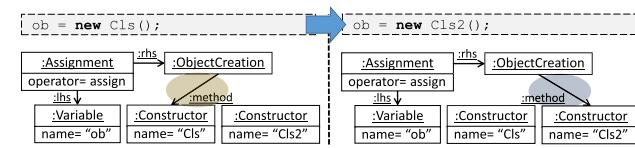
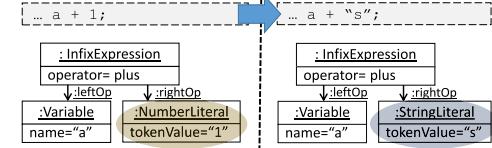
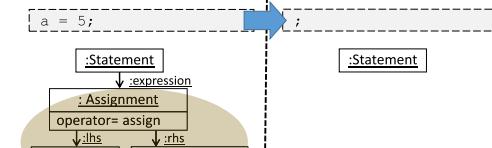
- *Object retying*. Primitive `retype` is used to retype an object to one of its sibling types, preserving all compatible attributes and references from the original object. In the example of row 6, the operator retypes a `NumberLiteral` object with value “1” to a `StringLiteral` object with value “s”.
- *Object/reference deletion*. Primitive `remove` deletes an object safely ensuring that no dangling edge to/from the object remains. Primitive `remove reference` deletes a reference of the given type between two selected objects. The table shows an example of object deletion in row 7, where the WODEL program deletes an `Assignment` object and its contents.

As the examples show, mutation primitives may need to select the objects to mutate. For this purpose, WODEL offers different selection strategies: selecting a random object, a specific object previously selected and stored in a variable, all objects satisfying some condition, a different object to the one selected by the current mutation, or an object of a given type. All strategies can be parameterized with a condition on the attributes and reference values of the selected object. For example, the object deletion mutation statement in row 7 of Table 1 uses a random selection strategy (one `Assignment`) parameterized with an attribute condition (where `{operator = '='}`).

Instead of relying on a general-purpose programming language, our approach provides the DSL WODEL to implement the mutation operators. The advantage is that WODEL has facilities specific to model mutation that otherwise would need to be encoded by hand. For example, it automatically selects an appropriate container reference¹ for the created and cloned objects; it avoids dangling edges from/to any deleted object; it ensures that any generated mutant is syntactically correct and satisfies the integrity constraints of its meta-model; and it supports the definition of extra OCL conditions that every mutant model is also enforced to satisfy. The latter is useful to rule out invalid mutants, e.g. non-executable ones. This check can be strengthened by the developer by adding custom criteria in an extension point

¹ WODEL is built atop EMF. In EMF, it is customary to place all objects (except one, acting as root) inside some composition reference, called its *container reference*.

Table 1 Examples of Wodel mutation primitives

	Primitive	Example of Wodel program	Example of model mutation
1	Object creation	// replaces a constructor call by null a = select one Assignment where {rhs is typed ObjectCreation} create NullLiteral in a->rhs	
2	Object cloning	// replaces right operand of a '+' expression // by a clone of an existing '%' expression exp1 = select one InfixExpression where {operator = '+'} exp2 = select one InfixExpression where {operator = '%'} deep clone exp2 in exp1->rightOp	
3	Reference modification	// modifies the numeric right operand of // an expression by another number exp = select one InfixExpression where {rightOp is typed NumberLiteral} num = select one NumberLiteral where {self <> exp->rightOp} modify exp with {rightOp = num}	
4	Attribute modification	// modifies the operator '+' or '-' of an expression by '*' modify one InfixExpression where {operator in ['+', '-']} with {operator = '*'}	
5	Target reference modification	// modifies a constructor call to a different one modify target method from one Constructor to other Constructor	
6	Object retyping	// retypes a numeric literal as a string literal rtype one NumberLiteral as StringLiteral with {tokenValue = 's'}	
7	Object deletion	// removes an assignment with operator '=' remove one Assignment where {operator = '='}	

aimed at checking mutant validity. Finally, an external DSL to define mutation operators provides extensibility and customizability of the operator set, e.g. enabling the creation of purpose-specific operators or operators emerging from novel language features (e.g. Java lambdas), as well as the removal of obsolete operators.

In addition to mutation operators, the MT tool creator can specify equivalence criteria for mutants. We distinguish syntactic equivalence from semantic equivalence, both of which are customizable. The former include mutants that are duplicated or have equivalent syntax (e.g. the mutant removes a comment or adds an additional parenthesis pair). The latter

include checks on the model semantics or its compilation. For example, for a Java MT tool, we may consider that two Java mutants are equivalent if they compile to the same bytecode [56]. In DSLs that enable describing models with finite behaviour (e.g. some classes of state-machines), we can use techniques based on language equivalence [24]. Distinguishing equivalent mutants is relevant for MT because they behave like the original program when the test suite is applied; therefore, they should not be included in any measure that estimates test effectiveness. WODEL- TEST provides flexibility in this respect by supporting the customization of the equivalence criteria used in the MT tool being defined.

3.4 Specification of execution support

The last part of the MT tool specification is the definition of how to programmatically compile the program under test and the generated mutants, and how to execute the test cases.

For some modelling languages (e.g. automata), compilation might not be necessary. In such cases, there is no need for a programmatic compilation, but the MT process still requires the ability to execute the test cases. Hence, the MT tool creator always needs to specify how to execute the programs against the test suites.

Overall, starting from the specification of the MT tool (language, mutation and execution support), our approach automatically synthesizes a tool that allows performing MT on programs of the given language, collecting the results, and calculating metrics related to the MT process. The next section presents the architecture of our supporting tool as well as the functionalities of the synthesized MT tools.

4 Tool support

This section details the realization of our approach in a tool called WODEL- TEST. In the following, Sect. 4.1 describes its architecture, and Sect. 4.2 shows the functionalities of the MT tools it synthesizes.

4.1 Defining MT tools with WODEL- TEST

Figure 6 shows the modular, component-based architecture of WODEL- TEST consisting of a set of Eclipse plug-ins [36]. It uses the Eclipse Modelling Framework (EMF) [87] as the underlying modelling technology, and extends the engine of the WODEL tool presented in [34] with functionalities specific to MT (label 1). WODEL and WODEL- TEST are available at <http://gomezabajo.github.io/Wodel/>. The web page also includes installation instructions, examples, video demos and the source code.

WODEL is a generic environment to define and apply model mutation operators using the DSL presented in

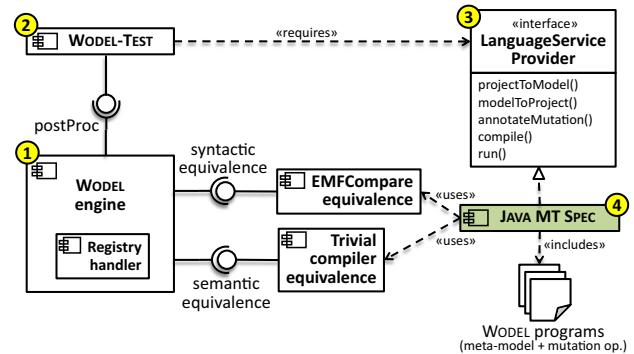


Fig. 6 Architecture of WODEL- TEST

Sect. 3.3. When mutating the models, WODEL creates a registry of applied mutations that is useful for traceability. In addition, WODEL offers three interfaces to allow extensibility. The first one permits specifying syntactic equivalence criteria. The default criterion is model comparison implemented with EMFCompare,² though this can be changed. The second interface permits defining semantic equivalence criteria. For instance, for the running example, we have implemented a trivial compiler equivalence checker that compares the binaries of the original and mutant programs for equality [56]. The last interface permits registering tools to post-process the generated mutants. WODEL- TEST (label 2) is designed as a post-processing extension of WODEL.

When building a new MT tool, defining equivalence criteria is optional, but WODEL- TEST necessarily requires the MT tool creator to provide one or more WODEL programs with the definition of the mutation operators, as well as to implement an interface called **LanguageServiceProvider** (label 3). The latter implies implementing the following methods:

- **projectToModel**: This is the method to convert the artefact to be mutated into an EMF model (i.e. it performs a T2M transformation). In our example, we use MODISCO [15] to convert a Java project into a model, and provide a wizard to select the classes to include in the MT process. The latter enables an optimization of the T2M transformation in case of large Java projects.
- **modelToProject**: This method must take care of converting a model back into a domain artefact (i.e. it performs a M2T transformation). In our example, we use MODISCO to serialize a model conformant to the MODISCO Java meta-model into a Java project.
- **annotateMutation**: This method can be used to insert comments in the mutated code, explaining the performed mutations. WODEL- TEST relies on the WODEL mutation registry to identify the applied mutations. This way, the MT tool creator only has to identify the meta-model class

² <https://www.eclipse.org/emf/compare/>.

used to represent comments in the language, and add an instance of it to the mutant models.

- **compile:** This method must include the code to compile the mutated artefacts.
- **run:** This method must encode how to run the programs against the test suite. In our running example, we use the JUNIT framework for unit testing and dynamically call the test cases included in the test suite.

If an EMF definition of the target language already exists, the implementation of `projectToModel` and `modelToProject` is not typically needed. For interpreted languages, implementing `compile` is not necessary either.

Starting from this specification, WODEL- TEST synthesizes an Eclipse tool that implements a generic MT process. This process first invokes `projectToModel` to convert the program under test into a model; then, it applies the mutation operators in the specified WODEL programs; next, it calls `modelToProject` to convert the mutant models into code; and finally, it executes the test suite against the program and mutants by invoking `compile` and `run`. In the next section, we explain the main functionalities of the generated MT tools.

4.2 Functionality of the synthesized MT tools

The tools synthesized by WODEL- TEST permit performing MT over programs written in the selected language (Java in our running example) and their test suites. The tester can select the mutation operators to apply in the MT process, among the ones defined by the tool creator. This is done using a preferences window like the one shown in Fig. 7, which corresponds to the MT tool created for Java. This window shows each block of mutation operators (see Listing 1) in a separate group.

The results of the MT process can be visualized in four different Eclipse views. Figure 8 shows the global view of results after applying the MT process to a Java project. Label 1 corresponds to the Eclipse project explorer, which contains a folder `src` with the Java project under test, and one folder for each mutation operator applied. Label 2 shows one mutant, where a comment below the mutated code explains the performed mutation. In this case, the operator replaced “%” by “/”. Label 3 shows the test suite. Both the test suite and the Java project under test are standard Eclipse projects. Label 4 shows the global view of results which reports the mutation score; the running time of the MT process; the number of applied and not applied mutation operators;³ the number of killed, equivalent and live mutants; and the number of failed

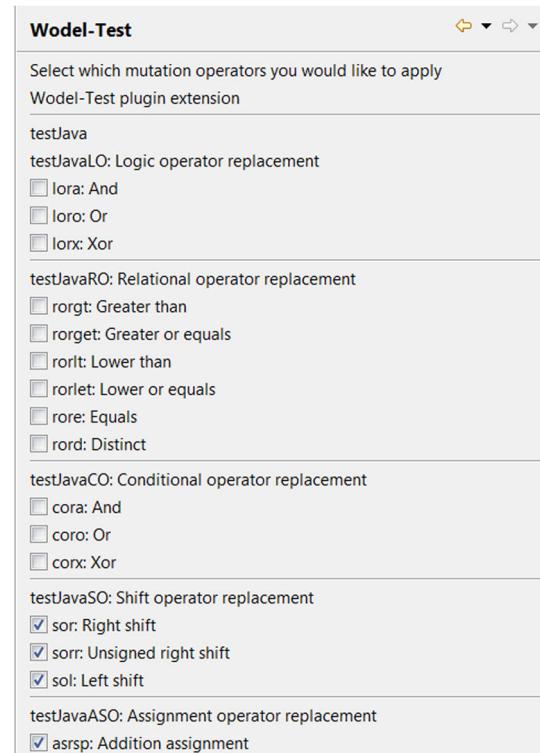


Fig. 7 Selection of mutation operators in the preferences window (excerpt)

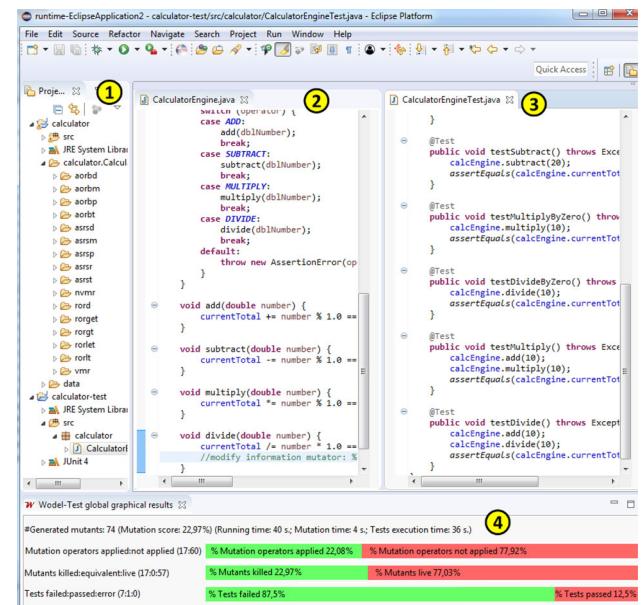


Fig. 8 Global view of MT results

and passed tests. This information is also displayed using bar graphs with the corresponding percentages.

The synthesized tool offers two detailed views of results focussing on mutants (tests passed/failed) and tests (mutants killed/alive). Figure 9 shows the mutants view, with the

³ A mutation operator may not get applied if its application conditions do not occur in the source program, or if it always produces incorrect programs.

WODEL-TEST mutant results						
Filter	Results	Equivalent	Package/class/mutant	#Exe...	#Fail...	#Pass...
All	calculator	calculator	calculator	264	16	248
Failed	CalculatorEngine	CalculatorEngine	/calculator/src/calculator/C...	264	8	248
Passed			/calculator/src/calculator/C...	8	8	0
			/calculator/aorbdb/Output0...	8	8	0
			/calculator/aorbdb/Output1...	8	8	0
			/calculator/aorbdb/Output0...	8	8	0
			/calculator/aorbdb/Output1...	8	8	0
			/calculator/aorbdb/Output0...	8	8	0
			/calculator/aorbdb/Output1...	8	8	0
			/calculator/aorbdb/Output0...	8	8	0
			/calculator/aorbdb/Output1...	8	8	0
			/calculator/asrsd/Output0...	8	3	5
			/calculator/asrsd/Output1...	1	1	0
			testSubtract	1	1	0
			testDivide	1	1	0
			testMultiplyByZero	1	1	0
			testAdd	1	1	0
			testMultiply	1	1	0
			testGetTotalStringInt	1	1	0
			testEqual	1	1	0
			testDivideByZero	1	1	0
			/calculator/asrsd/Output1...	8	1	7
						modify information mutator: *= replaced by /=

Fig. 9 Detailed view of MT results: tests passed and failed by each mutant

results of the tests passed and failed by each mutant. Failed tests (i.e. those that detected at least one mutant) are shown in green, and passed tests are shown in red. The results can be filtered to show only the failed or the passed tests. In addition, for each mutant, the view includes a description of the applied mutation operators (sixth column), as well as whether the mutant is equivalent or not according to the syntactic and semantic equivalence criteria specified by the MT tool creator (first column). The tester is allowed to mark any live mutant as equivalent. In such a case, the tool automatically recalculates the statistics of the global view discarding the equivalent mutants. By double-clicking in the mutant path, the tool opens the mutant program in the Eclipse editor.

The second detailed view is similar, but displays the mutants killed and left alive after executing the test cases. Just like in the mutants view, the tester can open the mutant files by double-clicking on their displayed path, and filter to show only the live or killed mutants.

Finally, there is a last view with details of the mutation generation process, listing the mutants that each mutation operator generated. In this view, it is possible to filter the applied/non-applied mutation operators.

5 Evaluation

In this section, we evaluate our approach from the point of view of the two user roles involved in the process: the tester and the MT tool creator.

Regarding the tester, Sect. 5.1 presents an experiment assessing the usefulness of the generated MT tools. For this purpose, we compare the functionality of the MT tool for Java developed using our approach, with other existing Java MT tools. This comparison aims to answer the following research question:

RQ1: Does WODEL- TEST allow creating MT tools with similar capabilities to existing MT tools developed by hand?

To assess our approach from the point of view of the MT tool creator, Sect. 5.2 presents a case study detailing the construction of a MT tool for the Atlas transformation language (ATL) [50]. This is a rule-based language widely used in MDE to describe model-to-model transformations. We use it as a case study because it has some aspects (e.g. the need to access several artefacts during the mutation process) that permit illustrating the full potential of WODEL- TEST. Our goal with this case study is to answer the following research question:

RQ2: How effective is WODEL- TEST to specify MT tools?

5.1 Comparison of synthesized and hand-made MT tools for Java

In this section, our goal is to assess whether the functionalities of the MT tools generated with our approach are comparable with those available in manually developed MT tools. For this purpose, we compare WODEL- TEST/Java, which is the MT tool for Java that we have created using WODEL- TEST, with some MT tools developed for the application of MT to Java. Among them, we consider five representative tools that have been previously used in the literature to perform comparisons of MT tools [57, 68]: Major [51], Javalanche [81], PITest [17], LittleDarwin [79] and muJava [67].

We structure our comparison in three parts. First, we look at general features of the tools (Sect. 5.1.1); then, we focus on the extensibility and customizability of mutation operators (Sect. 5.1.2); finally, we analyse the efficiency of the MT process (Sect. 5.1.3). The section concludes by answering RQ1 and discussing threats to validity (Sect. 5.1.4).

5.1.1 General features

In the following, we discuss the general features of the analysed MT tools, which are also summarized in Table 2. We have organized the comparison along three criteria: input to the MT process, mutant generation process and reporting.

Regarding the first criterion, Major, Javalanche and LittleDarwin run from the command line; muJava has a graphical user interface; WODEL- TEST/Java is available as an Eclipse plug-in; and PITest can run on the command line but also provides plug-ins for integration with Maven, Gradle, Eclipse and IntelliJ, which enables a rich integration

Table 2 Comparison of Java MT tools

	Major	Javalanche	PTTest	LittleDarwin	muljava	WODEL-TEST/Java
Version/Year	1.3.4/2018	0.3.6/2011	1.4.3/2018	0.2/2018	4/2015	1.0/2019
LOC	Unavailable	15,638	15,804	14,746	24,243	305+framework
<i>Input to MT process</i>						
UI	Command line	Command line	Command line	Command line	Java GUI	Eclipse plug-in
Scope	project/class	project	Tool plug-ins	project	project/class	project/class
Test suite	JUNIT 4	JUNIT 4	project/class	JUNIT 4	JUNIT4	JUNIT 4 (customizable)
<i>Mutant generation process</i>						
No. of operators	30 (default)	19	40	28	47	77 (default)
Op. extensibility	Yes (DSL)	No	Yes (API)	No	No	Yes DSL)
Mutated artefact	Bytecode	Bytecode	Bytecode	AST	Bytecode	Model
Mutant source	Yes	No	Yes	Yes	Yes	Yes
Mutant filter	Yes	Yes	Yes	No	No	No
Equivalent mutant detection	No	No	No	No	No	Yes (TCE)
<i>Reporting</i>						
Report type	CSV	HTML	HTML	HTML	GUI	Interactive views
No. of mutants	✓	✓	✓	✓	✓	✓
Mutation score	✓	✓	✓	✓	✓	✓
Killed/live mutants	Number	Number	Number	Number	Number, list	Number, list
Covered operators	Number	Number	Number	Number, list	Number, list	Number, list
Mutants by class	✗	✗	✗	✓	✓	✓
Mutants by test	✗	✗	✗	✗	✓	✓
Tests by mutant	✗	✗	✗	✗	✗	✓

of MT with developer processes, including continuous integration and reporting directly to the version management system. All tools can perform MT on Java projects; in addition, Major, PITest, muJava and WODEL- TEST/Java permit reducing the scope to specific classes, which is useful to narrow the MT process in the case of large projects. All tools support test suites specified with JUNIT 4. WODEL- TEST/Java also supports JUNIT 4, but extending the MT tool with the last versions of JUNIT (like the current JUNIT 5⁴) or other unit testing frameworks is possible.

Regarding mutant generation, each tool provides its own set of predefined Java mutation operators, and only Major and WODEL- TEST permit extending this set using DSLs (Major-Mml and WODEL respectively). We will compare these DSLs in Sect. 5.1.2. Major, Javalanche, PITest and muJava perform the mutation at the bytecode level, while LittleDarwin and WODEL- TEST/Java apply mutations to the AST and model-based representations of the source code. All tools working at a higher-level representation (AST, models) can make the mutants available for their inspection. Interestingly, this is also possible in most of the tools working at the bytecode level (but Javalanche does not offer this option). To improve the performance of the MT process, Major, Javalanche and PITest carry out a pre-filtering of mutants based on statement coverage, as a mutant that is not reached and executed cannot be detected under any circumstance. To avoid evaluating these uncovered mutants, conditional mutation collects information about the mutation coverage at runtime [51]. Please note that PITest permits configuring the filtering criteria using an extension point, and also includes extensions points to define test prioritization policies. Finally, another way to improve the performance is the exclusion of equivalent mutants from the MT process. Only Javalanche and WODEL- TEST/Java support this mechanism. Javalanche detects equivalent mutants by assessing the impact of mutations on dynamic invariants. In our case, equivalent Java mutants are detected by using trivial compiler equivalence (TCE). Experiments performed on both C and Java showed that TCE allows reducing the number of mutants by 11% for Java and 28% for C [56]. In addition, WODEL- TEST permits incorporating other equivalence criteria by implementing an extension point (see Fig. 6).

With respect to the output of the MT process, Major, Javalanche, PITest and LittleDarwin produce file-based reports; muJava provides this information in static views; and WODEL- TEST/Java can also visualize the results on interactive Eclipse views. In addition, PITest provides an extension point to define mutation result listeners, which can be configured to produce reports in other formats. Regarding the information produced, all tools report the number of generated mutants, the mutation score, and the number of killed

```

1 // Customize replacement list for AOR
2 BIN (*) -> {/, \%};
3 BIN (/) -> {*}, \{ \% \};
4 BIN (\%) -> {*}, /;
5
6 // Customize replacement list for ROR
7 BIN (>) -> {<, <=, !=, ==, >=};
8 BIN (==) -> {<, <=, !=, >, >=};
9
10 // Invoke mutation operators
11 AOR;
12 ROR;

```

Listing 2 Major-Mml program

and live mutants. Additionally, WODEL- TEST/Java provides a view with the list of killed and live mutants, and it is possible to navigate to them (see Fig. 9). All tools but LittleDarwin report the number of covered mutation operators (i.e. operators which produced a non-empty set of mutants). In addition, PITest, muJava and WODEL- TEST/Java also report the list of covered operators and the mutants generated by each one of them. PITest, LittleDarwin, muJava and WODEL- TEST/Java report the mutants generated from each class; muJava and WODEL- TEST/Java provide the information on the mutants killed by each test; and only WODEL- TEST/Java provides the information on the tests passed or failed by each mutant (see Fig. 9).

5.1.2 Extensibility of mutation operators

To the best of our knowledge, only PITest, Major and WODEL- TEST allow extending the set of predefined mutation operators. PITest provides an API for this purpose, so that operators must be defined programmatically with a general-programming language not designed for mutation. Instead, Major and WODEL- TEST provide dedicated DSLs. Listings 2 and 3 illustrate the DSLs of both tools on the basis of the same example.

Major provides a DSL (actually a script language) called Major-Mml to create mutation operators and configure the mutation process. Major-Mml provides 8 mutation primitives, 7 of them to replace operators and literals and 1 to delete statements. These primitives are the following: Arithmetic Operator Replacement (AOR), Logic Operator Replacement (LOR), Shift Operator Replacement (SOR), Conditional Operator Replacement (COR), Relational Operator Replacement (ROR), Literal Value Replacement (LVR), Operator Replacement Unary (ORU) and Statement Deletion (STD). These primitives can be customized with a specific package, class or method of application. In addition, it is possible to define a custom replacement list for the replacement primitives, as lines 2–4 and 7–8 of Listing 2 show for the AOR and ROR operators.

In contrast, as we have seen in Sect. 3.3, WODEL is a full-fledged DSL that enables the creation of arbitrary mutation

⁴ <https://junit.org/junit5/>.

```

1 AOR "Arithmetic operator replacement" {
2   modify one InfixExpression
3   where {operator=['*', '/', '%']}
4   with {operator=['*', '/', '%']}
5 }
6
7 ROR "Relational operator replacement" {
8   modify one InfixExpression
9   where {operator=['>', '==']}
10  with {operator=['<', '<=', '!=', '==', '>', '>=']}
11 }

```

Listing 3 WODEL program

operators, and not just the configuration of replacement lists. For instance, the mutation operator presented in Listing 1 cannot be defined with Major-Mml, as it requires identifying invocations to constructors. This demonstrates the expressivity of WODEL.

5.1.3 Efficiency and efficacy

In this section, we look at efficiency (*how many mutants per second can WODEL-TEST/Java produce?*) and efficacy (*are the generated mutants appropriate for evaluating the test cases?*). For this purpose, we perform an experiment running a MT process over a Java library created by a third party. Our goal is to evaluate to what extent we are able to produce results similar to other MT tools regarding the efficiency and efficacy of the MT process.

In the experiment, we used the functional-matrix-operator project⁵ which has 74 Java classes and 2586 LOC, with a test-suite having 10 Java classes and 647 LOC. The experiment was performed on a PC with an Intel Core i3-2350M processor and 8GB of RAM, under a Linux Ubuntu 16.04.6 LTS OS.

Table 3 summarizes the results. The columns show the number of generated mutants distinguishing between killed and alive, the mutation score, the total running time and the average time per mutant. The measured times include the time to generate the mutants and the time to execute them together.

Regarding efficacy, the MT tools report mutation scores ranging from 20.21% to 34.97%, being Major and WODEL-TEST/Java the ones with the lowest scores. However, WODEL-TEST/Java generated more mutants, mainly because its replacement operators are more exhaustive. Whereas LittleDarwin and PITest only apply one of the possible changes considered for replacement operators, WODEL-TEST/Java applies all of them. For example, in the case of AOR, WODEL-TEST/Java replaces each occurrence of the arithmetic operators by the other four, while LittleDarwin and PITest only perform a replacement. Another reason is that

WODEL-TEST/Java has the largest set of mutation operators, as it includes all operators defined by the other tools (“Appendix A” contains the complete list of operators). In any case, note that WODEL-TEST/Java allows applying a subset of its operators (cf. Fig. 7).

The table does not include the results for Javalanche because, in spite of our best efforts, we were not able to complete the execution for all tests. We had to exclude some tests to get an execution without errors, after which the tool generated 758 mutants and killed 75 of them, with a mutation score of about 10%.

With respect to efficiency, WODEL-TEST/Java took around 3 hours and a half to execute the whole MT process, which is less time than Major, LittleDarwin and muJava, but more time than PITest. Note however that WODEL-TEST/Java generated 10 times more mutants than LittleDarwin, 5 times more mutants than PITest, 3 times more mutants than Major and 2 times more mutants than muJava. If we look at the per-mutant time, WODEL-TEST/Java is the fastest.

Altogether, looking at the results of this experiment, we can conclude that the Java MT tool created with WODEL-TEST is comparable to existing MT tools regarding efficacy and efficiency of the MT process.

5.1.4 Discussion and threats to validity

Section 5.1.1 shows that WODEL-TEST/Java offers functionality comparable to existing MT tools for Java, and moreover, it offers a rich set of interactive views with the results of the MT process, makes the mutated code available for inspection, enables configuration of the unit testing technology and includes mechanisms to specify equivalence criteria for mutants. In particular, we believe that WODEL-TEST/Java is a better option for MT of Java programs than the other analysed MT tools in four scenarios: when the tester needs to have access to the source code of the mutants, wants to be able to reason on which mutants reduce the mutation score and why, wants to experiment with new mutation operators or needs to implement purpose-specific mutation operators, for instance, for security testing or other non-functional properties.

As an estimation of development effort, the existing MT tools developed manually required two orders of magnitude more code than the specification of a similar environment using WODEL-TEST. However, we have to acknowledge that a significant amount of code in those approaches may deal with advanced optimizations like reducing the execution time, which WODEL-TEST/Java currently neglects. Moreover, we were able to reuse the Java meta-model and the M2T/T2M transformations provided by MoDISCO, which might not be available when building a MT for other programming language.

One of the salient features of WODEL-TEST is its extensibility, as its DSL WODEL permits defining new mutation

⁵ <https://github.com/soursop/functional-matrix-operator>.

Table 3 Application of Java MT tools to the functional-matrix-operator project

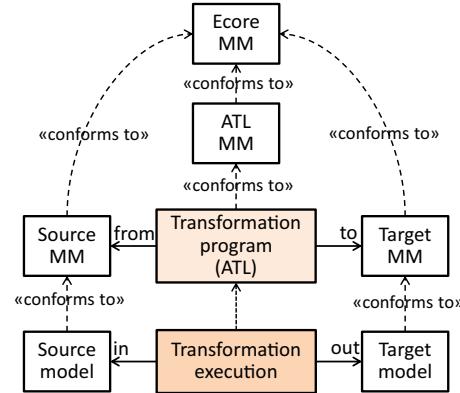
Tool	Mutants (killed/alive)	Mutation score (%)	Running time	Per-mutant time (s)
Major	1638 (331/864)	20.21	13 h 42 m 56 s	30.14
PITest	918 (321/597)	34.97	1 h 11 m 20 s	4.66
LittleDarwin	439 (130/309)	29.61	3 h 30 m 39 s	28.79
muJava	2589 (557/2032)	21.51	5 h 30 m 10 s	7.65
WODEL- TEST/Java	4756 (985/3771)	20.71	3 h 24 m 23 s	2.58

operators. PITest is also extensible, but requires programming to define new operators. Major is also extensible by the use of a DSL, but this is less expressive than WODEL due to two factors. First, Major works at the bytecode level while WODEL works at the model level. Second, Offutt et al. showed that a reduced set of mutation operators is enough to produce similarly strong test cases [75], and this set has not changed much in subsequent works [79]. However, the new features added to existing programming languages (e.g. lambdas) and the new languages appeared recently do require the ability to develop new mutation operators.

As Table 3 shows, the MT process of WODEL- TEST/Java is as effective as the MT process performed by the other tools (comparable mutation scores, almost the same as Major). Moreover, its per-mutant execution time is similar to that obtained for PITest, which works at the bytecode level, and much lower than Major and LittleDarwin. However, WODEL- TEST/Java generates more mutants than the other tools. This is because we cover all mutation operators of the other tools, but also because the other tools apply optimizations to reduce the number of generated mutants and be more efficient. Such optimizations are important because a high number of mutants can be difficult to manage by the tester and slows down the MT process. In this respect, WODEL- TEST/Java supports the filtering of equivalent mutants to optimize the MT process, as well as applying just a subset of the available mutation operators. However, WODEL- TEST/Java lacks other optimization mechanisms which the other tools do incorporate. In the future, we plan to improve WODEL- TEST with an extension point to include such optimization techniques, like prioritization [51], random selection [39], higher-order mutants [48] or reduction of mutants by removing subsumed ones [77].

Altogether, we can answer RQ1 positively: WODEL- TEST can generate MT tools comparable to existing tools created manually. The generated tools have advantages in terms of extensibility and customizability, offering a rich set of interactive views with information of the MT process.

Threats to validity. We have to mention two threats to the validity of our results. First, we have analysed five MT tools for Java, but there may be others with more sophisticated functionality or better efficiency. To mitigate this risk, the tools used in the evaluation are well-known, state-of-the-art,

**Fig. 10** Working scheme of ATL transformations

reputed tools in the MT community, which are representative of the typical MT functionality and efficiency. Second, our evaluation only considers one project. In the future, we plan to replicate the experiments with further projects to increase the confidence of our results.

5.2 Case study: building a MT tool for ATL

This section presents a case study on the development of a MT tool for the Atlas transformation language (ATL) [50]. ATL is a widely used model transformation language to define model-to-model transformations. This kind of transformations translates a model conforming to a source meta-model into another model conforming to a target meta-model (see Fig. 10).

We use ATL as a case study because despite its popularity in MDE, it is error-prone as it is dynamically typed [80]. Moreover, testing transformations is difficult [11] because it involves the creation of input test models and the assessment of the output models. Input test models can be created either manually or automatically using diverse methods such as randomly, covering the input meta-model, or covering the transformation [42, 72, 83]. Hence, MT can be useful to evaluate the quality of this input test set. However, there are few publicly available tools for MT of ATL programs. A possible reason is the cost and complexity of development, because mutation operators need to consider not only ATL programs but also query the source and target meta-models. The pur-

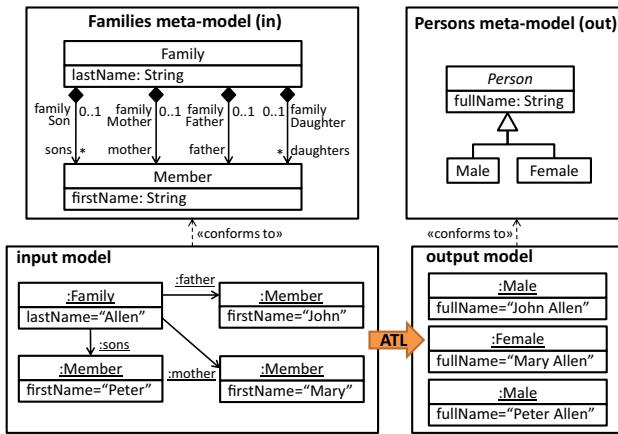


Fig. 11 ATL transformation example

pose of this section is to show how WODEL-TEST facilitates the creation of a MT tool for ATL, helping to answer RQ2.

In the remainder of this section, first, Sect. 5.2.1 introduces the basics of ATL. Then, Sect. 5.2.2 describes how to create the MT tool with WODEL-TEST. Section 5.2.3 assesses the creation process of the MT tool. Finally, Sect. 5.2.4 discusses the results and identifies possible threats.

5.2.1 The ATL language

ATL allows defining transformation programs using a rule-based textual notation. ATL programs conform to the ATL meta-model, and they are also typed with respect to the source and target meta-models of the transformation. As Fig. 10 shows, meta-models conform to Ecore, which is a metamodeling language that follows the Meta-Object Facility standard of the OMG [70].

Figure 11 shows the effect of executing an ATL program which transforms a Families model into a Persons model. The program simply creates a Male object for each Member object playing a father or sons role within a Family; and a Female object for each Member object playing mother or daughters roles. The fullName of the created objects is the concatenation of the firstName and lastName of the original objects.

ATL programs consist of declarative rules stating how a pattern of objects in the source model is to be translated into a pattern of objects in the target model. As an example, Listing 4 shows an ATL program that performs the transformation exemplified by Fig. 11.

Lines 2–4 of the listing indicate the input and output meta-models of the transformation. The program declares two rules in lines 25–30 (Member2Male) and 32–37 (Member2Female). Rule Member2Male is executed for each male Member object. Rules are generally made of a source pattern (line 26), an optional filter constraining the objects in the pattern

```

1 module Families2Persons;
2   --@path Families=/Families2Persons/Families.ecore
3   --@path Persons=/Families2Persons/Persons.ecore
4   create OUT:Persons from IN:Families;
5
6   -- Returns whether a member is a female
7   helper context Families!Member def: isFemale(): Boolean =
8     if not self.familyMother.oclsUndefined() then true
9     else if not self.familyDaughter.oclsUndefined() then true
10    else false endif
11  endif;
12  -- Returns the family name of a member
13  helper context Families!Member def: familyName: String =
14    if not self.familyFather.oclsUndefined() then
15      self.familyFather.lastName
16    else if not self.familyMother.oclsUndefined() then
17      self.familyMother.lastName
18    else
19      if not self.familySon.oclsUndefined() then
20        self.familySon.lastName
21      else self.familyDaughter.lastName endif
22    endif;
23  -- Creates a male from a family member that is a male
24  rule Member2Male {
25    from s:Families!Member (not s.isFemale())
26    to t:Persons!Male (
27      fullName <- s.firstName + '' + s.familyName
28    )
29  }
30  -- Creates a female from a family member that is a female
31  rule Member2Female {
32    from s:Families!Member (s.isFemale())
33    to t:Persons!Female (
34      fullName <- s.firstName + '' + s.familyName
35    )
36  }
37 }
```

Listing 4 ATL program example

(line 26, not s.isFemale()), an output pattern with the objects to be created (line 27), and attribute initializers called bindings (line 28). Rule Member2Female is similar, but matches female Member objects and creates Female objects.

The listing also declares two helpers in lines 7–11 (isFemale) and 13–23 (familyName). Helpers are auxiliary functions defined in the context of a meta-model class, similar to methods and derived attributes in object-oriented programming languages. The body of helpers is written using OCL. In the listing, helper isFemale is used in the rule filters, and helper familyName is used in the bindings to initialize the attribute fullName of the created objects.

ATL contains other features, like lazy and matched rules, and the working scheme of bindings is more complex than presented. However, this is enough to illustrate the main aspects of the language. We refer the reader to [50] for more details.

5.2.2 Using WODEL-TEST to build a MT tool for ATL

ATL is implemented based on EMF. Its abstract syntax is defined by a meta-model, of which Fig. 12 is a very small excerpt. The meta-model fragment shows that the InPattern of rules declares one or more VariableDeclarations with a variable name and a type. This type must be an OclModelEle-

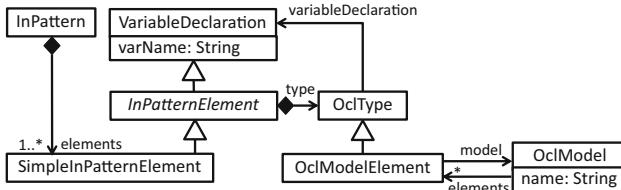


Fig. 12 ATL meta-model excerpt

ment from an OclModel, or in other words, a class from the input meta-model.

The ATL distribution provides a T2M transformation to parse textual ATL programs into instances of the ATL meta-model, and a M2T transformation to generate ATL code from an in-memory ATL model. This way, the language support required by WODEL- TEST just reuses these two transformations.

With respect to execution support, WODEL- TEST requires defining how to compile the ATL programs and how to execute the test cases. ATL programs can be programmatically compiled into bytecode by the ATL virtual machine. Since there is no standard unit testing framework for ATL, we use input models of the transformation as test cases. Using this approach, there is no need to specify an explicit test oracle. Instead, for each input model, we execute the original and mutant transformations with the model and compare the results: if the outputs are different, then the test failed, otherwise, it succeeded.

The last component of a MT tool specification concerns the mutation support. In this case, we have used WODEL to define the 18 mutation operators proposed in [89]. These operators, shown in Table 4, allow the creation, deletion and modification of rules, input pattern elements, output pattern elements, rule filters and bindings. Since the operators were originally specified as ATL transformations [89], the table also compares the LOC of their encoding using ATL and WODEL (in the former case, only when available in [89]).

As an example, Listing 5 shows the WODEL encoding of the mutation operator that creates an input pattern element (“Appendix B” contains the implementation of all other operators). The operator needs to create a variable declaration typed by an existing input meta-model class. For this purpose, lines 3–5 declare an auxiliary resource named `input` which points to the location of the input meta-model of the transformation, and conforms to the Ecore meta-model. Then, the program selects one class from the input meta-model (line 9) and creates an input pattern element (line 12) typed by the selected class (line 13).

WODEL ensures that all generated mutants will conform to the ATL meta-model. In addition, it provides an extension point to add other criteria that any mutant should fulfill. In this case, we have implemented this extension point to statically analyse the mutant and discard it if it has typing errors

```

1 generate 2 mutants in "out/" from "model/"
2 metamodel "mm/ATLecore"
3 with resources from {
4   input="mm/in"
5   metamodel="mm/Ecore.ecore"
6 }
7 with blocks {
8   cipe "Create in pattern element" {
9     cl = select one EClass from input resources
10    p = select one InPattern
11    mod = select one OclModel in p->elements->type->model
12    ipe = create SimpleInPatternElement in p->elements with {varName =
13      random-string(2, 4)}
14    elem = create OclModelElement in ipe->type with {name = cl.name,
15      variableDeclaration = ipe}
16    modify mod with {elements += elem}
17  }
18 }

```

Listing 5 WODEL program implementing the mutation operator *in pattern element addition*

that may lead to runtime errors. To perform the analysis, we programmatically use a static analyser for ATL called anATLyzer [80]. This ensures that the mutants will run without producing runtime exceptions.

Finally, we have defined a domain-specific equivalence criterion for ATL programs based on comparing the bytecode of their compilation, which is serialized in XML. Although the standard ATL virtual machine does not optimize the bytecode, the alternative EMFTVM does [93].

Figure 13 shows the generated MT tool for ATL, which has a similar structure to the one developed for Java. The ATL editor, with label 1, contains the code of a mutant transformation. In particular, the String concatenation in line 35 of Listing 4 has been replaced by a String literal in line 62 of Fig. 13. Label 2 corresponds to the package explorer, which contains a project with the input models used as test cases, and an ATL project with a folder for each generated mutant. Panel 3 shows the mutants view where the live mutants are depicted in red. Panel 4 shows the global view of MT results.

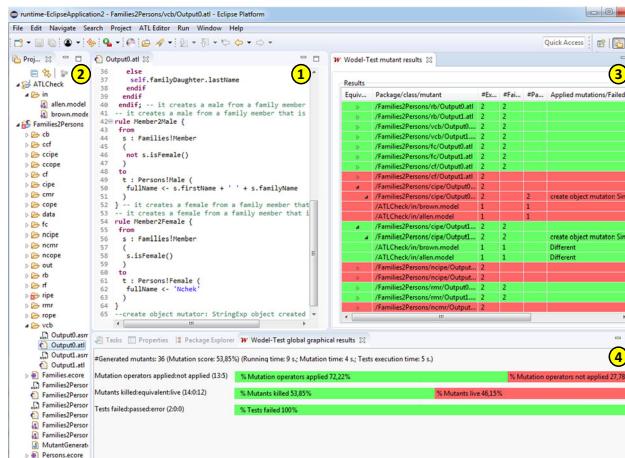
5.2.3 Assessing the creation process of the MT tool for ATL

Next, we assess the use of WODEL- TEST to construct a MT tool for ATL, in comparison with the technique proposed in [89] in which we based our suite of mutation operators. We analyse three aspects: (i) the definition of mutation operators, (ii) the execution control of the mutation operators and (iii) the specification and provision of other functionalities useful to MT.

Definition of mutation operators. Since ATL programs are internally represented as models, an alternative to WODEL to define mutation operators is the use of a general-purpose model transformation language. As we mentioned in the previous section, in [89], the authors propose 18 mutation operators for ATL, and implement some of them using ATL (specifically, using so-called higher-order ATL transformations).

Table 4 Mutation operators for ATL

Concept	Mutation operator	LOC in Wodel	LOC in ATL
Matched rule	Addition Deletion Name change	1 1 1	— — —
In pattern element	Addition Deletion Class change Name change	6 1 4 1	14 — — —
Filter	Addition Deletion Condition change	10 1 4	— — —
Out pattern element	Addition Deletion Class change Name change	6 1 4 1	— 6 — —
Binding	Addition Deletion Value change Feature change	6 1 2 6	— 3 — —

**Fig. 13** MT tool for ATL created with WODEL- TEST

Listing 6 shows the ATL program that implements the mutation operator *in pattern element addition*. The program creates an input pattern element with a non-existing type named “Complete_IPE” (line 22). Then, another ATL program that we show in Listing 7 is subsequently executed in refining mode⁶ to modify the mutant transformation generated by the first program. This second program retrieves a random class from the input meta-model and replaces the “Complete_IPE” literal by the class name. In contrast, the WODEL encoding of the same mutation operator is able to read the class name from the input meta-model, i.e. there is no need

⁶ In refining mode, the input model of a transformation is changed in-place and produced as output.

```

1 module AddInPatternElement_FirstRule;
2
3 create OUT : ATL refining IN : ATL;
4
5 -- Sequence for giving new variable names to new
6 -- pattern elements that are created
7 helper def : varNames : Sequence(String) =
8   Sequence('a','aa','b','bb','c','cc','d','dd','e','ee','f','ff','...');
9
10 rule AddInPatternElement {
11   from s : ATL!InPattern (ATL!Rule.allInstances()→first() = s."rule") -- Add
12     SYPE only in first rule
13   to t : ATL!InPattern(
14     elements <- s.elements → append(newIPE)
15   ),
16   newIPE : ATL!SimpleInPatternElement (
17     -- Give a variable name that no PatternElement has
18     varName <- thisModule.varNames→any(n |
19       ATL!PatternElement.allInstances()→collect(pe|pe.varName)→
20         excludes(n)),
21     type <- newOCLType
22   ),
23   -- The type is composed of a model and a name: model!name
24   newOCLType : ATL!OclModelElement(
25     model <- s.elements→first().type.model,
26     name <- 'Complete_IPE'
27   )
28 }
```

Listing 6 ATL program implementing the operator *in pattern element addition* [89]

to define two different programs for this purpose (see Listing 5). The way to assign a name to the new pattern element also differs, as the ATL program relies on a pre-constructed name set (lines 7–8 in Listing 6), while the WODEL program generates a random string (line 12 in Listing 5).

Mutation execution control. The ATL mutation operator in Listing 6 hard-codes the location where the operator is to be applied, in this case, in the first rule of the seed ATL

```

1 module SecondOrderHOT;
2
3 create OUT : ATL
4     refining IN : ATL, IN_MM : IN_MM, OUT_MM : OUT_MM;
5
6 helper def : random() : Real =
7     "#native""java::util::Random".newInstance().nextDouble();
8
9 --- A StringExp is one of the types that can conform to the value part
10 --- of a Binding. Since the generic mutation transformation
11 --- adds 'Complete_IPE' in the value part, a StringExp is created,
12 --- whose stringSymbol is 'Complete_IPE'
13 rule CompleteInMMNames {
14     from s : ATLStringExp (s.stringSymbol = 'Complete_IPE')
15     using {
16         classes : Sequence(IN_MM!EClass) =
17             IN_MM!EClass.allInstancesFrom('IN_MM')->select(c |
18                 not c.abstract);
19     }
20     to t : ATLStringExp(
21         --- A random class from the input model
22         stringSymbol <- classes->at((thisModule.random()*
23                                         classes->size()),floor()+1).name
24     )
25     do {
26         classes->at((thisModule.random()*classes->size(),
27                         floor())+1).name;
28     }
29 }
```

Listing 7 Refining transformation to replace the literal “Complete_IPE” by an input class name [89]

program (line 11). Instead, WODEL supports a more declarative style to control the mutation execution process, being possible to customize the minimum and maximum number of applications of the operator, or execute it exhaustively at every possible location.

The tool created with WODEL- TEST permits selecting the mutation operators to apply in the MT process. This fine-grained control would be challenging to emulate using only ATL due to its execution semantics. This is so as ATL rules are applied exactly once at every possible match, and a same object cannot be matched by two different rules because this would raise a runtime error. As a consequence, mutation operators specified with ATL cannot be grouped in the same transformation program, but need to be scattered in several programs orchestrated either manually or by means of a dedicated controller Java program. In the prototype tool provided by [89], this amounts to around 520 LOC, while implementing the LanguageServiceProvider support for ATL in WODEL- TEST has around 350 LOC.

Functionality of MT tool. Table 5 compares the MT functionality provided by both proposals. The MT tool generated with WODEL- TEST provides the typical functionalities required for an integral MT environment out-of-the-box, including the calculation of the mutation score, covered operators, and other useful metrics. The prototype tool in [89] allows the generation of mutants of ATL programs but gives no further support for MT. Hence, there is no mechanism to detect equivalent mutants, perform the MT process (i.e. execution of mutants against the test cases) and report the

results (not even the mutation score). Hence, to obtain a functional MT tool, these functionalities should be programmed by hand, which is not required using WODEL- TEST.

With respect to development effort, the WODEL- TEST specification of the MT tool for ATL consists of 327 LOC, which is comparable to the effort dedicated to specify the MT tool for Java, which has 305 LOC. We cannot compare with the LOC in Troya’s approach [89] because there is no tool providing support for the MT process.

5.2.4 Discussion and threats to validity

Even though ATL was constructed using MDE, implementing their mutation operators using a DSL has advantages with respect to using a general-purpose transformation language. For example, WODEL has facilities for object cloning (used in the *matched rule addition* operator in “Appendix B”), retyping, assignment of random attribute values (used in Listing 5), automated resolution of suitable containers for new objects, checking of mutant well-formedness, and access to auxiliary resources (e.g. the input meta-model in Listing 5). It also provides mutation execution control facilities that would be difficult to emulate using a general-purpose model transformation language. For instance, WODEL permits the exhaustive application of mutation operators on every possible location, while using ATL, this must be emulated by the use of different transformations. Moreover, our approach permits specifying mutant equivalence criteria (e.g. XML-based comparison of the ATL “bytecode”).

Another advantage of WODEL- TEST is that the synthesized MT tools provide rich functionality, which otherwise needs to be programmed manually. Likewise, if there is the need to change the functionality of a MT tool created with WODEL- TEST (e.g. to add mutation operators that consider new target language features due to new language versions, or modify the existing mutant equivalence criteria), these changes can be performed at the MT tool specification level, so that the MT tool can be regenerated from this specification. This simplifies the maintenance of our MT tools, as their specification is higher-level and hence simpler to modify than the equivalent code.

Last but not least, our MT tools are Eclipse plug-ins, and so, they can be easily integrated with other tools through extension points (the non-intrusive mechanism provided by Eclipse to permit contributing new functionality to existing plug-ins). In particular, WODEL- TEST defines extension points to let developers integrate other tools in almost every step of the MT process: validation of mutant correctness, detection of equivalent mutants, mutant compilation, etc. As an example, the MT tool developed for ATL integrates a static analyser to check the correctness of the generated mutants and discard them if they may throw runtime errors.

Table 5 Comparison of ATL MT tools

	Troya et al. [89]	WODEL- TEST/ATL
Last update	2015	2019
<i>Input to MT process</i>		
UI	Command line	Eclipse plug-in
Scope	ATL program	ATL program
Test suite	None	Input model (customizable)
<i>Mutant generation process</i>		
Implemented operators	3/18	18/18
Mutants per operator	1	All possible
Op. extensibility	No	Yes (DSL)
Mutated artefact	Model	Model
Mutant source	Yes	Yes
Mutant filter	No	No
Equivalent mutant detection	No	Yes (TCE)
<i>Reporting</i>		
Report type	None	Interactive views
No. of mutants	✗	✓
Mutation score	✗	✓
Killed/live mutants	✗	Number, list
Covered operators	✗	Number, list
Mutants by ATL program	✗	✓
Tests by mutant	✗	✓
Mutants by test	✗	✓

Altogether, based on the LOC required to build MT tools either manually or using WODEL- TEST, and based on the functionality that WODEL and WODEL- TEST provide out-of-the-box, we can answer RQ2 by stating that using WODEL- TEST is more effective than building a MT tool from scratch using a programming language or a combination of programming and transformation languages. Moreover, the MT tools built with WODEL- TEST also have advantages in terms of maintainability and integrability with external tools.

Threats to validity. In order to define the MT tools for Java and ATL using WODEL- TEST, we reused existing meta-models, M2T and T2M transformations. If these artefacts were not available, building the MT tools would have required more effort. However, as Sect. 3 mentioned, some MDE re-engineering projects have made available these artefacts for programming languages, while they are commonly available for modelling languages.

6 Related research

MT was initially applied to source code, and hence, catalogues of mutation operators exist for programming languages such as Ada [76], C [2,47], C++ [19,59], Fortran [55], Java [12,54,65] or SQL [91]. In addition, MT has also been applied to other artefacts such as formal specifica-

tions (e.g. finite state machines [26,46], statecharts [28,88], Petri nets [27], CSP [86]) or web services [25,63]. Consequently, a wide range of tools have been developed to support mutation analysis for different programming languages. Initially, these tools were designed to deal with the MT techniques proposed in the academic environment [18,20], but since the early 2000s, many tools aim to make MT practical [3,47,58,66,81,90].

The relevance of MT is demonstrated by the many kinds of software systems and application domains where it has been applied, including web applications [69], safety-critical software [10], function block diagrams [84], cloud and HPC environments [16], Android apps [22], GUIs [5], IoT protocols [9] and memory faults [94]. We believe that our approach may simplify the development of MT tools for these and other application domains.

In the remainder of this section, we review works related to the systematic construction of MT tools, the extensibility mechanisms devised for MT tools and the use of mutation within MDE.

Systematic construction of MT tools. To the best of our knowledge, there are not many proposals to automate the systematic construction of MT tools. One recent exception is [8], where the authors propose using MDE to support MT of Java. Just like WODEL- TEST, they represent Java programs as models conformant to the MoDISCO meta-model and reuse

the MODISCO M2T and T2M transformations. However, they encode mutation operators using the QVT-o model transformation language. That approach is only incipient as it does not support the execution of the test cases, while WODEL-TEST is a fully developed approach that automates the whole MT process. While that work targets Java, WODEL-TEST is generic and applicable to arbitrary languages. Finally, that work only defines 4 mutation operators, while WODEL-TEST provides 77 mutation operators that can be extended or modified using a DSL tailored to model mutation.

Extensibility of MT tools. Some MT tools permit some degree of customization of the mutation operators. For example, Sect. 5.1.2 showed that Major [51] permits configuring the replacement lists of mutation operators like Arithmetic Operator Replacement (AOR). MuCheck [61] is a MT framework for Haskell that offers an internal DSL embedded in Haskell to develop new mutation operators or alter existing ones. However, the expressiveness of this DSL is limited to the modification of values with a mutant replacement. SMutant [31] is a mutation framework for Smalltalk, which permits the definition of new mutation operators programmed in Smalltalk. Overall, our contribution with respect to this branch of works is the availability of an external DSL specifically designed to define mutation operators and applicable to arbitrary languages. Our DSL does not only support specifying new operators, but also controlling their execution policies and ensuring their validity with regards to OCL constraints.

While the previous works are language-dependent, the aim of *Universalmutator* [41] is being language-independent. Its approach is based on the replacement of simple regular expressions, which the tester can define. It works on unparsed text files so it does not need a parser, which makes it language-independent. However, as the authors mention, the use of unparsed text presents limitations on the kind of mutations the tool is able to express. In contrast, WODEL can express complex mutations and specify OCL validity criteria for the generated mutants, as it works at the model level.

Another approach to decouple mutation from specific languages is its application at the level of bytecode—as this allows defining mutation operators for languages targeting the Java Virtual Machine [51]—or its application at the intermediate representation level (IR) of compiler architectures like LLVM [23]. Mull [23] is a MT tool that works over IR, so it can be applied to C, C++ and any other language that compiles to LLVM IR, such as Rust, Swift and Objective-C. However, low-level code like IR is optimized for execution but not for analysis, limiting the mutation operators that can be expressed. For example, Mull supports 4 types of simple mutations like *replace arithmetical operator*, *negate condition*, *remove void function*, *replace call* and *replace scalar*. Moreover, mutations that are possible at the IR level may not be applicable or relevant at the source-level (so called

junk mutations [23]). Instead, our approach to language independence is based on expressing the language syntax as a meta-model. This allows creating more complex mutation operators at the price of a potentially lower efficiency.

MT within MDE. Several authors have applied MT in the area of MDE, especially to model transformations. For example, in [71], a collection of generic mutation operators is defined based on the main activities involved in a model transformation (navigation, filtering, output model creation and input model modification). In [42], the authors provide a Java framework to simplify the MT of ATL transformations. The framework incorporates many mutation operators, included the ones used in our evaluation. However, it has to be used programmatically and provides less functionality than the MT tool developed with WODEL-TEST (e.g. there is no graphical user interface, detailed MT metrics or automatic/manual detection of equivalent mutants).

Other works propose systematic approaches for generating mutation operators either for a specific language [53, 89] or for different ones [4]. Mutation has been applied in the context of UML [3, 40]. For example, in [3, 58], the authors generate test cases for UML state machines. These test cases are able to kill live mutants and can be automatically generated. In [40], the authors propose a set of mutation operators for UML class diagrams. WODEL-TEST could have helped in the design of the operators, and the automation of the MT process.

Mutation has been widely employed to evaluate strategies to generate input models for model transformation testing. For example, in [72], the authors compare techniques based on footprints, input domain coverage and random generation using mutation testing. In a same vein, [42] uses different mutation operator sets to evaluate input model generation techniques based on meta-model coverage, random generation and transformation path coverage. In [83], the authors use mutation to evaluate model generation based on partition with respect to a random strategy. WODEL-TEST can help in automating these evaluations by enabling the creation of mutations for this purpose.

Finally, mutation operators have also been used to synthesize input models for transformations [7, 82]. Interestingly, in [82] the operators are graph grammar rules automatically derived from the meta-model being transformed. We believe that WODEL-TEST is particularly useful for MT within MDE, as MDE artefacts are typically defined by meta-models to which our approach is readily applicable. We expect that our work can bring MT closer to the MDE community.

In summary, our approach is novel in that it is the first fully working framework that automates the construction of MT tools for arbitrary languages, hence reducing the effort to create this kind of tools. Instead, most existing MT tools have been developed by hand, which allows their specialization but requires a high development cost. Moreover, we provide

a DSL to build (model) mutation operators which, compared to the mutation languages provided by manually-coded MT tools, is more expressive, can be applied to arbitrary languages, supports the specification of validity criteria for the created mutants and can be extended with mechanisms for the detection of equivalent or duplicate mutants. While MT has been applied to different MDE artefacts, no mechanism for the systematic construction of MT tools for such artefacts has been proposed. Our work aims to cover this gap.

7 Conclusions and future work

In this paper, we have presented a model-based approach to automate the generation of MT tools for arbitrary languages. The approach is based on a DSL tailored to define mutation operators in a language-independent way. We have shown the benefits of our approach defining MT tools for Java and ATL.

Our mutation operators can be defined for a particular language (e.g. Java, ATL) and cannot be reused for other languages. This is so as each language has its own meta-model and semantics, and the mutation operators need to take both into account. However, we foresee extending our approach to allow the reuse of generic, non-domain-specific mutation operators across similar languages (e.g. operators defined for a C++ meta-model could be reused with a Java meta-model). To this aim, we may apply techniques like those proposed in [14]. Likewise, we also intend to extend the WODEL DSL to support the definition of functions that can be reused across mutation operators.

Currently, we are working on improving the tool efficiency to cope with the MT of larger programs. In the future, we would like to perform a user study to analyse to which extent users find our MT tools usable. We are also planning to provide WODEL-TEST with extension points to support mutant filtering (e.g. based on statement coverage), reduction techniques (e.g. based on sampling), control of redundant mutants (e.g. dynamic approximation of the disjoint mutation score [60]) or the possibility to generate test cases from the models [95]. For the latter, we could take each live mutant and automatically generate a test case hitting the mutated line, extending in this way the initial test suite. However, this analysis would be dependent on the programming language (e.g. ATL [42,80]). Another option could be to use mutation to produce variations of the initial test suite, for which we would need to provide a meta-model of the input to the test suite. We are also planning to apply our approach to languages of other paradigms, like functional [61] or dataflow-based [38]. Finally, another line of research that is worth investigating concerns the automated synthesis of M2T and T2M transformations out of the meta-model of the targeted language and sample instances of textual programs.

Acknowledgements This work has been partially funded by the Spanish Ministry of Science (projects RTI2018-093608-B-C31, RTI2018-095255-B-I00, TIN2015-65845-C3-1-R) and the R&D programme of the Madrid Region (project S2018/TCS-4314).

A Java mutation operators

This “Appendix” contains the definition of the Java mutation operators included in [17,51,67,79,81] using WODEL.

Java mutation	Wodel code
Replaces an arithmetic operator by ‘+’	modify one InfixExpression where { operator in [‘-’, ‘*’, ‘/’, ‘%’] } with { operator = ‘+’ }
Replaces an arithmetic operator by ‘-’	modify one InfixExpression where { operator in [‘+’, ‘*’, ‘/’, ‘%’] and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral } with { operator = ‘-’ }
Replaces an arithmetic operator by ‘*’	modify one InfixExpression where { operator in [‘+’, ‘-’, ‘/’, ‘%’] and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral } with { operator = ‘*’ }
Replaces an arithmetic operator by ‘/’	modify one InfixExpression where { operator in [‘+’, ‘-’, ‘*’, ‘%’] and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral } with { operator = ‘/’ }
Replaces an arithmetic operator by ‘%’	modify one InfixExpression where { operator in [‘+’, ‘-’, ‘*’, ‘/’] and leftOperand not typed StringLiteral and rightOperand not typed StringLiteral } with { operator = ‘%’ }
Replaces a postfix ‘++’ operator by ‘--’	modify one PostfixExpression where { operator = ‘++’ } with { operator = ‘--’ }
Replaces a postfix ‘--’ operator by ‘++’	modify one PostfixExpression where { operator = ‘--’ } with { operator = ‘++’ }
Replaces a prefix ‘++’ operator by ‘--’	modify one PrefixExpression where { operator = ‘++’ } with { operator = ‘--’ }
Replaces a prefix ‘--’ operator by ‘++’	modify one PrefixExpression where { operator = ‘--’ } with { operator = ‘++’ }
Replaces a prefix ‘+’ operator by ‘-’	modify one PrefixExpression where { operator = ‘+’ } with { operator = ‘-’ }
Replaces a prefix ‘-’ operator by ‘+’	modify one PrefixExpression where { operator = ‘-’ } with { operator = ‘+’ }
Replaces a prefix ‘+’ or ‘-’ operator by ‘--’	modify one PrefixExpression where { operator in [‘+’, ‘-’] and operand not typed NumberLiteral } with { operator = ‘--’ }
Replaces a prefix ‘+’ or ‘-’ operator by ‘++’	modify one PrefixExpression where { operator in [‘+’, ‘-’] and operand not typed NumberLiteral } with { operator = ‘++’ }

Continued on next column...

...continued from previous column

Java mutation	Wodel code
Replaces a prefix ‘++’ or ‘--’ operator by ‘+’	<code>modify one PrefixExpression where { operator in ['++', '--'] } with { operator = '+' }</code>
Replaces a relational operator by ‘>’	<code>modify one InfixExpression where { operator in ['>', '<', '<=', '==', '!='] } with { operator = '>' }</code>
Replaces a relational operator by ‘>=’	<code>modify one InfixExpression where { operator in ['>', '<', '<=', '==', '!='] } with { operator = '>=' }</code>
Replaces a relational operator by ‘<’	<code>modify one InfixExpression where { operator in ['>', '>=', '<=', '==', '!='] } with { operator = '<' }</code>
Replaces a relational operator by ‘<=’	<code>modify one InfixExpression where { operator in ['>', '>=', '<', '==', '!='] } with { operator = '<=' }</code>
Replaces a relational operator by ‘==’	<code>modify one InfixExpression where { operator in ['>', '>=', '<', '<=', '!='] } with { operator = '==' }</code>
Replaces a relational operator by ‘!=’	<code>modify one InfixExpression where { operator in ['>', '>=', '<', '<=', '=='] } with { operator = '!='</code>
Replaces a conditional operator by ‘&&’	<code>modify one InfixExpression where { operator in [' ', '^'] } with { operator = '&&' }</code>
Replaces a conditional operator by ‘ ’	<code>modify one InfixExpression where { operator in ['&&', '^'] } with { operator = ' ' }</code>
Replaces a conditional operator by ‘^’	<code>modify one InfixExpression where { operator in ['&&', ' '] } with { operator = '^' }</code>
Replaces a logic operator by ‘&’	<code>modify one InfixExpression where { operator in [' ', '^'] } with { operator = '&' }</code>
Replaces a logic operator by ‘ ’	<code>modify one InfixExpression where { operator in ['&', '^'] } with { operator = ' '</code>
Replaces a logic operator by ‘^’	<code>modify one InfixExpression where { operator in ['&', ' '] } with { operator = '^' }</code>
Replaces a shift operator by ‘>>’	<code>modify one InfixExpression where { operator in ['>>', '<<'] } with { operator = '>>' }</code>
Replaces a shift operator by ‘>>>’	<code>modify one InfixExpression where { operator in ['>>', '>>>'] } with { operator = '>>>' }</code>
Replaces a shift operator by ‘<<<’	<code>modify one InfixExpression where { operator in ['>>', '>>>'] } with { operator = '<<<' }</code>
Replaces an assignment operator by ‘+=’	<code>modify one Assignment where { operator in ['-=', '*=', '/=', '%='] } with { operator = '+=' }</code>
Replaces an assignment operator by ‘*=’	<code>modify one Assignment where { operator in ['+=', '*=', '/=', '%='] } with { operator = '-=' }</code>
Replaces an assignment operator by ‘*=_’	<code>modify one Assignment where { operator in ['+=', '-=', '/=', '%='] } with { operator = '*=' }</code>
Replaces an assignment operator by ‘/=’	<code>modify one Assignment where { operator in ['+=', '-=', '/=', '%='] } with { operator = '/=' }</code>

Continued on next column...

...continued from previous column

Java mutation	Wodel code
Replaces an assignment operator by ‘%=>’	<code>modify one Assignment where { operator in ['+=', '-=', '*=', '/='] } with { operator = '%=>' }</code>
Replaces an assignment operator by ‘&=>’	<code>modify one Assignment where { operator in [' =', '^=', '>>=', '>>>=', '<<='] } with { operator = '&=>' }</code>
Replaces an assignment operator by ‘ =’	<code>modify one Assignment where { operator in [' =', '^=', '>>=', '>>>=', '<<='] } with { operator = ' =' }</code>
Replaces an assignment operator by ‘^=>’	<code>modify one Assignment where { operator in ['&=', ' ='] } with { operator = '^=>' }</code>
Replaces an assignment operator by ‘>>=>’	<code>modify one Assignment where { operator in ['>>=>', '<<='] } with { operator = '>>=>' }</code>
Replaces an assignment operator by ‘>>>=>’	<code>modify one Assignment where { operator in ['>>>=>', '<<='] } with { operator = '>>>=>' }</code>
Replaces an assignment operator by ‘<<=>’	<code>modify one Assignment where { operator in ['>>=>', '>>>=>'] } with { operator = '<<=>' }</code>
Replaces a number by a random number	<code>modify one NumberLiteral with { tokenValue = random-int-string(0, 9) }</code>
Deletes a call to a void method	<code>remove one MethodInvocation where { method->returnType->type is typed PrimitiveTypeVoid }</code>
Deletes a call to a non-void method	<code>remove one MethodInvocation where { method->returnType->type not typed PrimitiveTypeVoid }</code>
Replaces a call to a constructor by null	<code>a = select one Assignment where { rightHandSide is typed ClassInstanceCreation } create NullLiteral in a->rightHandSide</code>
Deletes an assignment	<code>remove one Assignment where { rightHandSide is typed NumberLiteral }</code>
Propagates an argument to a return statement	<code>s = select one ReturnStatement where { expression is typed MethodInvocation } m = select one MethodInvocation in s->expression p = select one SingleVariableAccess in m->arguments modify s with { expression = p }</code>
Propagates an argument to the right operand of an infix expression	<code>e = select one InfixExpression where { rightOperand is typed MethodInvocation } m = select one MethodInvocation in e->rightOperand p = select one SingleVariableAccess in m->arguments modify e with { rightOperand = p }</code>
Propagates an argument to the left operand of an infix expression	<code>e = select one InfixExpression where { leftOperand is typed MethodInvocation } m = select one MethodInvocation in e->leftOperand p = select one SingleVariableAccess in m->arguments modify e with { leftOperand = p }</code>
Propagates an argument to an assignment	<code>a = select one Assignment where { rightHandSide is typed MethodInvocation } m = select one MethodInvocation in a->rightHandSide p = select one SingleVariableAccess in m->arguments modify a with { rightHandSide = p }</code>

Continued on next column...

...continued from previous column		...continued from previous column	
Java mutation	Wodel code	Java mutation	Wodel code
Increments number in the right operand of an infix expression	e = select one InfixExpression where { rightOperand is typed NumberLiteral } n = select one NumberLiteral in e→rightOperand inc = deep clone n with { tokenValue = '1' } create InfixExpression in e→rightOperand with { leftOperand = n, operator = '+', rightOperand = inc }	Replaces a number in a return statement by -1	rt = select one ReturnStatement where { expression is typed NumberLiteral } modify rt with { expression.tokenValue = '1' } p = create PrefixExpression with { operator = '-', operand = rt→expression } modify rt with { expression = p }
Increments number in the left operand of an infix expression	e = select one InfixExpression where { leftOperand is typed NumberLiteral } n = select one NumberLiteral in e→leftOperand inc = deep clone n with { tokenValue = '1' } create InfixExpression in e→leftOperand with { leftOperand = n, operator = '+', rightOperand = inc }	Replaces a number in an assignment by -1	a = select one Assignment where { rightHandSide is typed NumberLiteral } modify a with { rightHandSide.tokenValue = '1' } p = create PrefixExpression with { operator = '-', operand = a→rightHandSide } modify a with { rightHandSide = p }
Increments number in a return statement	e = select one ReturnStatement where { expression is typed NumberLiteral } n = select one NumberLiteral in e→expression inc = deep clone n with { tokenValue = '1' } create InfixExpression in e→expression with { leftOperand = n, operator = '+', rightOperand = inc }	Deletes a unary conditional operator in an if statement	if = select one IfStatement where { expression is typed PrefixExpression } pre = select one PrefixExpression in if→expression where { operator = '!' } exp = select one Expression in pre→operand modify if with { expression = exp }
Increments number in an assignment	e = select one Assignment where { rightHandSide is typed NumberLiteral } n = select one NumberLiteral in e→rightHandSide inc = deep clone n with { tokenValue = '1' } create InfixExpression in e→rightHandSide with { leftOperand = n, operator = '+', rightOperand = inc }	Deletes a unary conditional operator in a return statement	rt = select one ReturnStatement where { expression is typed PrefixExpression } pre = select one PrefixExpression in rt→expression where { operator = '!' } exp = select one Expression in pre→operand modify rt with { expression = exp }
Replaces a boolean value by true	modify one BooleanLiteral where { value = false } with { value = true }	Deletes a unary conditional operator in the right operand of an infix expression	inf = select one InfixExpression where { rightOperand is typed PrefixExpression } pre = select one PrefixExpression in inf→rightOperand where { operator = '!' } exp = select one Expression in pre→operand modify inf with { rightOperand = exp }
Replaces a boolean value by false	modify one BooleanLiteral where { value = true } with { value = false }	Deletes a unary conditional operator in the left operand of an infix expression	inf = select one InfixExpression where { leftOperand is typed PrefixExpression } pre = select one PrefixExpression in inf→leftOperand where { operator = '!' } exp = select one Expression in pre→operand modify inf with { leftOperand = exp }
Replaces a number by 1	modify one NumberLiteral where { tokenValue <> '1' } with { tokenValue = '1' }	Deletes one statement	remove one Statement where { self not typed VariableDeclarationStatement }
Replaces a return statement by null	rt = select one ReturnStatement create NullLiteral in rt→expression	Adds negation in an if statement	if = select one IfStatement exp = select one InfixExpression in if→expression neg = create PrefixExpression in if→expression with { operator = '!' } par = create ParenthesizedExpression in neg→operand with { expression = exp }
Replaces a number by 0	modify one NumberLiteral where { tokenValue <> '0' } with { tokenValue = '0' }	Adds negation in a return statement	rt = select one ReturnStatement exp = select one InfixExpression in rt→expression neg = create PrefixExpression in rt→expression with { operator = '!' } par = create ParenthesizedExpression in neg→operand with { expression = exp }
Replaces a number in the left operand of an infix expression by -1	exp = select one InfixExpression where { leftOperand is typed NumberLiteral } modify exp with { leftOperand.tokenValue = '1' } p = create PrefixExpression with { operator = '-', operand = exp→leftOperand } modify exp with { leftOperand = p }	Adds negation in the left operand of an infix expression	e0 = select one InfixExpression e1 = select one InfixExpression in e0→leftOperand neg = create PrefixExpression in e0→leftOperand with { operator = '!' } par = create ParenthesizedExpression in neg→operand with { expression = e1 }
Replaces a number in the right operand of an infix expression by -1	exp = select one InfixExpression where { rightOperand is typed NumberLiteral } modify exp with { rightOperand.tokenValue = '1' } p = create PrefixExpression with { operator = '-', operand = exp→rightOperand } modify exp with { rightOperand = p }	Adds negation in the right operand of an infix expression	e0 = select one InfixExpression e1 = select one InfixExpression in e0→rightOperand neg = create PrefixExpression in e0→rightOperand with { operator = '!' } par = create ParenthesizedExpression in neg→operand with { expression = e1 }

Continued on next column...

Continued on next column...

...concluded from previous column		...continued from previous column	
Java mutation	Wodel code	ATL mutation	Wodel code
Deletes a conditional statement	<pre>bl = select one Block where { statements is typed IfStatement } if = select one IfStatement in bl->statements modify bl with { statements += if->thenStatement } remove if</pre>	In pattern elem. name change	<pre>modify one InPatternElement with { varName = random-string(4, 6) }</pre>
Decrements number in the left operand of an infix expression	<pre>e = select one InfixExpression where { leftOperand is typed NumberLiteral } n = select one NumberLiteral in e->leftOperand dec = deep clone n with { tokenValue = '1' } create InfixExpression in e->leftOperand with { leftOperand = n, operator = '_', rightOperand = dec }</pre>	Filter addition	<pre>p = select one InPattern where { filter is typed OperatorCallExp } oc = select one OperatorCallExp in p->filter feat = select one OclFeature ie = select one SimpleInPatternElement in p->elements conj = create OperatorCallExp in p->filter with { operationName = 'and' } call = create OperationCallExp in conj->^source with { operationName = feat.name } exp = create VariableExp in call->^source modify ie with { variableExp += exp } modify conj with { arguments += oc } modify p with { filter += conj }</pre>
Decrements number in the right operand of an infix expression	<pre>e = select one InfixExpression where { rightOperand is typed NumberLiteral } n = select one NumberLiteral in e->rightOperand dec = deep clone n with { tokenValue = '1' } create InfixExpression in e->rightOperand with { leftOperand = n, operator = '_', rightOperand = dec }</pre>	Filter deletion	<pre>remove one OclExpression where { container is typed InPattern }</pre>
Decrements number in a return statement	<pre>e = select one ReturnStatement where { expression is typed NumberLiteral } n = select one NumberLiteral in e->expression dec = deep clone n with { tokenValue = '1' } create InfixExpression in e->expression with { leftOperand = n, operator = '_', rightOperand = dec }</pre>	Filter condition change	<pre>p = select all InPattern where { filter <> null } oc = select one OperationCallExp in p->filter where { operationName <> ['not', 'and', 'or'] } feat = select one OclFeature where { name <> oc.operationName } modify oc with { operationName = feat.name }</pre>
Decrements number in an assignment	<pre>e = select one Assignment where { rightHandSide is typed NumberLiteral } n = select one NumberLiteral in e->rightHandSide dec = deep clone n with { tokenValue = '1' } create InfixExpression in e->rightHandSide with { leftOperand = n, operator = '_', rightOperand = dec }</pre>	Out pattern element addition	<pre>cl = select one EClass from output resources p = select one OutPattern mod = select one OclModel in p->elements->type->model oe = create SimpleOutPatternElement in p->elements with { varName = random-string(2, 4) } elem = create OclModelElement in oe->type with { name = cl.name, variableDeclaration = oe } modify mod with { elements += elem }</pre>
		Out pattern element deletion	<pre>remove one OutPatternElement</pre>
		Out pattern element class change	<pre>oe = select one SimpleOutPatternElement t = select one OclModelElement in oe->type cl = select one EClass from output resources where { name <> t.name } modify t with { name = cl.name }</pre>
		Out pattern element name change	<pre>modify one OutPatternElement with { varName = random-string(4, 6) }</pre>
		Binding addition	<pre>oe = select one SimpleOutPatternElement type = select one OclModelElement in oe->type cl = select one EClass from output resources where { name = type.name } att = select one EAttribute in cl->eAllAttributes b = create Binding in oe->bindings with { isAssignment = false, propertyName = att.name } create StringExp in b->value with { stringSymbol = random-string(4, 6) }</pre>
		Binding deletion	<pre>remove one Binding</pre>
		Binding value change	<pre>b = select one Binding where { value is typed OperatorCallExp } create StringExp in b->value with { stringSymbol = random-string(4, 6) }</pre>
		Binding feature change	<pre>oe = select one SimpleOutPatternElement type = select one OclModelElement in oe->type cl = select one EClass from output resources where { name = type.name } b = select one Binding in oe->bindings att = select one EAttribute in cl->eAllAttributes where { name <> b.propertyName } modify b with { propertyName = att.name }</pre>

Continued on next column...

ATL mutation	Wodel code
Matched rule addition	<pre>deep clone one MatchedRule with { name = random-string(4, 6) }</pre>
Matched rule deletion	<pre>remove one MatchedRule</pre>
Matched rule name change	<pre>modify one MatchedRule with { name = random-string(4, 6) }</pre>
In pattern element deletion	<pre>remove one InPatternElement</pre>
In pattern element class change	<pre>ie = select one SimpleInPatternElement t = select one OclModelElement in ie->type cl = select one EClass from input resources where { name <> t.name } modify t with { name = cl.name }</pre>

References

- Architecture driven modernization. <https://www.omg.org/adm/>. Accessed Oct. (2019)
- Agrawal, H., DeMillo, R.A., Hathaway, B., Hsu, W., Hsu, W., Krauser, E., Martin, R., Mathur, A.P., Spafford, E.: Design of mutant operators for the C programming language. Technical report, Purdue University (1989)
- Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: Killing strategies for model-based mutation testing. *Softw. Test. Verif. Reliab.* **25**(8), 716–748 (2015)
- Alhwikem, F., Paige, R.F., Rose, L., Alexander, R.: A systematic approach for designing mutation operators for MDE languages. In: Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA), pp. 54–59 (2016)
- Alsmadi, I.M.: Using mutation to enhance GUI testing coverage. *IEEE Softw.* **30**(1), 67–73 (2013)
- Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: International Conference on Software Engineering (ICSE). ACM, pp. 402–411 (2005)
- Aranega, V., Mottu, J.-M., Etien, A., Degueule, T., Baudry, B., Dekeyser, J.-L.: Towards an automation of the mutation analysis dedicated to model transformation. *Softw. Test. Verif. Reliab.* **25**(5–7), 653–683 (2014)
- Ariel González, G.B., Carlos, L.: Mutation testing for Java based on model-driven development. In: Simposio Latinoamericano de Ingeniería de Software (CLEI-SLISW) (in Spanish) (2018)
- Aziz, B.: Towards a mutation analysis of IoT protocols. *Inf. Softw. Technol.* **100**, 183–184 (2018)
- Baker, R., Habli, I.: An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Trans. Softw. Eng.* **39**(6), 787–805 (2013)
- Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.: Barriers to systematic model transformation testing. *Commun. ACM* **53**(6), 139–143 (2010)
- Bradbury, J.S., Cordy, J.R., Dingel, J.: Mutation operators for concurrent Java (J2SE 5.0). In: Workshop on Mutation Analysis (Mutation), pp. 83–92 (2006)
- Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, Second Edition. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers (2017)
- Bruel, J., Combemale, B., Guerra, E., Jézéquel, J., Kienzle, J., de Lara, J., Mussbacher, G., Syriani, E., Vangheluwe, H.: Comparing and classifying model transformation reuse approaches across metamodels. *Softw. Syst. Model.* **19**(2), 441–465 (2020)
- Brunelière, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. *Inf. Softw. Technol.* **56**(8), 1012–1032 (2014)
- Cañizares, P.C., Núñez, A., Merayo, M.G.: Mutomvo: mutation testing framework for simulated cloud and HPC environments. *J. Syst. Softw.* **143**, 187–207 (2018)
- Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: a practical mutation testing tool for Java (demo). In: International Symposium on Software Testing and Analysis (ISSTA). ACM, pp. 449–452 (2016). <http://pitest.org/>, <https://github.com/hcoles/pitest>
- Delamaro, M.E., Maldonado.: Proteum—a tool for the assessment of test adequacy for C programs. In: Conference on Performativity in Computing Systems, pp. 79–95 (1996)
- Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., Domínguez-Jiménez, J.J.: Assessment of class mutation operators for C++ with the MuCPP mutation system. *Inf. Softw. Technol.* **81**, 169–184 (2017)
- DeMillo, R.A., Guindi, D.S., McCracken, W.M., Offutt, A.J., King, K.N.: An extended overview of the Mothra software testing environment. In: Workshop on Software Testing, Verification, and Analysis, pp. 142–151 (1988)
- DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. *IEEE Comput.* **11**(4), 34–41 (1978)
- Deng, L., Offutt, A.J., Ammann, P., Mirzaei, N.: Mutation operators for testing android apps. *Inf. Softw. Technol.* **81**, 154–168 (2017)
- Denisov, A., Pankevich, S.: Mull it over: mutation testing based on LLVM. In: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 25–31 (2018)
- Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P., Heymans, P.: Model-based mutant equivalence detection using automata language equivalence and simulations. *J. Syst. Softw.* **141**, 1–15 (2018)
- Esterro-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I.: Mutation operators for WS-BPEL 2.0. In: International Conference on Software & Systems Engineering and their Applications (ICSSEA) (2008)
- Fabbri, S.C.P.F., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Mutation analysis testing for finite state machines. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 220–229 (1994)
- Fabbri, S.C.P.F., Maldonado, J.C., P.C., Masiero, M.E., Delamaro, Wong, W.E.: Mutation testing applied to validate specifications based on Petri nets. In: International Conference on Formal Description Techniques, volume 43 of IFIP Conference Proceedings. Chapman & Hall, pp. 329–337 (1995)
- Fabbri, S.C.P.F., Maldonado, J.C., Sugeta, T., Masiero, P.C.: Mutation testing applied to validate specifications based on statecharts. In: International Symposium on Software Reliability Engineering (ISSRE), p. 210 (1999)
- Ferenc, R., Beszedes, A., Tarkiainen, M., Gyimothy, T.: Columbus—reverse engineering tool and schema for C++. In: International Conference on Software Maintenance, pp. 172–181 (2002)
- Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. *IEEE Trans. Softw. Eng.* **38**(2), 278–292 (2012)
- Gligoric, M., Badame, S., Johnson, R.: SMutant: a tool for type-sensitive mutation testing in a dynamic language. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE) and European Software Engineering Conference (ESEC), pp. 424–427 (2011)
- GMF. <https://www.eclipse.org/gmf-tooling/>. Accessed Oct. 2019
- Gómez-Abajo, P., Guerra, E., de Lara, J.: A domain-specific language for model mutation and its application to the automated generation of exercises. *Comput. Lang. Syst. Struct.* **49**, 152–173 (2017)
- Gómez-Abajo, P., Guerra, E., de Lara, J., Merayo, M.G.: A tool for domain-independent model mutation. *Sci. Comput. Program.* **163**, 85–92 (2018)
- Gómez-Abajo, P., Guerra, E., de Lara, J., Merayo, M.G.: Towards a model-driven engineering solution for language independent mutation testing. In: Jornadas de Ingeniería del Software y Bases de Datos (JISBD), p. 4. Biblioteca digital SISTEDES (2018)
- Gómez-Abajo, P., Guerra, E., de Lara, J., Merayo, M.G.: Mutation testing for DSLs (tool demo). In: ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM). ACM, pp. 60–62 (2019)
- Gong, D., Zhang, G., Yao, X., Meng, F.: Mutant reduction based on dominance relation for weak mutation testing. *Inf. Softw. Technol.* **81**, 82–96 (2017)
- González-Jiménez, M., de Lara, J.: Datalyzer: streaming data applications made easy. In: International Conference on Web Engineering (ICWE), volume 10845 of LNCS. Springer, pp. 420–429 (2018)

39. Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., Groce, A.: How hard does mutation analysis have to be, anyway? In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 216–227 (2015)
40. Granda, M.F., Condori-Fernández, N., Vos, T.E.J., Pastor, O.: Mutation operators for UML class diagrams. In: International Conference on Advanced Information Systems Engineering (CAiSE), volume 9694 of LNCS. Springer, pp. 325–341 (2016)
41. Groce, A., Holmes, J., Marinov, D., Shi, A., Zhang, L.: An extensible, regular-expression-based tool for multi-language mutant generation. In: International Conference on Software Engineering (ICSE). ACM, pp. 25–28 (2018)
42. Guerra, E., Cuadrado, J.S., de Lara, J.: Towards effective mutation testing for ATL. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS). IEEE, pp. 78–88 (2019)
43. Hamlet, R.G.: Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.* **3**(4), 279–290 (1977)
44. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and refinement of textual syntax for models. In: Proceedings of ECMDA-FA, volume 5562 of Lecture Notes in Computer Science. Springer, pp. 114–129 (2009)
45. Heidenreich, F., Johannes, J., Reimann, J., Seifert, M., Wende, C., Werner, C., Wilke, C., Assmann, U.: Model-driven modernisation of java programs with jamopp. In: Proceedings of Model-Driven Software Migration, volume 708 CEUR Workshop Proceedings, pp. 1–4 (2011)
46. Hierons, R.M., Merayo, M.G.: Mutation testing from probabilistic and stochastic finite state machines. *J. Syst. Softw.* **82**(11), 1804–1818 (2009)
47. Jia, Y., Harman, M.: MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language. In: Testing: Academic Industrial Conference—Practice and Research Techniques (TAICPART), pp. 94–98 (2008)
48. Jia, Y., Harman, M.: Higher order mutation testing. *Inf. Softw. Technol.* **51**(10), 1379–1393 (2009)
49. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* **37**(5), 649–678 (2011)
50. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. *Sci. Comput. Program.* **72**(1–2), 31–39 (2008)
51. Just, R.: The Major mutation framework: efficient and scalable mutation analysis for Java. In: International Symposium on Software Testing and Analysis (ISSTA). ACM, pp. 433–436 (2014). <http://mutation-testing.org/>
52. Kelly, S., Tolvanen, J.: Domain-Specific Modeling-Enabling Full Code Generation. Wiley, Hoboken (2008)
53. Khan, Y., Hassine, J.: Mutation operators for the Atlas Transformation Language. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 43–52 (2013)
54. Kim, S., Clark, J.A., McDermid, J.A.: Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Softw. Test. Verif. Reliab.* **11**(3), 207–225 (2001)
55. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. *Softw. Pract. Exp.* **21**(7), 685–718 (1991)
56. Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Traon, Y.L., Harman, M.: Detecting trivial mutant equivalences via compiler optimisations. *IEEE Trans. Softw. Eng.* **44**(4), 308–333 (2018)
57. Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., Traon, Y.L.: How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empir. Softw. Eng.* **23**(4), 2426–2463 (2018)
58. Krenn, W., Schlick, R., Tiran, S., Aichernig, B., Jobstl, E., Brandl, H.: MoMut: UML model-based mutation testing for UML. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 1–8 (2015)
59. Kusano, M., Wang, C.: CCmutator: mutation generator for concurrency constructs in multithreaded C/C++ applications. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE Press, pp. 722–725 (2013)
60. Laurent, T., Papadakis, M., Kintis, M., Henard, C., Traon, Y.L., Ventresque, A.: Assessing and improving the mutation testing practice of PIT. In: International Conference on Software Testing, Verification and Validation (ICST), pp. 430–435 (2017)
61. Le, D., Alipour, M.A., Gopinath, R., Groce, A.: Mucheck: an extensible tool for mutation testing of haskell programs. In: International Symposium on Software Testing and Analysis (ISSTA). ACM, pp. 429–432 (2014)
62. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. *IEEE Trans. Softw. Eng.* **38**(1), 54–72 (2012)
63. Lee, S.C., Offutt, A.J.: Generating test cases for XML-based web component interactions using mutation analysis. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 200–209 (2001)
64. López, S., Alfonzo, G.A., Perez, O., Gonzalez, S., Montes, R.: A metamodel to carry out reverse engineering of C++ code into UML sequence diagrams. In: Electronics, Robotics and Automotive Mechanics Conference (CERMA), vol. 2, pp. 331–336 (2006)
65. Ma, Y.S., Kwon, Y.R., Offutt, A.J.: Inter-class mutation operators for Java. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 352–366 (2002)
66. Ma, Y.-S., Offutt, J., Kwon, Y.R.: MuJava: a mutation system for Java. In: International Conference on Software Engineering (ICSE), pp. 827–830 (2006)
67. Ma, Y.-S., Offutt, J., Kwon, Y.R.: Mujava: an automated class mutation system. *Softw. Test. Verif. Reliab.* **15**(2), 97–133 (2005)
68. Mariya, F., Barkhas, D.: A comparative analysis of mutation testing tools for Java. In: IEEE East-West Design Test Symposium (EWDTS), pp. 1–3 (2016)
69. Mirshokraie, S., Mesbah, A., Pattabiraman, K.: Guided mutation testing for JavaScript web applications. *IEEE Trans. Softw. Eng.* **41**(5), 429–444 (2015)
70. MOF. <http://www.omg.org/spec/MOF> (2016)
71. Mottu, J., Baudry, B., Traon, Y.L.: Mutation analysis testing for model transformations. In: European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA), pp. 376–390 (2006)
72. Mottu, J., Sen, S., Tisi, M., Cabot, J.: Static analysis of model transformations for effective test generation. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 291–300 (2012)
73. Object Management Group. UML 2.4 OCL Specification (2014). <http://www.omg.org/spec/OCL/>
74. Object Management Group. UML 2.5.1 Specification (2017). <https://www.omg.org/spec/UML/About-UML/>
75. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* **5**(2), 99–118 (1996)
76. Offutt, A.J., Voas, J., Payne, J.: Mutation operators for ADA. Technical report, Information and Software Systems Engineering, George Mason University (1996)
77. Papadakis, M., Henard, C., Harman, M., Jia, Y., Le Traon, Y.: Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 354–365 (2016)
78. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: an analysis and survey, volume 112 of Advances in Computers. Elsevier, pp. 275–378 (2019)
79. Parsai, A., Murgia, A., Demeyer, S.: LittleDarwin: a feature-rich and extensible mutation testing framework for large and complex

- Java systems. In: Fundamentals of Software Engineering (FSEN), volume 10522 of LNCS. Springer, pp. 148–163 (2017). <http://littledarwin.parsai.net/>, <https://github.com/aliparsai/LittleDarwin>
80. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Static analysis of model transformations. *IEEE Trans. Softw. Eng.* **43**(9), 868–897 (2017)
 81. Schuler, D., Zeller, A.: Javalanche: efficient mutation testing for Java. In: Joint Meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering. ACM, pp. 297–298 (2009). <http://javalanche.org/>, <https://github.com/david-schuler/javalanche>
 82. Sen, S., Baudry, B.: Mutation-based model synthesis in model driven engineering. In: Workshop on Mutation Analysis (Mutation) (2006)
 83. Sen, S., Baudry, B., Mottu, J.-M.: Automatic model generation strategies for model transformation testing. In: Theory and Practice of Model Transformations. Springer, Berlin, pp. 148–164 (2009)
 84. Shin, D., Jee, E., Bae, D.: Comprehensive analysis of FBD test coverage criteria using mutants. *Softw. Syst. Model.* **15**(3), 631–645 (2016)
 85. Sirius. <https://www.eclipse.org/sirius/>. Accessed Oct. 2019
 86. Srivatanakul, T., Clark, J.A., Stepney, S., Polack, F.: Challenging formal specifications by mutation: a CSP security example. In: Asia-Pacific Software Engineering Conference (APSEC), pp. 340–350 (2003)
 87. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd Edition. Addison-Wesley Professional (2008)
 88. Trakhtenbrot, M.: New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models. In: Workshop on Mutation Analysis (Mutation), pp. 151–160 (2007)
 89. Troya, J., Bergmayr, A., Burgueño, L., Wimmer, M.: Towards systematic mutations for and with ATL model transformations. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10 (2015)
 90. Tuya, J., Cabal, M.J.S., de la Riva, C.: SQLMutation: a tool to generate mutants of SQL database queries. In: Workshop on Mutation Analysis (Mutation), p. 1 (2006)
 91. Tuya, J., Cabal, M.J.S., de la Riva, C.: Mutating database queries. *Inf. Softw. Technol.* **49**(4), 398–417 (2007)
 92. Voelter, M.: DSL Engineering—Designing, Implementing and Using Domain-Specific Languages. dslbook.org (2013)
 93. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), volume 6981 of LNCS. Springer, pp. 623–637 (2011)
 94. Wu, F., Nanavati, J., Harman, M., Jia, Y., Krinke, J.: Memory mutation testing. *Inf. Softw. Technol.* **81**, 97–111 (2017)
 95. Yazdani, S.A., Amiri, M.J., Parsa, S., Koupaee, M.: Automatic test cases generation from business process models. *Requir. Eng.* **24**(8), 119–132 (2018)
 96. Zhang, L., Gligoric, M., Marinov, D., Khurshid, S.: Operator-based and random mutant selection: better together. In: Proceedings of ASE. IEEE, pp. 92–102 (2013)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Pablo Gómez-Abajo is Assistant Professor at the Computer Science department of the Universidad Autónoma de Madrid. After around 8 years working in the industry, he returned to Academia and joined the modelling and Software Engineering research group in 2015.



Esther Guerra is Associate Professor at the Computer Science department of the Universidad Autónoma de Madrid. Together with J. de Lara, she leads the modelling and Software Engineering research group <http://miso.es>. She is interested in flexible modelling, meta-modelling, domain specific languages and model transformation.



Juan de Lara is Full professor at the Computer Science department of the Universidad Autónoma de Madrid. Together with E. Guerra, he leads the modelling and Software Engineering research group. His research interests are in model-driven engineering and automated software development.



Mercedes G. Merayo holds an Associate Professor position in the Computer Systems and Computation Department of the Universidad Complutense de Madrid. She leads, together with Luis Llana the Design and Testing of Reliable Systems research group. Her current research interests include model based testing, distributed testing, asynchronous testing, mutation testing, metamorphic testing and timed extensions in formal testing.