International Journal on Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Wodel-Test: A Model-Based Framework for Language-Independent Mutation Testing

Pablo Gómez-Abajo · Esther Guerra · Juan de Lara · Mercedes G. Merayo

the date of receipt and acceptance should be inserted later

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. 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.

Pablo Gómez-Abajo
Modelling & Software Engineering Research Group
Universidad Autónoma de Madrid (Spain)
E-mail: Pablo.GomezA@uam.es

Esther Guerra
Modelling & Software Engineering Research Group
Universidad Autónoma de Madrid (Spain)
E-mail: Esther.Guerra@uam.es

Juan de Lara
Modelling & Software Engineering Research Group
Universidad Autónoma de Madrid (Spain)
E-mail: Juan.deLara@uam.es

Mercedes G. Merayo
Design and Testing of Reliable Systems Research Group
Universidad Complutense de Madrid (Spain)
E-mail: mgmerayo@fdi.ucm.es

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 extensibility 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 plugins 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, Section 2 provides background on MT and MDE and introduces a running example. Next, Section 3 presents our approach to specify MT tools, and Section 4 shows the proposed architecture and tool support. Section 5 reports on two case studies consisting 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, Section 6 analyses related research and Section 7 finishes with the conclusions and future work.

2 Background

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

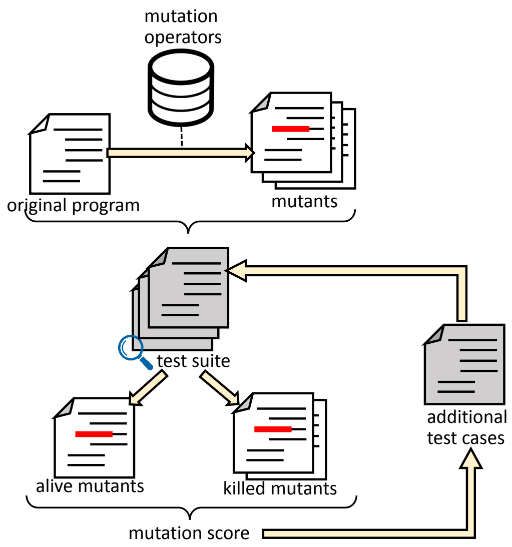


Figure 1: MT process.

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.

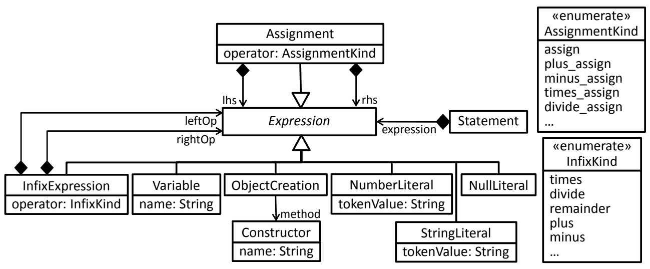
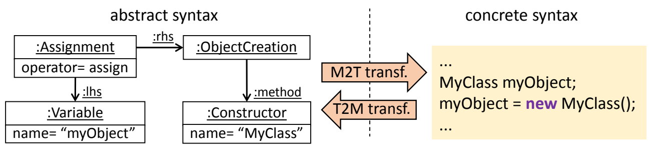


Figure 2: Java meta-model excerpt (from MoDisco [15]).

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".



```
...
MyClass myObject;
myObject = new MyClass();
...
```

Figure 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 realise reverse-engineering processes that transform an input text into an output model. In Figure 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 Figure 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 Sections 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 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 tests 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.

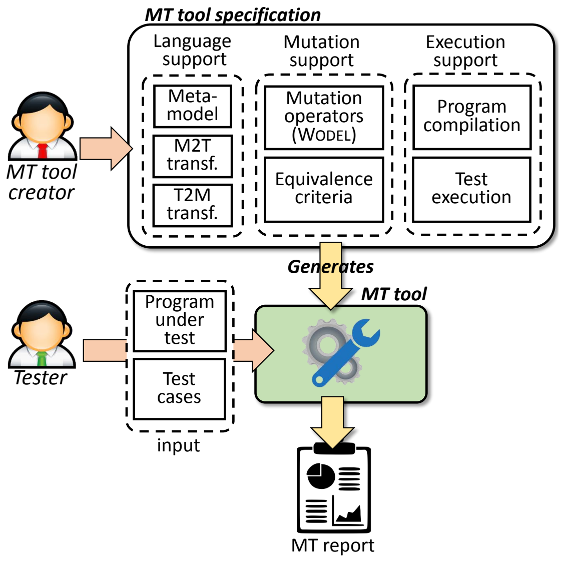


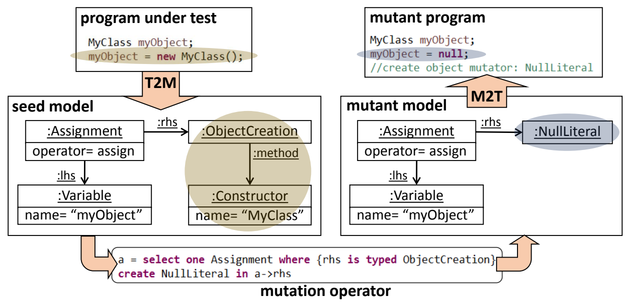
Figure 4: Creating and using a MT tool with WODEL-TEST.

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, Figure 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 Figure 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.



```
MyClass myObject;
myObject = new MyClass();
```

```
MyClass myObject;
myObject = null;
//create object mutator: NullLiteral
```

```
a = select one Assignment where {rhs is typed ObjectCreation}
create NullLiteral in a->rhs
```

Figure 5: Working scheme of the MT tool for Java.

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 Figure 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 Modeling 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.

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

```
 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.

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.

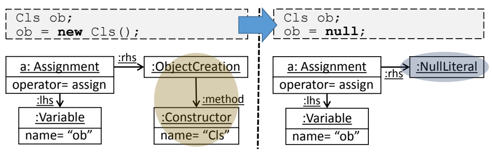
Line 2 indicates the meta-model of the language the seed models conform to, and line 3 provides a description of the program, which is optional. Although not illustrated in the listing, we have extended Wodel to allow including the meta-model of auxiliary resources that can be queried in the mutation 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.

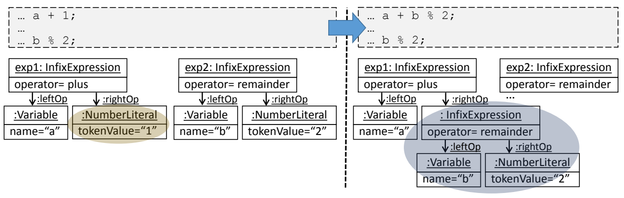
Lines 5–10 of Listing 1 contain the body of the program with the mutation operators. The operators can be organized in blocks (line 5) and use the mutants generated by other blocks as seed models, e.g., to define higher-order mutants. Lines 6–9 declare a mutation operator named CIR composed of two statements. 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. Figure 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*.

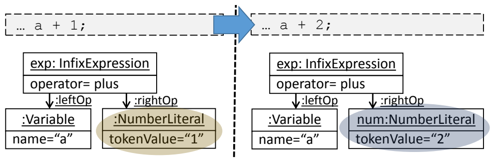
Wodel provides primitives to *select*, *create*, *clone*, *modify*, *retype* and *delete* information from models. Table 1 illustrates these primitives using the running example. In each case, we show a sample Wodel program and its application to an excerpt of Java code. In particular, Wodel supports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

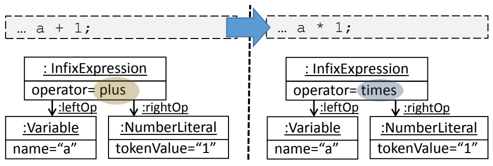
– *Object selection.* Primitive select selects an object with the given type and field values. The table shows two examples in rows 1 and 2. In the first case, the Wodel program selects an object of the type Assignment whose rhs reference is of the type ObjectCreation. In the second case, the program 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 initialises 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 program 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 exam-

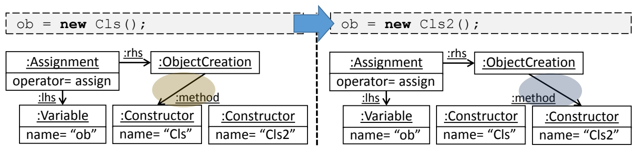
	Primitive	Example of Wodel program	Example of model mutation

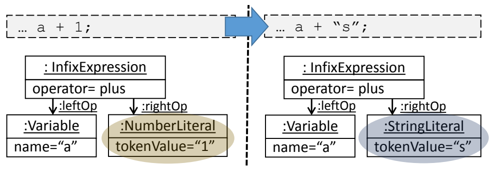
1	Object creation	// replaces a constructor call by null
		a = **select one** Assignment
		    where {rhs **is typed** ObjectCreation}
		create NullLiteral **in** a→rhs
			

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 modifica-tion	// 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 modifica-tion	// modifies the operator '+' or '−' of an expression by '∗'
		modify one InfixExpression
		    where {operator **in** ['+', '−']}
		    with {operator = '∗'}
			

5	Target reference modifica-tion	// 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
		retype one NumberLiteral
		    as StringLiteral
		    with {tokenValue = 's'}
			

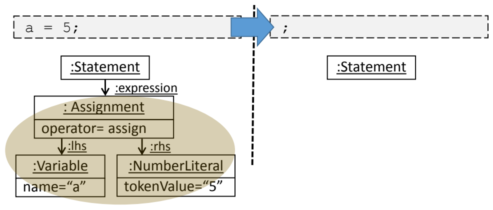
7	Object deletion	// removes an assignment with operator '=' **remove one** Assignment **where** {operator = '='}	

Table 1: Examples of Wodel mutation primitives.

ple in row 5, where the target of a reference called method is changed from one Constructor to a different one.

– *Object retyping.* 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 reference[1] 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 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, Section 4.1 describes its architecture, and Section 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 plugins [36]. It uses the Eclipse Modeling 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.

[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*.

github.io/Wodel/. The web page also includes installation instructions, examples, video demos, and the source code.

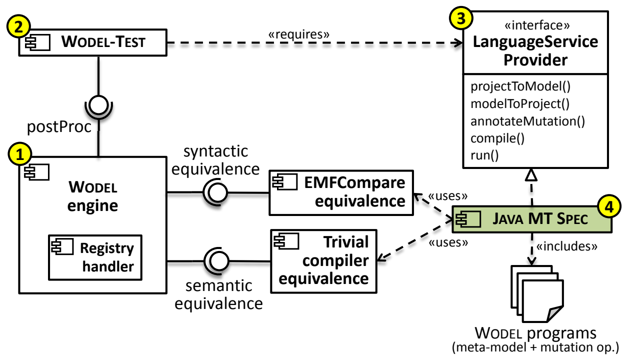


Figure 6: Architecture of Wodel-Test.

Wodel is a generic environment to define and apply model mutation operators using the DSL presented in Section 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 optimisation 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 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.

[2] https://www.eclipse.org/emf/compare/

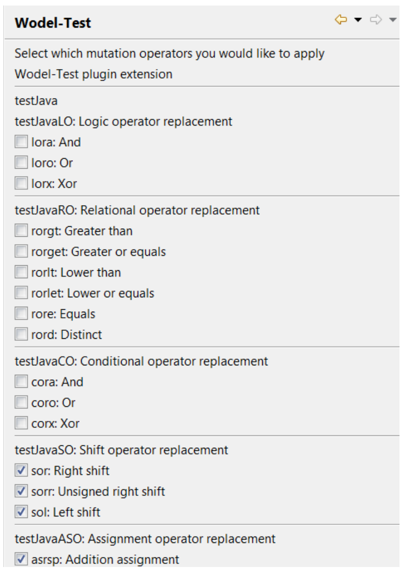
If an EMF definition of the target language already exists, the implementation of projectToModel and modelToProject is not typically needed. For interpreted languages, implementing compile is not necessary either.

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 Figure 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 visualised 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



Wodel-Test

Select which mutation operators you would like to apply
Wodel-Test plugin extension

testJava

testJavaLO: Logic operator replacement
- lora: And
- loro: Or
- lorx: Xor

testJavaRO: Relational operator replacement
- rorgt: Greater than
- rorget: Greater or equals
- rorlt: Lower than
- rorlet: Lower or equals
- rore: Equals
- rord: Distinct

testJavaCO: Conditional operator replacement
- cora: And
- coro: Or
- corx: Xor

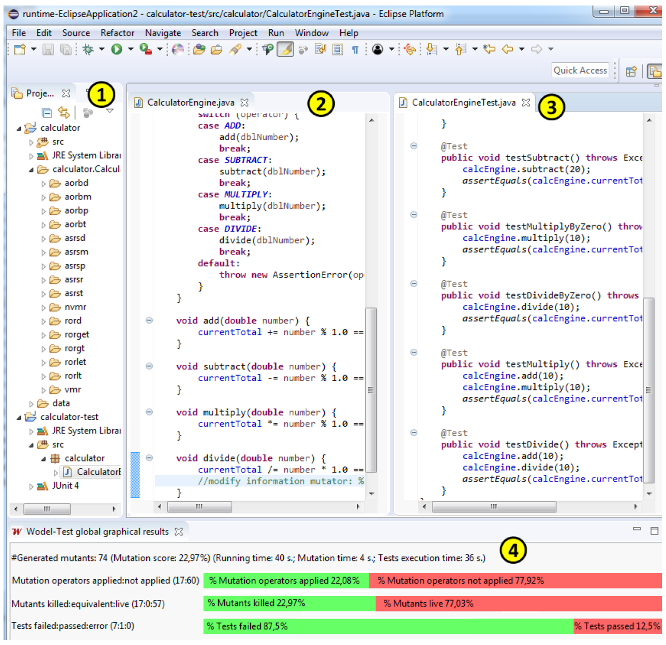
testJavaSO: Shift operator replacement
- [✓ sor: Right shift]
- [✓ sorr: Unsigned right shift]
- [✓ sol: Left shift]

testJavaASO: Assignment operator replacement
- [✓ asrsp: Addition assignment]

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

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 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 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.



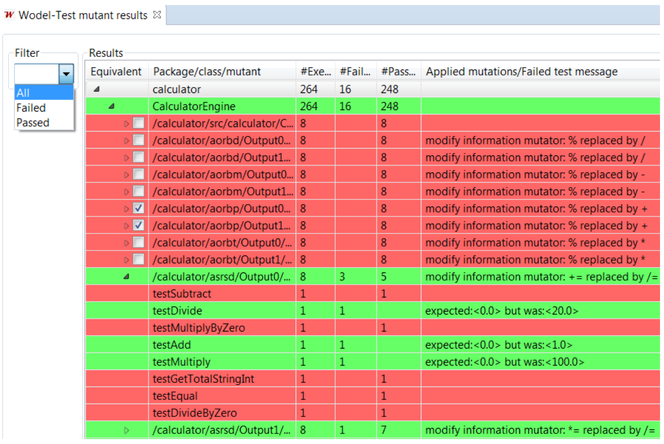
```
        switch (operator) {
            case ADD:
                add(dblNumber);
                break;
            case SUBTRACT:
                subtract(dblNumber);
                break;
            case MULTIPLY:
                multiply(dblNumber);
                break;
            case DIVIDE:
                divide(dblNumber);
                break;
            default:
                throw new AssertionError(op
            }
        }

    void add(double number) {
        currentTotal += number % 1.0 ==
    }

    void subtract(double number) {
        currentTotal -= number % 1.0 ==
    }

    void multiply(double number) {
        currentTotal *= number % 1.0 ==
    }

    void divide(double number) {
        currentTotal /= number * 1.0 ==
        //modify information mutator: %
    }
```

```
    }

    @Test
    public void testSubtract() throws Exce
        calcEngine.subtract(20);
        assertEquals(calcEngine.currentTot
    }

    @Test
    public void testMultiplyByZero() throw
        calcEngine.multiply(10);
        assertEquals(calcEngine.currentTot
    }

    @Test
    public void testDivideByZero() throws
        calcEngine.divide(10);
        assertEquals(calcEngine.currentTot
    }

    @Test
    public void testMultiply() throws Exce
        calcEngine.add(10);
        calcEngine.multiply(10);
        assertEquals(calcEngine.currentTot
    }

    @Test
    public void testDivide() throws Except
        calcEngine.add(10);
        calcEngine.divide(10);
        assertEquals(calcEngine.currentTot
    }
```

#Generated mutants: 74 (Mutation score: 22,97%) (Running time: 40 s.; Mutation time: 4 s.; Tests execution time: 36 s.)

Mutation operators applied:not applied (17:60)	% Mutation operators applied 22,08%	% Mutation operators not applied 77,92%
Mutants killed:equivalent:live (17:0:57)	% Mutants killed 22,97%	% Mutants live 77,03%
Tests failed:passed:error (7:1:0)	% Tests failed 87,5%	% Tests passed 12,5%

Figure 8: Global view of MT results.



Equivalent	Package/class/mutant	#Exe...	#Fail...	#Pass...	Applied mutations/Failed test message
	calculator	264	16	248	
	CalculatorEngine	264	16	248	
☐	/calculator/src/calculator/C...	8		8	
☐	/calculator/aorbd/Output0...	8		8	modify information mutator: % replaced by /
☐	/calculator/aorbd/Output1...	8		8	modify information mutator: % replaced by /
☐	/calculator/aorbm/Output0...	8		8	modify information mutator: % replaced by -
☐	/calculator/aorbm/Output1...	8		8	modify information mutator: % replaced by -
✓	/calculator/aorbp/Output0...	8		8	modify information mutator: % replaced by +
✓	/calculator/aorbp/Output1...	8		8	modify information mutator: % replaced by +
☐	/calculator/aorbt/Output0/...	8		8	modify information mutator: % replaced by *
☐	/calculator/aorbt/Output1/...	8		8	modify information mutator: % replaced by *
	/calculator/asrsd/Output0/...	8	3	5	modify information mutator: += replaced by /=
	testSubtract	1		1	
	testDivide	1	1		expected:<0.0> but was:<20.0>
	testMultiplyByZero	1		1	
	testAdd	1	1		expected:<0.0> but was:<1.0>
	testMultiply	1	1		expected:<0.0> but was:<100.0>
	testGetTotalStringInt	1		1	
	testEqual	1		1	
	testDivideByZero	1		1	
	/calculator/asrsd/Output1/...	8	1	7	modify information mutator: *= replaced by /=

Figure 9: Detailed view of MT results: Tests passed and failed by each mutant.

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.

[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.

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, Section 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, Section 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 (Section 5.1.1); then, we focus on the extensibility and customizability of mutation operators (Section 5.1.2); finally, we analyse the efficiency of the MT process (Section 5.1.3). The section concludes by answering RQ1 and discussing threats to validity (Section 5.1.4).

5.1.1 General features

In the following, we discuss the general features of the analysed MT tools, which are also summarised 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 plugin; and PITest can run on the command line but also provides plugins for integration with Maven, Gradle, Eclipse and IntelliJ, which enables a rich integration 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 5[4]) 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 Section 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

[4] `https://junit.org/junit5/`

	Major	**Javalanche**	**PITest**	**LittleDarwin**	**muJava**	**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
			Input to MT process			
UI	Command line	Command line	Command line Tool plugins	Command line	Java GUI	Eclipse plugin
Scope	project/class	project	project/class	project	project/class	project/class
Test suite	JUnit 4	JUnit 4	JUnit 4	JUnit 4	JUnit 4	JUnit 4 (customizable)
			Mutant generation process			
N. 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	Yes (dynamic inv.)	No	No	No	Yes (TCE)
			Reporting			
Report type	CSV	HTML	HTML	HTML	GUI	Interactive views
N. of mutants	✓	✓	✓	✓	✓	✓
Mutation score	✓	✓	✓	✓	✓	✓
Killed/live mutants	Number	Number	Number	Number	Number, list	Number, list
Covered operators	Number	Number	Number, list	✗	Number, list	Number, list
Mutants by class	✗	✗	✓	✓	✓	✓
Mutants by test	✗	✗	✗	✗	✓	✓
Tests by mutant	✗	✗	✗	✗	✗	✓

Table 2: Comparison of Java MT tools.

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 Figure 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 visualise 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 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 Figure 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 Figure 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

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

Listing 2: Major-Mml program.

```
AOR "Arihtmetic operator replacement" {
 modify one InfixExpression
   where {operator=['*', '/', '%']}
   with {operator=['*', '/', '%']}
}

ROR "Relational operator replacement" {
 modify one InfixExpression
   where {operator=['>', '==']}
   with {operator=['<','<=','!=','==','>','>=']}
}
```

Listing 3: Wodel program.

(STD). These primitives can be customised 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 Section 3.3, Wodel is a full-fledged DSL that enables the creation of arbitrary mutation 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 project[5], which has 74 Java classes and 2 586 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.

Tool	**Mutants (killed/alive)**	**Mutation score**	**Running time**	**Per-mutant time**
Major	1 638 (331/864)	20.21%	13h42m56s	30.14s
PITest	918 (321/597)	34.97%	1h11m20s	4.66s
LittleDarwin	439 (130/309)	29.61%	3h30m39s	28.79s
muJava	2 589 (557/2 032)	21.51%	5h30m10s	7.65s
Wodel-Test/Java	4 756 (985/3 771)	20.71%	3h24m23s	2.58s

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

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. Figure 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

[5] `https://github.com/soursop/functional-matrix-operator`

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 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, 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 Figure 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,

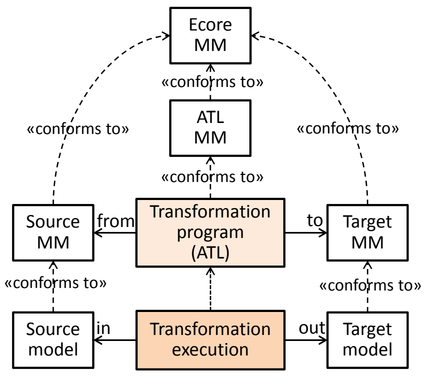


Figure 10: Working scheme of ATL transformations.

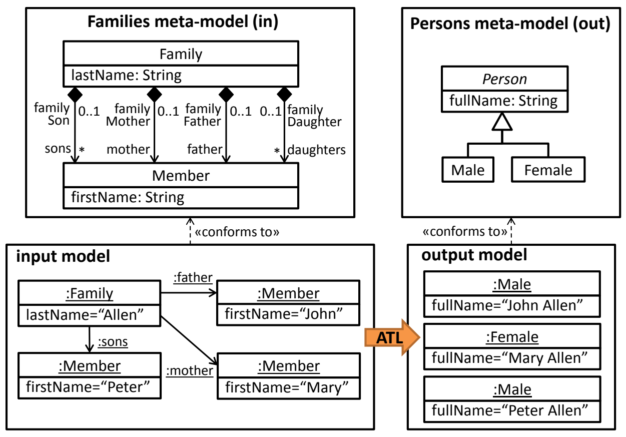


Figure 11: ATL transformation example.

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 purpose 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, Section 5.2.1 introduces the basics of ATL. Then, Section 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, Section 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 Figure 10 shows, meta-models conform to Ecore, which is a meta-modelling 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 Figure 11.

```
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.oclIsUndefined() then true
9      else if not self.familyDaughter.oclIsUndefined() 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.oclIsUndefined() then
15                self.familyFather.lastName
16     else if not self.familyMother.oclIsUndefined() then
17               self.familyMother.lastName
18          else
19                if not self.familySon.oclIsUndefined() then
20                    self.familySon.lastName
21               else self.familyDaughter.lastName endif
22          endif
23     endif;
24 -- Creates a male from a family member that is a male
25 rule Member2Male {
26     from s:Families!Member (not s.isFemale())
27     to t:Persons!Male (
28         fullName <- s.firstName + ' ' + s.familyName
29     )
30 }
31 -- Creates a female from a family member that is a female
32 rule Member2Female {
33     from s:Families!Member (s.isFemale())
34     to t:Persons!Female (
35         fullName <- s.firstName + ' ' + s.familyName
36     )
37 }
```

Listing 4: ATL program example.

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

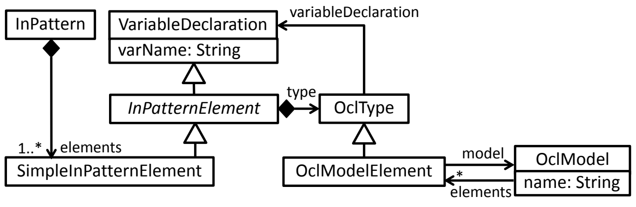


Figure 12: ATL meta-model excerpt.

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]).

Concept	**Mutation operator**	**LOC in Wodel**	**LOC in ATL**
Matched Rule	Addition	1	-
	Deletion	1	-
	Name change	1	-
In Pattern Element	Addition	6	14
	Deletion	1	-
	Class change	4	-
	Name change	1	-
Filter	Addition	10	-
	Deletion	1	-
	Condition change	4	-
Out Pattern Element	Addition	6	-
	Deletion	1	6
	Class change	4	-
	Name change	1	-
Binding	Addition	6	-
	Deletion	1	3
	Value change	2	-
	Feature change	6	-

Table 4: Mutation operators for ATL.

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).

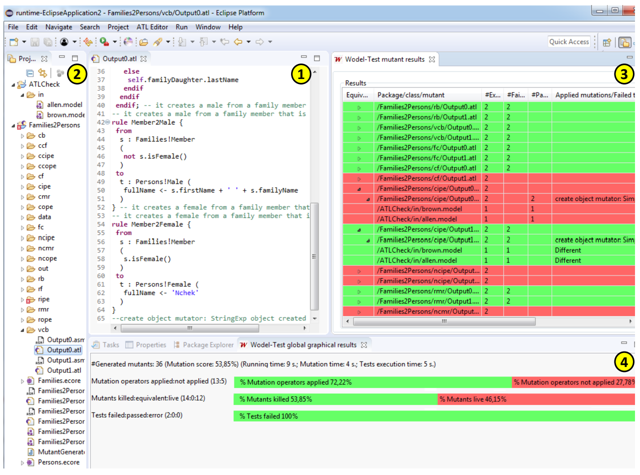
WODEL ensures that all generated mutants will conform to the ATL meta-model. In addition, it provides an extension point to add other criteria that any mutant should fulfil. In this case, we have implemented this extension point to statically analyse the mutant and discard it if it has typing errors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

```
1  generate 2 mutants in "out/" from "model/"
2  metamodel "mm/ATL.ecore"
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 = random−string(2, 4)}
13       elem = create OclModelElement in ipe→type with {name = cl.
                name, variableDeclaration = ipe}
14       modify mod with {elements += elem}
15    }
16 }
```

Listing 5: Wodel program implementing the mutation operator *in pattern element addition.*

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 Figure 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.



```
36     else
37      self.familyDaughter.lastName
38     endif
39    endif
40   endif; -- it creates a male from a family member
41 -- it creates a male from a family member that is
42 rule Member2Male {
43  from
44   s : Families!Member
45   (
46    not s.isFemale()
47   )
48  to
49   t : Persons!Male (
50    fullName <- s.firstName + ' ' + s.familyName
51   )
52 } -- it creates a female from a family member that
53 -- it creates a female from a family member that i
54 rule Member2Female {
55  from
56   s : Families!Member
57   (
58    s.isFemale()
59   )
60  to
61   t : Persons!Female (
62    fullName <- 'Nchek'
63   )
64 }
65 --create object mutator: StringExp object created
```

Results

Equiv...	Package/class/mutant	#Ex...	#Fai...	#Pa...	Applied mutations/Failed t
🟩 ▷	/Families2Persons/rb/Output0.atl	2	2		
🟩 ▷	/Families2Persons/rb/Output1.atl	2	2		
🟩 ▷	/Families2Persons/vcb/Output0....	2	2		
🟩 ▷	/Families2Persons/vcb/Output1....	2	2		
🟩 ▷	/Families2Persons/fc/Output0.atl	2	2		
🟩 ▷	/Families2Persons/fc/Output1.atl	2	2		
🟩 ▷	/Families2Persons/cf/Output0.atl	2	2		
🟥 ▷	/Families2Persons/cf/Output1.atl	2			
🟥 ◢	/Families2Persons/cipe/Output0...	2			
🟥 ◢	/Families2Persons/cipe/Output0...	2		2	create object mutator: Sim
🟥	/ATLCheck/in/brown.model	1		1	
🟥	/ATLCheck/in/allen.model	1		1	
🟩 ◢	/Families2Persons/cipe/Output1...	2	2		
🟩 ◢	/Families2Persons/cipe/Output1...	2	2		create object mutator: Sim
🟩	/ATLCheck/in/brown.model	1	1		Different
🟩	/ATLCheck/in/allen.model	1	1		Different
🟥 ▷	/Families2Persons/ncipe/Output...	2			
🟥 ▷	/Families2Persons/ncipe/Output...	2			
🟩 ▷	/Families2Persons/rmr/Output0....	2	2		
🟩 ▷	/Families2Persons/rmr/Output1....	2	2		
🟥 ▷	/Families2Persons/ncmr/Output...	2			

#Generated mutants: 36 (Mutation score: 53,85%) (Running time: 9 s.; Mutation time: 4 s.; Tests execution time: 5 s.)

Mutation operators applied:not applied (13:5)	🟩 % Mutation operators applied 72,22%	🟥 % Mutation operators not applied 27,78%
Mutants killed:equivalent:live (14:0:12)	🟩 % Mutants killed 53,85%	🟥 % Mutants live 46,15%
Tests failed:passed:error (2:0:0)	🟩 % Tests failed 100%	

Figure 13: MT tool for ATL created with Wodel-Test.

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).

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 mode[6] 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 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 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

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

```
module AddInPatternElement_FirstRule;

create OUT : ATL refining IN : ATL;

-- Sequence for giving new variable names to new
-- pattern elements that are created
helper def : varNames : Sequence(String) =
  Sequence{'a','aa','b','bb','c','cc','d','dd','e','ee','f','ff','...'};

rule AddInPatternElement {
  from s : ATL!InPattern (ATL!Rule.allInstances()→first() = s."rule")
        -- Add SIPE only in first rule
  to t : ATL!InPattern(
      elements <- s.elements →append(newIPE)
  ),
  newIPE : ATL!SimpleInPatternElement (
      -- Give a variable name that no PatternElement has
      varName <- thisModule.varNames→any(n |
          ATL!PatternElement.allInstances()→collect(pe|pe.varName)
              →excludes(n)),
      type <- newOCLType
  ),
  -- The type is composed of a model and a name: model!name
  newOCLType : ATL!OclModelElement(
      model <- s.elements→first().type.model,
      name <- 'Complete_IPE'
  )
}
```

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

```
module SecondOrderHOT;

create OUT : ATL
  refining IN : ATL, IN_MM : IN_MM, OUT_MM : OUT_MM;

helper def : random() : Real =
    "#native"!"java::util::Random".newInstance().nextDouble();

-- A StringExp is one of the types that can conform to the value part
-- of a Binding. Since the generic mutation transformation
-- adds 'Complete_IPE' in the value part, a StringExp is created,
-- whose stringSymbol is 'Complete_IPE'
rule CompleteInMMNames {
    from s : ATL!StringExp (s.stringSymbol = 'Complete_IPE')
    using {
        classes : Sequence(IN_MM!EClass) =
          IN_MM!EClass.allInstancesFrom('IN_MM')→select(c |
                not c.abstract);
    }
    to t : ATL!StringExp(
        -- A random class from the input model
        stringSymbol <- classes→at((thisModule.random()*
                                 classes→size()).floor()+1).name
    )
    do {
        classes→at((thisModule.random()*classes→size()).
                    floor()+1).name;
    }
}
```

Listing 7: Refining transformation to replace the literal "Complete_IPE" by an input class name [89].

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.

	Troya et al. [89]	**Wodel-Test/ATL**
Last update	2015	2019
	Input to MT process	
UI	Command line	Eclipse plugin
Scope	ATL program	ATL program
Test suite	None	Input model (customizable)
	Mutant generation process	
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)
	Reporting	
Report type	None	Interactive views
N. of mutants	✗	✓
Mutation score	✗	✓
Killed/live mutants	✗	Number, list
Covered operators	✗	Number, list
Mutants by ATL program	✗	✓
Tests by mutant	✗	✓
Mutants by test	✗	✓

Table 5: Comparison of ATL MT tools.

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 plugins, 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 plugins). 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.

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 Section 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 specifications (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, Section 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).

References

1. Architecture driven modernization. `https://www.omg.org/adm/`. (last accessed in Oct. 2019).
2. H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. P. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical report, Purdue University, 1989.
3. B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification & Reliability*, 25(8):716–748, 2015.
4. F. Alhwikem, R. F. Paige, L. Rose, and R. Alexander. A systematic approach for designing mutation operators for MDE languages. In *Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*, pages 54–59, 2016.
5. I. M. Alsmadi. Using mutation to enhance GUI testing coverage. *IEEE Software*, 30(1):67–73, 2013.
6. J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering (ICSE)*, pages 402–411. ACM, 2005.
7. V. Aranega, J.-M. Mottu, A. Etien, T. Degueule, B. Baudry, and J.-L. Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability*, 25(5-7):653–683, 2014.
8. G. B. Ariel González, Carlos Luna. Mutation testing for Java based on model-driven development (in spanish). In *Simposio Latinoamericano de Ingeniería de Software (CLEI-SLISW)*, 2018.
9. B. Aziz. Towards a mutation analysis of IoT protocols. *Information & Software Technology*, 100:183–184, 2018.
10. R. Baker and I. Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Trans. Software Eng.*, 39(6):787–805, 2013.
11. B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.
12. J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent Java (J2SE 5.0). In *Workshop on Mutation Analysis (Mutation)*, pages 83–92, 2006.

13. M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice, Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017.
14. J. Bruel, B. Combemale, E. Guerra, J. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe. Comparing and classifying model transformation reuse approaches across metamodels. *Software and Systems Modeling*, 19(2):441–465, 2020.
15. H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *Information & Software Technology*, 56(8):1012–1032, 2014.
16. P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.
17. H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A practical mutation testing tool for Java (demo). In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 449–452. ACM, 2016. See also `http://pitest.org/`, `https://github.com/hcoles/pitest`.
18. M. E. Delamaro and Maldonado. Proteum – a tool for the assessment of test adequacy for C programs. In *Conf. Performability in Computing Systems*, pages 79–95, 1996.
19. P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information & Software Technology*, 81:169–184, 2017.
20. R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. An extended overview of the Mothra software testing environment. In *Workshop on Software Testing, Verification, and Analysis*, pages 142–151, 1988.
21. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.
22. L. Deng, A. J. Offutt, P. Ammann, and N. Mirzaei. Mutation operators for testing android apps. *Information & Software Technology*, 81:154–168, 2017.
23. A. Denisov and S. Pankevich. Mull it over: Mutation testing based on LLVM. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31, 2018.
24. X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans. Model-based mutant equivalence detection using automata language equivalence and simulations. *Journal of Systems and Software*, 141:1–15, 2018.
25. A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Mutation operators for ws-bpel 2.0. In *International Conference on Software & Systems Engineering and their Applications (ICSSEA)*, 2008.
26. S. C. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero. Mutation analysis testing for finite state machines. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 220–229, 1994.
27. S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and W. E. Wong. Mutation testing applied to validate specifications based on Petri nets. In *International Conference on Formal Description Techniques*, volume 43 of *IFIP Conference Proceedings*, pages 329–337. Chapman & Hall, 1995.
28. S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *International Symposium on Software Reliability Engineering (ISSRE)*, page 210, 1999.
29. R. Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for C++. In *International Conference on Software Maintenance*, pages 172–181, 2002.
30. G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012.
31. M. Gligoric, S. Badame, and R. Johnson. 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)*, pages 424–427, 2011.
32. GMF. `https://www.eclipse.org/gmf-tooling/`. (last accessed in Oct. 2019).
33. P. Gómez-Abajo, E. Guerra, and J. de Lara. A domain-specific language for model mutation and its application to the automated generation of exercises. *Computer Languages, Systems & Structures*, 49:152–173, 2017.
34. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Sci. Comput. Program.*, 163:85–92, 2018.
35. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Towards a model-driven engineering solution for language independent mutation testing. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, page 4pps. Biblioteca digital SISTEDES, 2018.
36. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Mutation testing for DSLs (tool demo). In *ACM SIGPLAN International Workshop on Domain-Specific Modeling (DSM)*, pages 60–62. ACM, 2019.
37. D. Gong, G. Zhang, X. Yao, and F. Meng. Mutant reduction based on dominance relation for weak mutation testing. *Information & Software Technology*, 81:82–96, 2017.
38. M. González-Jiménez and J. de Lara. Datalyzer: Streaming data applications made easy. In *International Conference on Web Engineering (ICWE)*, volume 10845 of *LNCS*, pages 420–429. Springer, 2018.
39. R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. How hard does mutation analysis have to be, anyway? In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 216–227, 2015.
40. M. F. Granda, N. Condori-Fernández, T. E. J. Vos, and O. Pastor. Mutation operators for UML class diagrams. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 9694 of *LNCS*, pages 325–341. Springer, 2016.
41. A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering (ICSE)*, pages 25–28. ACM, 2018.
42. E. Guerra, J. S. Cuadrado, and J. de Lara. Towards effective mutation testing for ATL. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 78–88. IEEE, 2019.
43. R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, 1977.

44. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proc. ECMDA-FA*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2009.
45. F. Heidenreich, J. Johannes, J. Reimann, M. Seifert, C. Wende, C. Werner, C. Wilke, and U. Assmann. Model-driven modernisation of java programs with jamopp. In *Proc. Model-Driven Software Migration*, volume 708, pages 1–4. CEUR Workshop Proceedings, 2011.
46. R. M. Hierons and M. G. Merayo. Mutation testing from probabilistic and stochastic finite state machines. *Journal of Systems and Software*, 82(11):1804–1818, 2009.
47. Y. Jia and M. Harman. 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)*, pages 94–98, 2008.
48. Y. Jia and M. Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10):1379–1393, 2009.
49. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.
50. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008. See also `http://www.eclipse.org/atl/`.
51. R. Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436. ACM, 2014. See also `http://mutation-testing.org/`.
52. S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation.* Wiley, 2008.
53. Y. Khan and J. Hassine. Mutation operators for the Atlas Transformation Language. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 43–52, 2013.
54. S. Kim, J. A. Clark, and J. A. McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Softw. Test., Verif. Reliab.*, 11(3):207–225, 2001.
55. K. N. King and A. J. Offutt. A Fortran language system for mutation-based software testing. *Softw., Pract. Exper.*, 21(7):685–718, 1991.
56. M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Trans. Software Eng.*, 44(4):308–333, 2018.
57. M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. L. Traon. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23(4):2426–2463, 2018.
58. W. Krenn, R. Schlick, S. Tiran, B. Aichernig, E. Jobstl, and H. Brandl. MoMut: UML model-based mutation testing for UML. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8, 2015.
59. M. Kusano and C. Wang. CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–725. IEEE Press, 2013.
60. T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque. Assessing and improving the mutation testing practice of PIT. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 430–435, 2017.
61. D. Le, M. A. Alipour, R. Gopinath, and A. Groce. Mucheck: an extensible tool for mutation testing of haskell programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 429–432. ACM, 2014.
62. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.
63. S. C. Lee and A. J. Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 200–209, 2001.
64. S. López, G. A. Alfonzo, O. Perez, S. Gonzalez, and R. Montes. A metamodel to carry out reverse engineering of C++ code into UML sequence diagrams. *Electronics, Robotics and Automotive Mechanics Conference (CERMA)*, 2:331–336, 2006.
65. Y. S. Ma, Y. R. Kwon, and A. J. Offutt. Inter-class mutation operators for Java. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 352–366, 2002.
66. Y.-S. Ma, A. J. Offutt, and Y. R. Kwon. MuJava: a mutation system for Java. In *International Conference on Software Engineering (ICSE)*, pages 827–830, 2006.
67. Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system. *Software Testing, Verification & Reliability*, 15(2):97–133, 2005. See also `https://cs.gmu.edu/~offutt/mujava/`, `https://github.com/jeffoffutt/muJava`.
68. F. Mariya and D. Barkhas. A comparative analysis of mutation testing tools for Java. In *IEEE East-West Design Test Symposium (EWDTS)*, pages 1–3, 2016.
69. S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for JavaScript web applications. *IEEE Trans. Software Eng.*, 41(5):429–444, 2015.
70. MOF. `http://www.omg.org/spec/MOF`, 2016.
71. J. Mottu, B. Baudry, and Y. L. Traon. Mutation analysis testing for model transformations. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 376–390, 2006.
72. J. Mottu, S. Sen, M. Tisi, and J. Cabot. Static analysis of model transformations for effective test generation. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 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. A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996.
76. A. J. Offutt, J. Voas, and J. Payne. Mutation operators for ada. Technical report, Information and Software Systems Engineering, George Mason University, 1996.
77. M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 354–365, 2016.
78. M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.
79. A. Parsai, A. Murgia, and S. Demeyer. 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*, pages 148–163. Springer, 2017. See also `http://littledarwin.parsai.net/`, `https://github.com/aliparsai/LittleDarwin`.
80. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Static analysis of model transformations. *IEEE Trans. Software Eng.*, 43(9):868–897, 2017.
81. D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for Java. In *Joint Meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, pages 297–298. ACM, 2009. See also `http://javalanche.org/`, `https://github.com/david-schuler/javalanche`.
82. S. Sen and B. Baudry. Mutation-based model synthesis in model driven engineering. In *Workshop on Mutation Analysis (Mutation)*, 2006.
83. S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *Theory and Practice of Model Transformations*, pages 148–164. Springer Berlin Heidelberg, 2009.
84. D. Shin, E. Jee, and D. Bae. Comprehensive analysis of FBD test coverage criteria using mutants. *Software and System Modeling*, 15(3):631–645, 2016.
85. Sirius. `https://www.eclipse.org/sirius/`. (last accessed in Oct. 2019).
86. T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: a CSP security example. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 340–350, 2003.
87. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2^{nd} Edition.* Addison-Wesley Professional, 2008.
88. M. Trakhtenbrot. New mutations for evaluation of specification and implementation levels of adequacy in testing of statecharts models. In *Workshop on Mutation Analysis (Mutation)*, pages 151–160, 2007.
89. J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer. Towards systematic mutations for and with ATL model transformations. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, 2015.
90. J. Tuya, M. J. S. Cabal, and C. de la Riva. SQLMutation: A tool to generate mutants of SQL database queries. In *Workshop on Mutation Analysis (Mutation)*, page 1, 2006.
91. J. Tuya, M. J. S. Cabal, and C. de la Riva. Mutating database queries. *Information & Software Technology*, 49(4):398–417, 2007.
92. M. Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013.
93. D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. 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*, pages 623–637. Springer, 2011.
94. F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke. Memory mutation testing. *Information & Software Technology*, 81:97–111, 2017.
95. A. Yazdani Seqerloo, M. J. Amiri, S. Parsa, and M. Koupaee. Automatic test cases generation from business process models. *Requirements Engineering*, 24(8):119–132, 2018.
96. L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *Proc. ASE*, pages 92–102. IEEE, 2013.

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 '+'	**modify one** PrefixExpression
	  where { operator **in** ['++', '−−'] }
	  with { operator = '+' }

Replaces a relational operator by '>'	**modify one** InfixExpression
	  where { operator **in** ['>=', '<', '<=', '==', '!='] }
	  with { operator = '>' }

Replaces a relational operator by '>='	**modify one** InfixExpression
	  where { operator **in** ['>', '<', '<=', '==', '!='] }
	  with { operator = '>=' }

Replaces a relational operator by '<'	**modify one** InfixExpression
	  where { operator **in** ['>', '>=', '<=', '==', '!='] }
	  with { operator = '<' }

Replaces a relational operator by '<='	**modify one** InfixExpression
	  where { operator **in** ['>', '>=', '<', '==', '!='] }
	  with { operator = '<=' }

Replaces a relational operator by '=='	**modify one** InfixExpression
	  where { operator **in** ['>', '>=', '<', '<=', '!='] }
	  with { operator = '==' }

Replaces a relational operator by '!='	**modify one** InfixExpression
	  where { operator **in** ['>', '>=', '<', '<=', '=='] }
	  with { operator = '!=' }

Replaces a conditional operator by '&&'	**modify one** InfixExpression
	  where { operator **in** ['||', 'ˆ'] }
	  with { operator = '&&' }

Replaces a conditional operator by '||'	**modify one** InfixExpression
	  where { operator **in** ['&&', 'ˆ'] }
	  with { operator = '||' }

Replaces a conditional operator by 'ˆ'	**modify one** InfixExpression
	  where { operator **in** ['&&', '||'] }
	  with { operator = 'ˆ' }

Replaces a logic operator by '&'	**modify one** InfixExpression
	  where { operator **in** ['|', 'ˆ'] }
	  with { operator = '&' }

Replaces a logic operator by '|'	**modify one** InfixExpression
	  where { operator **in** ['&', 'ˆ'] }
	  with { operator = '|' }

Replaces a logic operator by 'ˆ'	**modify one** InfixExpression
	  where { operator **in** ['&', '|'] }
	  with { operator = 'ˆ' }

Replaces a shift operator by '>>'	**modify one** InfixExpression
	  where { operator **in** ['>>>', '<<'] }
	  with { operator = '>>' }

Replaces a shift operator by '>>>'	**modify one** InfixExpression
	  where { operator **in** ['>>', '<<'] }
	  with { operator = '>>>' }

Replaces a shift operator by '<<'	**modify one** InfixExpression
	  where { operator **in** ['>>', '>>>'] }
	  with { operator = '<<' }

Replaces an assignment operator by '+='	**modify one** Assignment
	  where { operator **in** ['−=', '∗=', '/=', '%='] }
	  with { operator = '+=' }

Replaces an assignment operator by '-='	**modify one** Assignment
	  where { operator **in** ['+=', '∗=', '/=', '%='] }
	  with { operator = '−=' }

Replaces an assignment operator by '*='	**modify one** Assignment
	  where { operator **in** ['+=', '−=', '/=', '%='] }
	  with { operator = '∗=' }

Replaces an assignment operator by '/='	**modify one** Assignment
	  where { operator **in** ['+=', '−=', '∗=', '%='] }
	  with { operator = '/=' }

Replaces an assignment operator by '%='	**modify one** Assignment
	  where { operator **in** ['+=', '−=', '∗=', '/='] }
	  with { operator = '%=' }

Replaces an assignment operator by '&='	**modify one** Assignment
	  where { operator **in**
	    ['|=', 'ˆ=', '>>=', '>>>=', '<<='] }
	  with { operator = '&=' }

Replaces an assignment operator by '|='	**modify one** Assignment
	  where { operator **in**
	    ['&=', 'ˆ=', '>>=', '>>>=', '<<='] }
	  with { operator = '|=' }

Replaces an assignment operator by '≙'	**modify one** Assignment
	  where { operator **in** ['&=', '|='] }
	  with { operator = 'ˆ=' }

Replaces an assignment operator by '>>='	**modify one** Assignment
	  where { operator **in** ['>>>=', '<<='] }
	  with { operator = '>>=' }

Replaces an assignment operator by '>>>='	**modify one** Assignment
	  where { operator **in** ['>>=', '<<='] }
	  with { operator = '>>>=' }

Replaces an assignment operator by '<<='	**modify one** Assignment
	  where { operator **in** ['>>=', '>>>='] }
	  with { operator = '<<=' }

Replaces a number by a random number	**modify one** NumberLiteral **with** {tokenValue = **random**−int−**string**(0, 9)}

Deletes a call to a void method	**remove one** MethodInvocation
	  where { method→returnType→type
	    is typed PrimitiveTypeVoid }

Deletes a call to a non-void method	**remove one** MethodInvocation
	  where { method→returnType→type
	    not typed PrimitiveTypeVoid }

Replaces a call to a constructor by null	a = **select one** Assignment
	  where { rightHandSide
	    is typed ClassInstanceCreation }
	create NullLiteral **in** a→rightHandSide

Deletes an assignment	**remove one** Assignment **where** { rightHandSide **is typed** NumberLiteral }

Propagates an argument to a return statement	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}

Propagates an argument to the right operand of an infix expression	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 }

Propagates an argument to the left operand of an infix expression	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 }

Propagates an argument to an assignment	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 }

Continued on next column...

...continued from previous column

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 }

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 }

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 }

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 }

Replaces a boolean value by true	**modify one** BooleanLiteral
	  where { value = false }
	  with { value = true }

Replaces a boolean value by false	**modify one** BooleanLiteral
	  where { value = true }
	  with { value = false }

Replaces a number by 1	**modify one** NumberLiteral
	  where { tokenValue <> '1' }
	  with { tokenValue = '1' }

Replaces a return statement by null	rt = **select one** ReturnStatement **create** NullLiteral **in** rt→expression

Replaces a number by 0	**modify one** NumberLiteral
	  where { tokenValue <> '0' }
	  with { tokenValue = '0' }

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 }

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 }

Continued on next column...

...continued from previous column

Java mutation	Wodel code

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 }

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 }

Replaces a string literal by ‘’	**modify one** StringLiteral
	  where { escapedValue <> '' }
	  with { escapedValue = '' }

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 }

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 }

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 }

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 }

Deletes one statement	**remove one** Statement
	  where {
	    self not typed VariableDeclarationStatement }

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 }

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 }

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 }

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...

...concluded from previous column

Java mutation	**Wodel code**

Deletes a conditional statement	bl = **select one** Block
	  where { statements **is typed** IfStatement }
	if = **select one** IfStatement **in** bl→statements
	modify bl **with** { statements += if→thenStatement }
	remove if

Decrements 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
	dec = **deep clone** n **with** { tokenValue = '1' }
	create InfixExpression **in** e→leftOperand
	  with {
	    leftOperand = n,
	    operator = '−',
	    rightOperand = dec }

Decrements 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
	dec = **deep clone** n **with** { tokenValue = '1' }
	create InfixExpression **in** e→rightOperand
	  with {
	    leftOperand = n,
	    operator = '−',
	    rightOperand = dec }

Decrements number in a return statement	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 }

Decrements number in an assignment	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 }

B ATL mutation operators

This appendix contains the definition of the ATL mutation operators proposed in [89] using Wodel.

ATL mutation	**Wodel code**
Matched rule addition	**deep clone one** MatchedRule **with** { name = **random−string**(4, 6) }
Matched rule deletion	**remove one** MatchedRule
Matched rule name change	**modify one** MatchedRule **with** { name = **random−string**(4, 6) }
In pattern element deletion	**remove one** InPatternElement

In pattern element class change	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 }

Continued on next column...

...continued from previous column

ATL mutation	**Wodel code**
In pattern elem. name change	**modify one** InPatternElement **with** { varName = **random−string**(4, 6) }

Filter addition	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 }

Filter deletion	**remove one** OclExpression **where** { **container is typed** InPattern }

Filter condition change	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 }

Out pattern element addition	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}

Out pattern element deletion	**remove one** OutPatternElement

Out pattern element class change	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 }

Out pattern element name change	**modify one** OutPatternElement **with** { varName = **random−string**(4, 6) }

Binding addition	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) }

Binding deletion	**remove one** Binding

Binding value change	b = **select one** Binding
	  where { value **is typed** OperatorCallExp }
	create StringExp **in** b→value
	  with { stringSymbol = **random−string**(4, 6) }

Binding feature change	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 }

Continued on next column...

...concluded from previous column

ATL mutation	**Wodel code**