Generating Automatic Unit Tests of JavaScript Code from UML Class and Activity Diagrams

Agnieszka Malanowska[a] and Adrianna Małkiewicz-Błotniak
Warsaw University of Technology, Institute of Computer Science, Nowowiejska 15/19, Warsaw, Poland

Keywords: Automatic Test Generation, Unit Tests, JavaScript, UML, Class Diagram, Activity Diagram, Jest, StarUML, Test-driven Development.

Abstract: As testing phase plays a significant role in the software lifecycle, all facilitations that can speed up and automate this process seem to be very useful. One of the biggest group of approaches covers automatic test generation. In this paper, we describe our solution for fully automated unit test generation from UML class and activity diagrams. We have adapted and completely redesigned two algorithms from the literature. The first of them tests conformance of types of attributes and method return values between the class diagram and class implementation. The second one serves as a basis for testing all paths of the activity diagram. As a result, we generate tests in dynamically typed language, JavaScript, in the format required by Jest testing framework. We have implemented this approach in the extensible UML2Test tool, a plug-in to StarUML modeling environment. The tool generates complete executable unit tests from the UML model, so it can be used in conjunction with the test-driven development methodology. Usefulness of our approach and tool was successfully verified on the exemplary system for recruitment support.

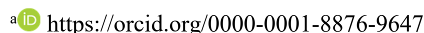
SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Software testing is a crucial activity necessary to ensure that the implemented software can be released without significant errors. Unfortunately, this process requires also much time and effort, so all facilitations that can speed up the testing phase of software lifecycle are worth a careful consideration. One category of such facilitations is test automation, among which automatic test generation plays a significant role. Quite often tests are generated on the basis of the UML (OMG, 2017) model.

There are various approaches to automatic test generation from UML diagrams. Testing compliance of the UML model and the source code is particularly important when the teams working on the design and the implementation are disjoint. Checking compatibility of types used in the implementation is especially vital in dynamically typed programming languages, where it is impossible to determine variable type before the code execution. It seems that among many test generation approaches, there is a lack of solutions suited for dynamically typed languages, such as JavaScript (JS). This fact and the popularity of the UML as a modeling language and JS as a programming language motivated our work on the UML-based JS tests generation.

In this paper, we present an approach to fully-automated generation of JS unit tests from the UML class and activity diagrams. To achieve this, we have modified two algorithms from the literature and combined them into one solution (Małkiewicz-Błotniak, 2020). The first algorithm we used is an adaptation of (Pires et al., 2008) method to generate JUnit tests which check the conformance of the class implementation and the class diagram. The second approach is based on the (Kurth et al., 2014) solution, in which test data to cover all paths on the activity diagram are obtained. Our modifications of algorithms led to generation of complete unit tests in JS, according to the syntax defined by Jest (Jest) testing framework. We have also implemented them in UML2Test tool (Małkiewicz-Błotniak, 2020), an extensible plug-in to StarUML (StarUML) modeling environment. The main novelty of this work is the adaptation of algorithms to support test generation in dynamically typed language, JS. Our contribution is also the new design of the test generator architecture



[a] https://orcid.org/0000-0001-8876-9647

Malanowska, A. and Małkiewicz-Błotniak, A.
Generating Automatic Unit Tests of JavaScript Code from UML Class and Activity Diagrams.
DOI: 10.5220/0010401401890196
In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pages 189-196
ISBN: 978-989-758-508-1
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

and combination of two independent algorithms into one useful tool, which was successfully verified on the exemplary system for recruitment support.

The rest of the paper is organized as follows. In Section 2, we briefly discuss the related work on UML-based test generation. Section 3 describes the original algorithms we have taken advantage of. In Section 4, there is an exhaustive description of our solution, including modifications of original algorithms and architecture of our tool. Section 5 presents the example of usage of this tool and results of experiments. Section 6 concludes the paper.

2 RELATED WORK

The topic of test generation is very wide and one of its many branches uses UML model as an input. Below we present only a small excerpt from the very rich literature on UML-based test generation. Different approaches exploit various types of the UML diagrams. Generated tests can also take many diverse forms. Although the most useful are automatically generated tests ready to be executed, such full solutions are not as common as one may expect.

One of the oldest publications on test generation from UML diagram is the one by (Offutt et al., 1999). The authors propose the UMLTest tool which generates system tests using UML state machine diagram as an input. Their tool does not generate fully automatic tests, but only test input data, test scenario and expected results. This approach covers 3 of 4 test coverage levels defined by the authors, i.e. transition coverage level, full predicate coverage level and transition-pair coverage level. The complete sequence coverage level was not implemented, as it requires additional information about the real use case of the software (Offutt et al., 1999).

Similar output is also generated by the UTG tool proposed by (Samuel et al., 2009) and implemented in Java. In this case, the activity diagram is used to prepare test data, test scenario and test postcondition. The authors use program slicing approach. They map the activity into the flow dependency graph and execute the edge marking algorithm. This approach covers all paths of the activity diagram. At least one test case for each path is obtained, boundary testing criterion is also applied.

Sequence and state machine diagrams serve as an input to the method and tool described by (Barisas et al., 2013). Again, only test input data and test scenarios for manual execution are generated. They cover integration testing level and their approach offers twofold method of test data generation. Input data are generated either randomly or using symbolic execution. One of the biggest drawbacks of this approach is that, although the generated tests should be as independent of the application code as possible, the tests cannot be obtained without the existing source code and have to be created again after introduction of changes in the code.

Quite different approach is presented by (Arora et al., 2020). The activity diagram serves as an input for test scenario generation. In this case, the modified ant colony algorithm is exploited. To overcome the difficulties resulting from this algorithm (long execution time and danger of local optimum selection), the authors have modified it and added the orientation factor. As this approach is based on the heuristic, it still takes quite long to obtain the results, but the authors indicate that it gives better results than traditional genetic algorithm and basic ant colony optimization. The described method can be used only for a concurrent part of the activity diagram.

3 ALGORITHMS USED

Our approach for JS automatic unit test generation is based on two algorithms from the literature. Here we present the original approaches and highlight some issues with them.

3.1 Testing Conformance of Class Definition with Class Diagram

The first algorithm that inspired our work was proposed by (Pires et al., 2008). The authors describe generation of fully automated design tests from the class diagram using Model Driven Architecture (MDA) approach (OMG, 2014a), implemented as an Eclipse plug-in. They claim to have succeeded in obtaining executable JUnit tests for simple design rules which check the conformance of Java classes and their design on the class diagram in terms of (Pires et al., 2008): attributes and method signatures (conformance of type names), generalization and association.

Only the first category of design tests is described in (Pires et al., 2008) and we have focused only on it. Although the tests that only check the compatibility of types of attributes and return values of methods between the class diagram the source code may not seem practically useful, especially when other automation mechanisms (e.g. UML-based code generation) are exploited, they turned out to be quite valuable in our case. Firstly, in the dynamically typed

language, there is no easy way to check the conformance of types in the implementation with expectations. Secondly, there are still many projects in which developers manually implement the designed architecture. Moreover, often the design and implementation are performed by the separate teams, so the probability of error introduction increases. It all convinced us that the idea of type conformance testing is worth a new adaptation.

(Pires et al., 2008) implement their approach using the MDA framework. They used existing UML metamodel published by OMG as a source and Java Abstract Syntax as a target Java metamodel available in Eclipse. The concrete models of those source and target metamodels are class diagrams and Java or JUnit code. The rules of transformation between the input and output model are defined using the ATL language. To access the information about the code structure, Pires et al. reuse their previously developed DesignWizard (DesignWizard) library, which offers an API to access the structure of the code by reading Java bytecode. The ATL transformation rule defined by (Pires et al., 2008) consists of names of the: transformation module, input and output metamodels (i.e. UML2 and Java Abstract Syntax), rule, source element from the input metamodel (i.e. Class) and target elements to which the source will be transformed (i.e. Java output). The authors present only the extract of the whole transformation rule.

At the beginning of the JUnit test generated by (Pires et al., 2008), the DesignWizard object is created on the basis of the indicated Java project. Then, specific class under test is obtained from the Java code using the DesignWizard object's API. Expected names and types of attributes and names and return values of methods are read from the class diagram and hardcoded in the string arrays. Then, two separate loops check the conformance of types for attributes and methods. In each case, the Java field or method representation is obtained by the DesignWizard object on the basis of the Java class name and the name of the class member defined in the UML model. Then, an assertion verifies whether the string representing the UML type of the given member is the same as the name of the member type obtained from the DesignWizard representation of the Java bytecode.

3.2 Testing Conformance of Method Implementation with Activity Diagram

The second approach we have taken advantage of is proposed by (Kurth et al., 2014). That paper describes a method of test data generation from the activity diagram with OCL (OMG, 2014b) constraints. The activity diagram is transformed into an AMPL (AMPL) program describing all control flow paths of the input diagram. Such program is then transferred to a constraint solver to obtain values of variables for each path of the original diagram, which form the test data. The authors implemented their approach in the tool called Activity Tester and claim that it can generate fully automated unit tests in C++. Moreover, this tool is said to be a part of the bigger Eclipse plug-in, Partition Test Generator (ParTeG). However, the paper (Kurth et al., 2014) focuses only on the UML-AMPL transformation and experiments with different constraint solvers. Generation of fully automatic C++ unit tests is not described there.

The input activity diagram supported by the (Kurth et al., 2014) transformation should represent one UML operation and can contain actions, control nodes and control flows. OCL constraints can be used for every action (as local postconditions, written in a note) and control flow (as guards). All attributes of the class owning the given method and all parameters of this method can be referred to in constraints. The constraints and properties and parameters used inside the constraints will be represented in the resulting AMPL model. The whole transformation is performed by means of symbolic execution.

As the authors of (Kurth et al., 2014) indicate, execution of each path of the activity diagram is modelled as a series of states. They define the activation set as a subset of all states in which the given action or control flow is being executed and indicate that there exists one such activation set for each action with a local postcondition and each control flow with a guard. OCL constraints are represented as AMPL constraints and can be enabled or disabled for each state (Kurth et al., 2014). Local postconditions of actions can use the `@pre` mark, which means that they refer to the value of the given variable in the previous state. Guards describe only the relationship between variables within a state. Only integer, float and boolean variables are supported in this approach.

The resulting AMPL code consists of declarations of parameters and variables, specification of constraints and activation sets for the constraints (Kurth et al., 2014). The authors also point out that boundary values of variables are much more useful for testing purposes than the others, so they add the linear objective function to the AMPL model to enforce boundary values detection (Kurth et al., 2014). The authors also introduce an early elimination of infeasible paths (Kurth et al., 2014),

which uses the modification of the depth-first search (DFS) algorithm. Feasibility of the given path is verified by execution of the constraint solver on it. There are also special parameters of the pruning algorithm: number of steps of the algorithm performed without feasibility checking, maximum desired path length and maximum number of test cases.

The paper (Kurth et al., 2014) mainly describes the transformation from the activity diagram to the AMPL program, an interface common for the majority of constraint solvers. The AMPL model is used as an input to the solver (selected for specific problem nature), which is expected to generate values of variables satisfying conditions specified on all paths of the diagram. Those values form the test data. Successful experiments with different types of problems and solvers are reported by (Kurth et al., 2014), but they are performed manually – no automation of constraint solver usage is presented.

4 UML2Test

We have redesigned the algorithms described in Section 3 and preserved only the core ideas of the original methods. Those adaptations of two distinct algorithms were also integrated into one tool, UML2Test (Małkiewicz-Błotniak, 2020). It is a plug-in to the StarUML (StarUML) modeler, which has a JS API. Tests generated by UML2Test are prepared for the Jest (Jest) testing framework and can be executed fully automatically. As the generated tests check the conformance of implementation with the UML model, both the design and implementation have to exist for the tests to be successfully created and executed. However, to only generate tests without running them, one does not need an existing implementation of the designed system. It allows for usage of our solution in a test-driven development approach.

Decisions made during the adaptation of original algorithms result from the selected development environment and availability of JS libraries. As our modifications of methods are closely related to the UML2Test architecture, we firstly describe the latter and then present the former.

4.1 Architecture

UML2Test consists of three main modules. The first one loads the project from the StarUML modeler and transforms it to the internal representation defined for our tool. The second module is responsible for test code generation. It prepares textual representation of files with JS tests compatible with Jest framework. Both modules offer easy extensibility, as they implement simple interfaces and the rest of the implementation refers only to those interfaces. The third module is used to generate test cases and is necessary only for the algorithm based on (Kurth et al., 2014) approach. Test cases store input data and expected output for all paths of the activity diagram. The module responsible for test case generation employs a constraint solver (to find test data and expected results, as in (Kurth et al., 2014)) and a parser (to parse OCL constraints from the activity diagram). All modules are managed by the main class of the plug-in, called `UML2Test`. The latter is run by the StarUML environment using its public API. The high-level illustration of UML2Test architecture is presented in Figure 1.

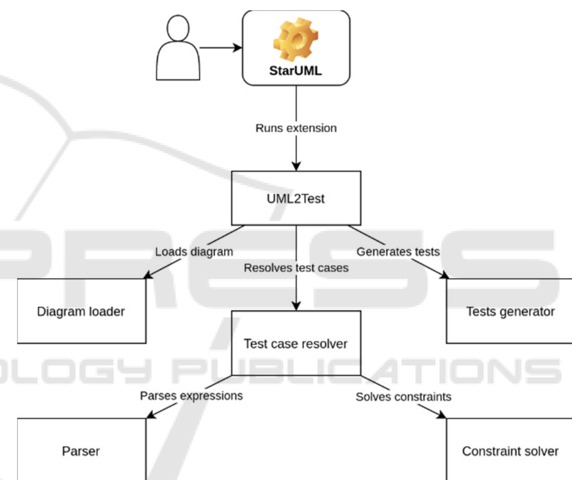


Figure 1: UML2Test architecture.

4.2 Modifications of Algorithm based on Class Diagrams

The approach described by (Pires et al., 2008) is based on the MDA architecture and uses a specific DesignWizard tool to perform code inspection of Java classes statically. Our architecture, as well as the target language, differs completely from theirs. We only reuse their idea that from the class diagram tests checking conformance of types can be generated. We perform test generation for a dynamically typed language in accordance with UML2Test control flow.

We assume that the input class diagram is correct and only five JS primitive types (`string`, `number`, `boolean`, `bigint`, `object`) can be used on this diagram. In the first stage, StarUML representation of the whole project is read by UML2Test and is

transformed into our internal representation. The transformation uses Lodash (Lodash) library as a functional paradigm helper. JS test generator is executed on this internal representation and prepares the textual output, which then will be saved in the test files. Test generation itself consists of three stages and is repeated for each class. All members of a given class are considered, also those inherited from superclasses, as all generalization hierarchy is traversed during test generation.

Firstly, the class under test from the source code is imported into the unit test file. Then, for each attribute which has type defined in the class diagram, a separate test is created. Unlike (Pires et al., 2008), we test the code written in a dynamically typed language, so types have to be determined in runtime. Thus, in every test an instance of the tested class is created. In the second line of each test there is a comparison of types. We write an assertion to check if the type of the implemented attribute is the same as in the UML model. The type from the UML model is read from our internal project representation and hardcoded in the test code. The actual attribute from implementation is obtained dynamically from the previously created instance of the class under test. To check its type, we use JS `typeof` operator.

The last stage of class test generation is similar. Tests are generated for each method, for which the return type was defined and is a primitive JS type. Again, the type returned by the implemented method can be obtained only dynamically, so the method needs to be executed within the test. An instance of the class is created in every such test. Moreover, to execute a method, correct arguments need to be provided. We achieve this by defining mock-ups of all parameters. The only worth information is the type of the return value, so we can pass any values of arguments of expected types. We randomly select values of parameters using Chance (Chance) library. As the tests have to be repeatable, we define a constant seed at the first usage of this library. The result of method execution is assigned to a variable. Finally, the assertion checks whether the type of this variable is compliant with the hardcoded type obtained from the class diagram.

4.3 Modifications of Algorithm based on Activity Diagrams

Our adaptation of the method proposed by (Kurth et al., 2014) is also very different from the original. The aforementioned paper describes only test data generation, while we propose a fully automatic approach to unit tests generation. Here we assume the correctness of the input diagrams again. The constraint solver we use, Constrained (Constrained), supports only integer variables, so we assume that all variables in the input activity diagram are integers. We also expect that the initial node of the diagram is related to a note with a method signature. As in the original solution (Kurth et al., 2014), all actions can have notes with their local postconditions, possibly with `@pre` marks, and guards can be placed on control flows. All constraints should be written according to the specified grammar (Małkiewicz-Błotniak, 2020) for the subset of OCL language. Test conditions are extracted automatically from the activity diagram.

Similarly to the adaptation described in the previous section, we transform the StarUML project into our own internal representation. Then, test cases, which will be stored in test files, are obtained. Here we use our adaptation of (Kurth et al., 2014) method. Firstly, it is necessary to generate all possible paths in the graph representing the input activity diagram. We use Graphlib (Graphlib) library to perform operations on the graph. All paths are found using the modified version of DFS algorithm. The authors of the original method also used modified DFS, but their modification focused on infeasible path elimination (Kurth et al., 2014), while our version of DFS ensures that all of the paths are found, even if they have some nodes in common. We achieve this by marking currently processed node on the stack as unvisited after successfully finding a path.

Once all paths are obtained, values of test data (i.e. parameters of the method, referenced attributes of the enclosing class and expected return value) for each path are generated. They are calculated with the help of the constraint solver, we have chosen a JS tool named Constrained (Constrained). This solver has some limitations, e.g. it supports only integer variables, but has also a great advantage – it always finds boundary values satisfying given condition, so no separate boundary value analysis is necessary. Before using the solver, the string representations of constraints are parsed. We have defined a simplified grammar of a subset of OCL language, which can be found in (Małkiewicz-Błotniak, 2020). The parser for this grammar is implemented with the help of Nearley (Nearley) library and employs a lexer defined using Moo (Moo) library. To analyze abstract syntax trees, we use functional paradigm utilities provided by the Lodash (Lodash) library. All new variables and constraints are added to the Constrained solver. As this tool supports only non-strict inequalities, we have added a transformation that allows for usage of all types of inequalities on the input diagram. Strict

inequality symbols are replaced with their equivalents which are true for integers (as only they can be used on the diagram), i.e. > symbol is replaced with '>= 1 +' string and < symbol is changed to '<= -1 +'. <> symbol, used to express that two values are not equal, is also defined – both rules for < and > operators are applied. When all constraints on a given path are parsed and transferred to the solver, the latter is used to find values of variables that satisfy all conditions on the path. The return value for this path is also calculated, as it is just one of the variables specified in OCL constraints.

Test data for all paths are then delivered to the JS test generator, which prepares textual representation of the test code. Firstly, there is an import of the JS class owning the tested method. Then, separate tests for each path are created. As those tests check the behavioral properties of the system, the method needs to be invoked on an instance of the tested class. Thus, every test begins with creating such an instance. Then, all parameters (including attributes of the given class) are assigned values generated by the constraint solver. Finally, an assertion checks whether the value returned by the invocation of method on the created instance with the specified parameters is equal to the return value generated by the solver. At the end, the string representation of the tests is saved in a test file.

Similarly to the authors of the original approach (Kurth et al., 2014), we use the constraint solver to generate values of parameters for each path. Unlike them, we do not transform the diagrams into AMPL model, but we pass the constraints directly to the selected solver. We analyze the graph of activity with the different modification of DFS than this used by (Kurth et al., 2014). We have formally defined a specific simplified OCL grammar and substitution rules for some operators and built lexer and parser to read the textual constraints. Although we guess that Kurth et al. also must have used some parser, there is no description of that in their paper (Kurth et al., 2014). Moreover, we generate fully executable tests, instead of test data only. Therefore, although the main idea remains the same as in the original approach, we have redesigned and reimplemented it completely, with our own assumptions and improvements.

5 EVALUATION ON EXAMPLE

To verify the usefulness of our adaptation of algorithms and UML2 Test tool, we have designed an exemplary system supporting recruitment process in a company. Due to the lack of space, we cannot present the whole model and results here, but they can be found in (Małkiewicz-Błotniak, 2020). Unfortunately, we have no access to real-world UML projects data, so we were unable to validate this tool on the industrial example.

The purpose of the exemplary system is to automate tasks related to recruitment tests and accountancy. As a recruitment support, the system estimates the level of knowledge of a candidate and calculates the proposed salary. It is also responsible for counting the value of the raise for an employee.

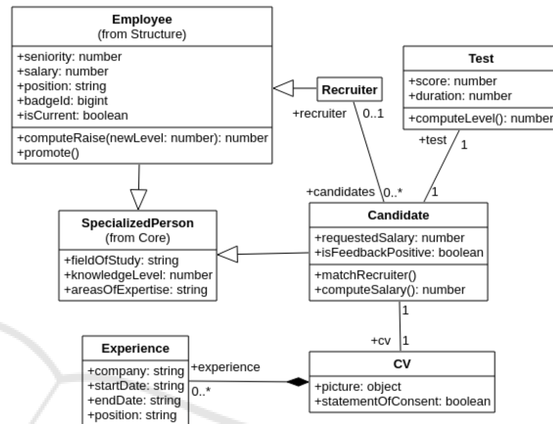


Figure 2: Class diagram for the recruitment support system.

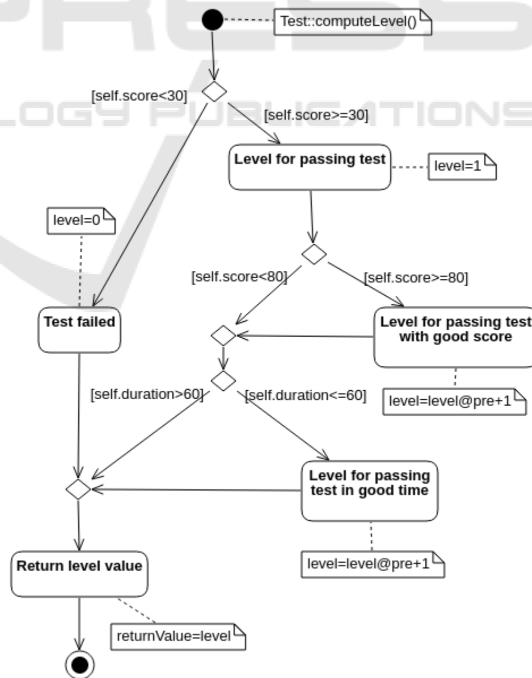


Figure 3: Activity diagram for `computeLevel` method.

The StarUML project for this system consists of three models: Core (basic classes), Structure (of a company) and Recruitment (candidates and

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

recruitment data). The project contains three class diagrams (13 classes in total) and three activity diagrams. The latter were designed for methods that operate on integers, as only those are supported by our solution. For brevity, we present here only the extracts of the whole project and generated tests. As an example, the class diagram for the Recruitment model is presented in Figure 2. It is a typical class diagram, we just assume that only specific JS primitive types can be used. Tests which check conformance of the class implementation with its design are very similar for each class. Below we present three exemplary tests generated for the `Employee` class.

```
   test("checks seniority attribute to
have number type", () => {
     const instance = new Employee();
     expect(typeof
instance.seniority).toBe("number");
   });
   test("checks fieldOfStudy attribute
to have string type", () => {
     const instance = new Employee();
     expect(typeof
instance.fieldOfStudy).toBe("string");
   });
   test("checks computeRaise operation
return value to have number type", ()
=> {
     const instance = new Employee();
     const mockedNewLevel =
7347197741891584;
     const result =
instance.computeRaise(mockedNewLevel);
     expect(typeof
result).toBe("number");
   });
```

The test file contains a set of tests dynamically checking the conformance of types of attributes between the diagram and implementation. They are generated for the owned attributes of the `Employee` class (first test above), attributes inherited from superclasses (second test), return values of the owned methods (third test) and inherited methods. If the method has some parameters, they are mocked with random values. UML2Test generated 13 test files using this algorithm, as there are 13 classes in the project. Each class contains in average 3 fields or methods and in this case, total time necessary for test generation was 12 ms. As a result, 70 simple unit tests (stored in 13 files) were created.

Figure 3 presents the activity diagram for `computeLevel` method. This operation is responsible for assigning level of knowledge to the candidate on the basis of the recruitment test score and time. The initial node of the diagram contains a note with method signature. All actions that cause a change of variable values also have corresponding notes. The `@pre` mark, indicating the value from the previous state, is used. We refer to attributes of the class owning the method under test (i.e. `Test` class) using the `self` keyword. Local variables, such as `level` and `returnValue`, also appear on the diagram. It is worth noting that the activity diagram shown in Figure 3 presents a method which operates only on class attributes, but diagrams describing methods which exploit both external parameters and internal attributes of the enclosing class were also successfully tested, what can be found in (Małkiewicz-Błotniak, 2020).

As can be seen from the diagram, there are five possible paths of execution of the `computeLevel` method. For every path, a corresponding test was automatically generated. Tests for two first paths of this method look as follows.

```
   test("checks path #1 for method
computeLevel", () => {
     const instance = new Test();
     instance.score = 29;
expect(instance.computeLevel()).toBe(0)
;
   });
   test("checks path #2 for method
computeLevel", () => {
     const instance = new Test();
     instance.score = 80;
     instance.duration = 60;
expect(instance.computeLevel()).toBe(3)
;
   });
```

In each test, an instance of `Test` class is created, its attributes which appear on the given path are assigned values obtained from the constraint solver and an assertion is invoked on the expected and actual return values. Due to the usage of the Constrained (Constrained) solver, the attributes used within the diagram are assigned boundary values in the test code. In case of this algorithm, 3 test files were generated, as there were 3 input activity diagrams. It took 125 ms to create all 41 individual unit tests. Although the time required for this algorithm to complete is much bigger than for the first one, it is still unnoticeable for a human. It suggests that the results can be obtained in a reasonable time even for projects with many more diagrams.

All unit tests generated by UML2Test can be executed using Jest (Jest) testing framework. However, if the tests are to be successfully run, a

ready implementation of the corresponding JS classes has to exist. On the other hand, the implementation is necessary *only* to run the tests, what allows for usage of test-driven development approach.

6 CONCLUSIONS

Testing process is one of the most important phases of the software lifecycle and each improvement in it is worth a consideration. In this paper, we have presented our solution for fully automatic unit tests generation. The tests are created on the basis of the UML class and activity diagrams and written in JS, according to the rules defined by Jest (Jest) testing framework. To achieve this, we have modified two already existing algorithms. The first one (Pires et al., 2008) was intended to generate JUnit tests checking conformance of types of attributes and method return values between class design and implementation. The second, proposed by (Kurth et al., 2014) and based on the usage of the constraint solver, generates test data for all paths from the activity diagram.

We have redesigned the original methods in order to generate executable unit tests in a dynamically typed language, JS. We decided to use this language, as testing conformance of types for dynamically typed language is much more valuable than for those typed statically. Such testing is also particularly useful in cases when design and implementation are prepared by separate teams. Our solution is implemented as an extensible and easy to use StarUML (StarUML) plug-in, called UML2Test (Małkiewicz-Błotniak, 2020). The approach presented in this paper was verified on the exemplary system supporting the recruitment process with promising results.

The most serious limitation of the approach described here is the fact that only integers can be specified on the input activity diagram. The other drawback is the lack of industrial evaluation of the approach caused by the lack of the industrial data. As the UML2Test generates simple unit tests checking the conformance of class member types and covering all paths of the methods, no additional constraints, such as pre- or postconditions referring to the state of the whole system are generated. Finally, some disadvantages of the approach can be caused by the fact that we rely on the UML models as inputs. Although the UML has some drawbacks, e.g. diagrams can be interpreted differently or used only partially, its main advantages are popularity and ease of understanding. In the future, we are planning to perform experiments with other tools, e.g. constraint solvers, as well as further modifications of the algorithms or additions of the new ones. For instance, tests checking conformance of types could be extended to cover the types of class attributes coming from association relationships.

REFERENCES

All URLs were valid on 21.11.2020

Arora, V., Singh, M., Bhatia, R., 2020. Orientation-based Ant colony algorithm for synthesizing the test scenarios in UML activity diagram. *Inf Softw Technol*. Vol. 123. 1–21. DOI: 10.1016/j.infsof.2020.106292.

Barisas, D., Bareisa, E., Packevicius, S., 2013. Automated Method for Software Integration Testing Based on UML Behavioral Models. In *Communications in Computer and Information Science*. Vol. 403. Springer. 272-284. DOI: 10.1007/978-3-642-41947-8_23.

Kurth, F., Schupp, S., Weißleder, S., 2014. Generating Test Data from a UML Activity Using the AMPL Interface for Constraint Solvers. In Seidl M., Tillmann N. (eds) *Tests and Proofs. TAP 2014*. Springer. 169-186. DOI: 10.1007/978-3-319-09099-3_14.

Małkiewicz-Błotniak, A., 2020. *Generating tests based on UML models* [BSc thesis]. Warsaw University of Technology, Institute of Computer Science (in Polish).

Offutt, J., Abdurazik, A., 1999. Generating tests from UML specifications. In *Intl Conf on UML*. Springer. 416-429. DOI: 10.1007/3-540-46852-8_30.

OMG, 2014. Model Driven Architecture (MDA): MDA Guide rev. 2.0.

OMG, 2014. Object Constraint Language: Version 2.4.

OMG, 2017. Unified Modeling Language: Version 2.5.1.

Pires, W., Brunet, J., Ramalho, F., 2008. UML-based design test generation. In *SAC '08: Proc of 2008 ACM symposium on Applied computing*. ACM. 735-740. DOI: 10.1145/1363686.1363859.

Samuel, P., Mall, R., 2009. Slicing-based test case generation from UML activity diagrams. *ACM SIGSOFT SEN*. Vol. 34. No. 6. ACM. 1-14. DOI: 10.1145/1640162.1666579.

AMPL, https://ampl.com/.

Chance, https://chancejs.com/.

Constrained, https://github.com/Wizcorp/constrained.

DesignWizard, https://github.com/joaoarthurbm/designwizard.

Graphlib, https://github.com/dagrejs/graphlib.

Jest, https://jestjs.io/.

Lodash, https://lodash.com/.

Moo, https://github.com/no-context/moo.

Nearley, https://nearley.js.org/.

ParTeG, http://parteg.sourceforge.net/.

StarUML, http://staruml.io/.