Merging Modeling and Programming using Umple

Timothy C. Lethbridge, Vahdat Abdelzad, Mahmoud Husseini Orabi, Ahmed Husseini Orabi, Opeyemi Adesina

University of Ottawa, Canada
{tcltcl, vabde040, mhuss092, ahuss045, oades013}@uottawa.ca

Abstract. We discuss how Umple merges modeling and programming by adding modeling constructs to programming languages and vice-versa. Umple has what we call model-code duality; we show how it fulfills key attributes of being both a programming language and a modeling technology. Umple also has what we call text-diagram duality in that the model or code can be updated by editing the textual or diagram form. We give an example of Umple, and explain how key benefits of textual programming languages are found in Umple, as are important features of modeling technology.

Keywords: Modeling. Programming languages. Duality. State machines. Associations.

1 Introduction

Umple is an open-source software development technology designed to fully merge modeling and programming [1]. Umple is motivated by the vision that model-driven development will become ubiquitous in the not-too-distant future, but that this can only be realized if the best capabilities of modeling and programming are blended, while eliminating the biggest drawbacks of each.

Section 2 of this paper gives an overview of how Umple implements a duality between modeling and code, and between diagrams and text. It also briefly discusses various features of Umple and the uses to which it has been put. Section 3 gives an example of Umple to help the reader understand the concepts explained in the paper. Section 4 gives an overview of the strengths obtained by being able to operate on an Umple system in its programming-language perspective; we suggest that modeling tools in general should have a programming-language perspective in order to benefit from these sorts of strengths. Section 5 overviews the capabilities Umple exhibits as a modeling tool. Section 6 highlights the evidence for Umple's effectiveness, while Section 7 provides the conclusion.

Preprint of: Lethbridge, T.C., Abdelzad, V., Husseini Orabi, M., Husseini Orabi, A., Adesina, O. (2016).
Merging Modeling and Programming Using Umple.
In: Margaria, T., Steffen, B. (eds)
Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications.
ISoLA 2016. Lecture Notes in Computer Science(), vol 9953. Springer
https://doi.org/10.1007/978-3-319-47169-3_14

2 Perspectives of Umple

2.1 Duality: Model-code and text-diagram.

Umple, which has been under development since 2007. incorporates what we call *model-code duality*: An Umple system looks and feels like code when viewed in a programming environment, but looks and feels like a model when viewed in a modeling environment.

By *looking and feeling like code*, we mean that,

c1) The system, or parts of it, are composed of a set of units (files in the case of Umple), which can be edited using a text editor supporting syntax highlighting.

c2) The textual syntax is designed to be usable by programmers.

c3) When it is processed (compiled in the case of Umple), feedback such as warnings and errors is produced, highlighting issues on specific lines.

By *looking and feeling like a model*, we adopt Ludewig's criteria [2]. We summarize these as follows:

m1) There is a *mapping* between the model and the system being modeled, or part of it. The system is called the 'original' by Ludewig.

m2) This mapping *abstracts* some properties of the system, hence providing a simplified view. Typical abstractions focus on behavioural properties or structural properties, but the same model may include both, as well as other types of abstractions.

m3) The model is *useful* in that it one can do things with the model instead of having to have access to the full (executable) system. Key things one can do with a model under m3 include analyzing it to measure it or to find defects, and transforming it into other forms. Models are therefore useful in early stages of design, but in some cases can also be used to generate some or all of the system.

Model-code duality applies to a technology when all six of the above criteria apply simultaneously. It can be argued that all programming languages to some extent meet criteria m1 to m3; for example, a C++ program abstracts from the much more complex machine code and can be analyzed to find various kinds of defects. The more abstract and non-procedural (declarative) an abstraction is, however, the more strongly it would seem that model-code duality applies.

Umple is designed so it can be treated exactly like any programming language, with a syntax that follows characteristics of C-family languages. Criteria c1 to c3 clearly apply [7]. Umple also incorporates abstractions commonly considered to be at the modeling level such as UML associations, state machines and patterns; it also provides measurement, defect-analysis and transformation capabilities for these. Thus, m1 to m3 also clearly apply. We argue in this paper that in the long term, model-code duality should apply increasingly strongly to all software development technologies, and part of the objective of Umple is to show a possible path towards this future vision.

Nothing in Ludewig's criteria states what the concrete syntax of a model has to look like; therefore text should be just as good as diagrams. In fact, Umple is designed such that various diagram views (e.g. class diagrams, state diagrams, composite structure diagrams) can be used as the concrete syntax by which the developer explores and edits the system. We call this *text-diagram duality*.

Umple therefore goes beyond what typical software development technology provides: It is common to be able to extract a diagram from textual code (as a reverse-engineering operation), or to generate code from a UML diagram. *Round-tripping* may also often be employed in various other technologies, wherein extraction and regeneration cycles occur. However, for true text-diagram duality, there should be no need for round-tripping. This is the case in Umple: its model/code abstract syntax can simultaneously be viewed and edited in textual or diagram form.

Model-diagram duality does not preclude the possibility that not all aspects of the model have or routinely use a diagram representation: For example, Umple embeds pure algorithmic methods; it does not provide equivalent flowcharts as their usefulness is questionable. In Umple, therefore, while much of the model/code can be viewed or edited diagrammatically or textually, there are portions (algorithms, constraints, identifier labels) that require textual editing.

2.2 Components and functions of the Umple technology

A key component of Umple is its compiler whose input language consists of a blend of textual modeling constructs (including associations [3] [4], state machines [5], patterns and many more) and code in programming languages such as Java, PhP and C++. The compiler analyses the model, giving feedback to the software engineer as to correctness, as would any compiler. It then generates complete systems in any of its input programming languages (i.e. Java, PhP, C++). A developer used to any of these languages would therefore see Umple just like any other programming language – in fact, she or he might perceive of Umple as a pre-processor, simply extending these programming languages with additional abstractions.

In addition to generating an executable system, Umple also can generate SQL database code, Formal methods code (e.g. Alloy) [6], and other modeling syntaxes such as XMI. Umple can also generate metrics and various forms of analysis.

In addition to being a transform engine, Umple also has development environment components, allowing its model diagrams or text to be edited. UmpleOnline [7] is its web-based environment, principally useful for small-scale systems – theoretically unlimited, but practically up to about 1000 lines of Umple code. Larger Umple systems are better manipulated using IDEs like Eclipse or else command-line technology.

2.3 Uses of Umple

Umple has been put to use in two contexts: The first is model-driven development of significantly-sized software, and the second is the teaching of software engineering in general, and modeling specifically.

It is being used in several universities for teaching modeling. Students, who are typically adept at programming, come to see the value of modeling, and develop deeper skills in modeling, when they can work with a tool that allows them to see a system from either a modeling or programming perspective [8] [9]

The largest system so far build in Umple (that we are aware of) is Umple itself. After a bootstrapping period, the initial Java version of Umple was re-written in Umple. Until 2016, there remained a few non-Umple component in Umple; most notably Jet was used in code generation. However as of early 2016 even the Jet code was replaced with Umple's native UmpleTL templating code.

The use of Umple to develop itself serves as a proof that the vision of blending modeling and programming is indeed possible; it also provides a large case study and research testbed to explore how to move this vision forward into practical use.

3 A small executable example of Umple

An extensive set of examples of Umple can be found online, including in Umple-Online [7], and in the Umple GitHub site [10]. Examples can also be found in many of the papers we will cite in this paper.

However, to aid understanding we present a small sample of Umple code below:

```
1   class VehicleModel {
2     name;  // attribute, defaults to String
3     Integer modelYear;
4   }
5
6   class CityVehicle {
7     * -- 1 VehicleModel; // association
8     vin;
9     trace vState record vin; // umple trace sublanguage
10    vState { // start of state machine
11      Active { // first state
12        sell -> Sold; // transition
13        scrap -> Scrapped;
14        confirmStolen -> Stolen;
15        BeingAcceptanceTested { // substate of Active
16          accepted -> InService;
17          unacceptable -> Returned;
18        }
19        InService {
20          InUse {
21            routineMaintenaceDue -> UnderMaintenance;
22            accident -> UnderMaintenance;
23          }
24          UnderMaintenance {
25            maintenanceComplete -> InUse;
26          }
27        }
28      }
29      Retired {
```

```
30        Sold {}
31        Scrapped {}
32      }
33      Returned {} // one of several end states
34      Stolen {}
35    }
36
37    public static void main (String[] args) { // standard Java
38      VehicleModel m1 = new VehicleModel("AX7 Digger", 2015);
39      VehicleModel m2 = new VehicleModel("E25 Truck", 2016);
40      CityVehicle v1 = new CityVehicle("AD13743",m1);
41      CityVehicle v2 = new CityVehicle("GT29754",m2);
42      CityVehicle v3 = new CityVehicle("GT31974",m2);
43      v1.unacceptable();
44      v2.accepted();
45      v3.accepted();
46      v2.accident();
47      v3.sell();
48      v2.maintenanceComplete();
49      v2.confirmStolen();
50    }
51  }
```

We assume the reader has some understanding of UML, the widely used modeling language, but for clarity: an *association* is a UML abstraction that expresses the fact that there is to be a run-time relationship between instances of particular classes. Notations called *multiplicities* constrain how many instances of the class at one end of the association can be related to an instance at the other end, and vice-versa. The multiplicity '*' means 'many' or 'any number.' Umple enforces *referential integrity* in associations: If class A has an association with class B, and if an instance of A is linked to an instance of B, then that instance of B is also linked to the instance of A.

State machines in Umple follow standard UML semantics, each describing the state of instances of a particular class. The state machine can be considered an attribute whose values are the states. *Events* (method calls) are the only way to cause changes in the state of the machine.

The above code shows two classes (lines 1-4 and 6-51 respectively). The classes are linked by an association (line 7). CityVehicles go through a lifecycle represented using a state machine (lines 10-35). Lines 37-50 are ordinary Java, embedded in the Umple, used to instantiate objects and take the objects through their lifecycles. Line 9 is a sample of Umple's tracing capability [11], instructing Umple to output details as the system executes; the output is in Figure 3.

When loaded into UmpleOnline, various diagrams appear: a class diagram in Figure 1 and a state diagram in Figure 2. These diagrams will change as the code is edited. The system can be compiled using Eclipse or the command line. The resulting Java, which amounts to about 800 lines, is designed to be readable for teaching and inspection purposes, but never needs to be edited or read since the original Umple is the 'gold master' of the system, and any modifications can be applied there.

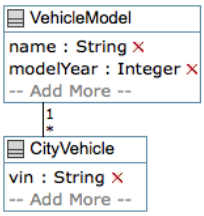

Figure 1: Class diagram of the example system; in UmpleOnline this diagram can be edited, with changes reflected in the Umple code, or the text can be edited with changes reflected in this diagram. Other presentation friendly views are available too.

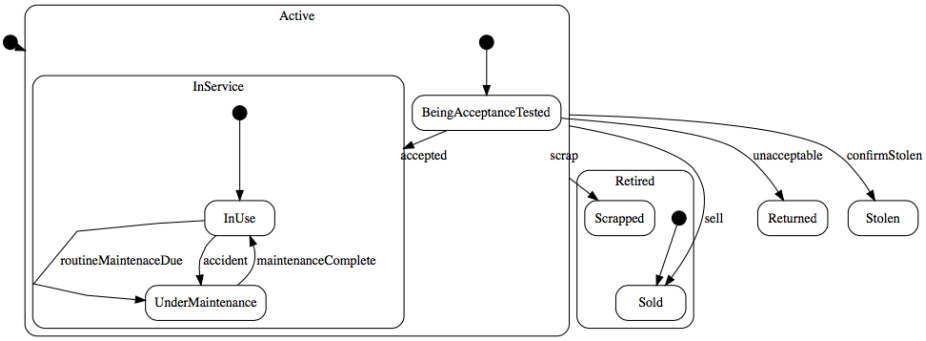

Figure 2: State diagram of the example system that Umple dynamically updates.

An Umple system is compiled and executed just like one might compile and execute a Java system or C++. Umple generates Java (or other programming languages) as intermediate languages. But the Umple compiler is designed to hide this from the developer, in the same manner as the developer does not need to be involved with the Java bytecode files (.class files).

When the above system is executed, the output shown in Figure 3 is produced by the tracing mechanism (to save space in the paper, only the last 2 digits of the time and object identifiers have been kept). These lines describe the various state transitions of the various objects.

```
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
87,1,CityVehicle.ump,9,CityVehicle,54,sm_t,BeingAcceptanceTested,exitActive,Null,AD13743
88,1,CityVehicle.ump,9,CityVehicle,54,sm_t,BeingAcceptanceTested,unacceptable,Returned,AD13743
89,1,CityVehicle.ump,9,CityVehicle,35,sm_t,BeingAcceptanceTested,accepted,InService,GT29754
89,1,CityVehicle.ump,9,CityVehicle,62,sm_t,BeingAcceptanceTested,accepted,InService,GT31974
89,1,CityVehicle.ump,9,CityVehicle,35,sm_t,InUse,accident,UnderMaintenance,GT29754
89,1,CityVehicle.ump,9,CityVehicle,62,sm_t,InService,exitActive,Null,GT31974
89,1,CityVehicle.ump,9,CityVehicle,62,sm_t,InUse,exitActive,Null,GT31974
89,1,CityVehicle.ump,9,CityVehicle,62,sm_t,Active,sell,Sold,GT31974
89,1,CityVehicle.ump,9,CityVehicle,35,sm_t,UnderMaintenance,maintenanceComplete,InUse,GT29754
89,1,CityVehicle.ump,9,CityVehicle,35,sm_t,InService,exitActive,Null,GT29754
```

```
89,1,CityVehicle.ump,9,CityVehicle,35,sm_t,InUse,exitActive,Null,GT29754
89,1,CityVehicle.ump,9,CityVehicle,35,sm_t,Active,confirmStolen,Stolen,GT29754
```

Figure 3: Trace of state transitions of the example system

4 The programming language perspective of Umple

Part of Umple's vision is to incorporate a set of strengths, listed below, that are generally attributable to modern programming languages. We highlight places where Umple's incorporation of these, provides advantages over other modeling tools.

Flexible textual IDE support: As with any other mainline programming language, developers using Umple have the ability to use powerful IDEs such as Eclipse or Microsoft Visual Studio, or any of the wide variety of standalone text editors, for tasks like syntax-directed editing and debugging. Umple has an Eclipse plugin, but does not have any dependencies on Eclipse, unlike most competing modeling platforms – this allows potentially greater scope of use. Plugins for various other tools are also available for Umple.

Configuration management: Umple gives the ability to take full advantage of the power of Git and tools like GitHub for versioning, code inspection and differencing. Umple is open-source and hosted on GitHub. Unlike other modeling tools, Umple's programming-language nature makes versioning and merging of models transparent. Competing modeling tools either use awkward XML-based differencing and versioning, or require complex special-purpose algorithms [12].

Manipulability of text: As a language that looks like any other programming language, Umple gives programmers the ability to work fluidly with command-line tools of the developer's choice, including Vim, Grep, AWK, and many others.

Separation and combination of concerns: Researchers have evolved several technologies designed to help separate concerns or weave concerns or features together in textual languages. Umple natively supports three of these

- **Mixins**: Mixins in Umple allow files to be combined in a variety of ways, to build various different products from the same source code base. Different mixins can contribute elements to a given class or state machine.
- **Traits**: Traits are a powerful capability finding their way into various languages [13]. A trait can be seen as building on the power of the interface, in that implementation can be injected into multiple classes. Traits are fundamentally a textual construct. Umple is the first technology to allow traits to work at the modeling level, injecting pieces of model (such as state machine fragments) into other models. More details on Umple traits can be found here [14].
- **Aspects**: Umple has a basic aspect technology for injecting code before or after various join points using pattern-matching pointcuts.

All three of the above work together with each other.

Text-generation templates: Umple incorporates text-generation template technology as found in languages such as PhP and technologies such as Xpand [15]. Umple has its own best-of-breed sublanguage for this called UmpleTL, which is central to Umple's own generation of itself. This replaced Jet, a now-deprecated Eclipse technology, which was the original template language in which Umple was bootstrapped. In fact any user of Jet can make use of an application written in Umple to convert Jet to Umple [16].

Legacy and libraries: Umple, as with many other programming languages, gives the ability to work with legacy code frameworks. Umple, in fact works fluidly with legacy or library code in Java, C++ and PhP. It can embed or call APIs in any of these languages, and its modeling constructs generate these languages. In fact, Umple can serve as a medium for cross-language development, since Umple code can simultaneously blend with more than one of these languages.

Testing: Fundamental to modern programming methods is the ability employ test-driven development, including use of xUnit tools such as Junit and CppUnit. The Umple system has an extensive multi-level test suite and is developed using test-driven development [17].

Compatibility with open source methods: Umple gives accessibility to programmers worldwide as it enables adoption of tools and techniques standard in the open-source community, including those listed above [18].

5 The modeling perspective of Umple

The other side of Umple's code-model duality vision is to leverage the following strengths of modern model-driven development:

High-level abstractions. The abstractions below supported by Umple have long been a part of requirements and design methods, but have required translation into code. Umple brings these directly to the programming language level. The abstractions provide concise ways of expressing design concepts that would be much more complex and repetitive if written using classic code. Additionally, the presence of these in the source allows the compiler to do high-level analysis to find defects that otherwise might be hidden. Examples of the latter include detecting violations of constraints, and unreachable states.

- **UML attributes**: Attributes in Umple are more than just variable declarations; they have richer semantics including being subject to constraints, and patterns such as immutability. More details can be found here [19].
- **UML associations**: Umple's supports the rich feature set of UML associations; multiplicity and referential integrity are enforced in the running system. More details can be found here [3] and an example is found in section 3.

- **State machines**: Umple supports the sophisticated semantics of state machines, including nesting, orthogonal regions, concurrent activities in separate threads, and so on. More details are here [5].
- **Constraints**: Umple supports a core subset of OCL constraints, service as class invariants, method preconditions and transition guards.
- **Components and structural modeling**: Umple has been extended to incorporate distributed features such as components, ports, and connectors [20]. It supports the core features of UML related to composite and component structural modeling.
- **Patterns**: Umple has various built-in notions such as immutable and singleton that provide capabilities which in a programming environment would require following a formulaic 'pattern'.

Diagrams: As discussed earlier, Umple provides class diagrams, state diagrams and composite structure diagrams that are always in full agreement with the source code view, with and the diagrams can be edited to update the textual source. In our experience developers most often find it easier to edit text, and sometimes easier to edit diagrams. Likewise developers most often find it easier to inspect diagrams for defects, but also find different kinds of defects by inspecting the textual form.

Transformations. Model transformations have been central to the modeling community. Umple has numerous built-in model transformations [6], such as from Umple to formal languages Alloy and nuXmv; to programming languages, and to diagrams and tables of various kinds. Since there is a built-in transformation to Ecore, transformations designed to work with Ecore can also be used. Some of the transformations represent ongoing research and are thus still under development, with various limitations. For example, the transformation to nuXmv [26] currently transforms only state machines, simple attributes and simple Boolean expressions.

6 Evidence for Umple's effectiveness

Umple is, first and foremost, proof of its own effectiveness. Essentially every feature of Umple, as it has been developed, as been rolled into improvements to the Umple compiler and supporting tools.

Over 60 developers (full list with the Umple MIT license [21]), mostly 4th-year undergraduates working for 4 months at a time, plus 8 PhD students and a few masters students, have been able to rapidly develop the technology. Its maintainability is witnessed by the ability of students to rapidly understand the system, and make contributions within 4-month semester windows. Developers have been able to not only use the best capabilities of textual tools, but have also been able to navigate Umple's self-descriptive metamodel, which is available online and is hyperlinked to online code [22]. Umple's quality has been maintained, as proven by the fact that it is always been required that it compiles itself, and that over 6000 tests always execute with 100% pass rate.

Studies have shown that the Umple language is usable by developers [23] [24] [25] – moreover it is more usable than plain traditional code, and models in Umple's textual form are as usable as in the UML diagrammatic form.

Umple has also proved beneficial at teaching modeling in the classroom [9]. Studies have shown that grades and comprehension improve after Umple is introduced.

7 Conclusions

In this paper, we have described the Umple technology and demonstrated that it has what we call model-code duality. This means that it has the advantages of both code (being textual in addition to diagrammatic) and model (incorporating declarative abstractions for views of a system), and can be used as a programming or modeling language interchangeably.

We anticipate that tools like Umple will become the norm. Our experience is that Umple helps developers develop systems more effectively, since they can work at the abstract level and always see diagrammatic representations of their code without reverse engineering. Also Umple helps students learn to model and to develop systems faster.

Acknowledgements

We would like to thank the following for supporting the development of Umple over the years: NSERC through Discovery Grants, the Ontario Research Fund (ORF), and IBM who supported much of the early development of Umple. We would also like to acknowledge the over-60 open-source developers of Umple.

References

1. Umple.org, Model-Oriented Programming, htttp://www.umple.org
2. Ludewig, J.: Models in Software Engineering – an introduction. Softw. Syst. Model (2003), 2:5-14
3. Forward, A. (2010): The Convergence of Modeling and Programming: Facilitating the Representation of Attributes and Associations in the Umple Model-Oriented Programming Language, PhD thesis, University of Ottawa, http://www.site.uottawa.ca/~tcl/gradtheses/aforwardphd/
4. Badreddin, O., Forward, A., and Lethbridge, T.C. (2013), Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity, SERA 2013 http://dx.doi.org/10.1007/978-3-319-00948-3_9
5. Badreddin, O., Lethbridge, T.C., Forward, A., Elasaar, M. Aljamaan, H, Garzon, M. (2014), Enhanced Code Generation from UML Composite State Machines, MODELSWARD 2013, Portugal
6. Adesina, O.: Integrating Formal Methods with Model-Driven Engineering, Models 2015 Doctoral Symposium, http://ceur-ws.org/Vol-1531/paper2.pdf
7. UmpleOnline http://try.umple.org

8. Lethbridge, T.C. (2014), Teaching Modeling Using Umple: Principles for the Development of an Effective Tool, CSEET 2014, Klagenfurt Austria, IEEE Computer Society
9. Lethbridge, T., Mussbacher, G, Forward, A. and Badreddin, O., (2011) Teaching UML Using Umple: Applying Model-Oriented Programming in the Classroom, CSEE&T 2011, pp. 421-428. DOI: 10.1109/CSEET.2011.5876118
10. Github Umple: https://github.com/umple/umple
11. Aljamaan, H., Lethbridge T.C. (2015) MOTL: a Textual Language for Trace Specification of State Machines and Associations, Cascon 2015, ACM
12. Badreddin, O., Lethbridge, T.C. and Forward, A. (2014), A Novel Approach to Versioning and Merging Model and Code Uniformly, MODELSWARD 2014, Portugal.
13. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: a mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)
14. Abdelzad, V, and Lethbridge, T.C. (2015) Promoting Traits into Model-Driven Development, Software and Systems Modeling, Springer. DOI:10.1007/s10270-015-0505-x
15. Xpand: https://eclipse.org/modeling/m2t/?project=xpand
16. JetToUmpleTL: https://github.com/umple/JETToUmpleTL
17. Badreddin, O., Forward, A., and Lethbridge, T.C. (2014), A Test-Driven Approach for Developing Software Languages, MODELSWARD 2014, Portugal
18. Badreddin, O., Lethbridge, T.C. and Elassar, M. (2013), "Modeling Practices in Open Source Software", OSS 2013, 9th International Conference on Open Source Systems.
19. Badreddin, O. Forward, A., and Lethbridge, T.C. (2013), "Exploring a Model-Oriented and Executable Syntax for UML Attributes", SERA 2013, Springer, Springer SCI 496, pp. 33-53. http://dx.doi.org/10.1007/978-3-319-00948-3_3
20. Husseini Orabi, M., Husseini Orabi,A., Lethbridge, T.C. (2016) "Umple as a component-based language for the development of real-time and embedded applications", Modelsward 2016
21. Github Umple: Open-Source License for the Umple Model-Oriented Software Technology https://github.com/umple/umple/blob/master/LICENSE.md
22. Github Umple: Umple Metamodel: http://metamodel.umple.org
23. Badreddin, O., Forward, A., and Lethbridge, T. (2012), Model Oriented Programming: An Empirical Study of Comprehension, Cascon, ACM.
24. Badreddin, O. and Lethbridge, T. (2012) Combining Experiments and Grounded Theory to Evaluate a Research Prototype: Lessons from the Umple Model-Oriented Programming Technology, 2012 First International Workshop on User evaluation for Software Engineering Researchers (USER 2012), in conjunction with ICSE 2012, pp 1-4. DOI: 10.1109/USER.2012.6226575.
25. Aljamaan, H., Model-Oriented Tracing Language: Producing Execution Traces from Tracepoints Injected into Code Generated from UML Models, PhD Thesis, University of Ottawa, 2015, https://www.ruor.uottawa.ca/handle/10393/33419
26. Adesina, O., Lethbridge, T., and Somé, S., (2016) A fully automated approach to discovering non-determinism in state machine diagrams, 10th Int. Conf. On the Quality of Information and Communications Technology, Portugal