From Formal Results to UML Model
A MDA Tracing Approach

Vinícius Pereira[1], **Rafael S. Durelli**[2], **Márcio E. Delamaro**[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) — São Carlos – SP – Brazil

[2]Departamento de Ciência da Computação (DCC)
Universidade Federal de Lavras (UFLA) — Lavras – MG – Brazil.

{vpereira,delamaro}@icmc.usp.br, rafael.durelli@dcc.ufla.br

Abstract. *The Unified Modeling Language (UML) is the de-facto industrial standard for modeling object-oriented software systems. Nevertheless, the UML has some limitations such as: (i) ambiguity, (ii) semantic unclear, and (iii) lack of formal semantics. To deal with this, researchers propose UML transformations for formal models. These models are precise but difficult to be analyzed by people with no knowledge of formalism. This paper describes a model-driven approach which enable traceability of counterexamples from formal results back into UML Model. The traceability is made via a mapping between existing elements in the formal results and the elements present in the UML model. Using our approach, it is feasible to analyze and understand the formal results even without a thorough knowledge of formal methods, and consequently fix the UML model where needed.*

1. Introduction

Nowadays UML (Unified Modeling Language) is widely used among professionals from different areas of computer science [Hutchinson et al. 2011]. However, a system modeled by means of just UML is not complete in the sense of missing relevant information such as formal notation. For example, considering a car-collision avoidance system (CCAS), one can use UML to model such system, however, it is not possible to accurately describe situations where the brakes are triggered automatically. This occurs because the UML does not have a semantic free of ambiguities – i.e., UML does not have a formal notation. One possible way to fix this drawback in UML is using a temporal logic. Therefore, CCAS could be modeled using formal notation, free from ambiguity and with a clear semantics.

A formal model is almost the opposite model to UML. As formal models do not have the UML problems, they are more difficult to create and reuse than UML models. Furthermore, the developer must be a formalism expert to create, read, and understand a formal model. Consequently, it is more common to find developers working with UML than developers dealing with formal models. For this reason, over the years, researchers seek ways to unite the strengths of both formal models and UML. Usually the formalization of UML semantics found in the literature uses only one type of diagram [Eshuis 2006, Lund and Stolen 2006, Bouabana-Tebiel 2009, Micskei and Waeselynck 2011]: (i) State Diagram or (ii) Activity Diagram.

These researchers provide solutions that involve different formal techniques, but can be summarized by the following process: (i) it is necessary to formalize semantically one or more UML diagrams, (ii) these diagrams are then converted to a formal model, (iii) a model checker formally checks this formal model, and (iv) a analysis is performed and the results are provided by the model checker. Although they are valid approach, them assume that the developer has sufficient knowledge of formalism to interpret the formal results, analyze them and make the necessary corrections in the UML model, and then repeat this whole process. However, if the developer has sufficient knowledge to perform this kind of analysis, he probably would use the formal model directly, instead. To the best of our knowledge, up to this moment, there is no research concentrated on analyzing and understanding the formal results even without a thorough knowledge of formal methods, and consequently fix the UML model where needed.

Therefore, this paper presents a generic Model-Driven Architecture (MDA) that defines a meta-model for the traceability of formal results within the UML model regardless of the formalization used in the UML model. By using this approach, the developer might not possess knowledge in formalism (or have little knowledge) and yet he will be able to follow step-by-step the formal results given by the model checker. The main contributions of this paper are threefold: (i) we show a new meta-model for describe formal results, (ii) we demonstrate the feasibility of our meta-model by implementing it as two DSLs (Mapping and Trace) in the Eclipse environment, (iii) we also show a complete example of how to use our meta-model and its DSLs.

2. Related Studies

Studies that formalize the UML semantics generally propose formalization of only one UML diagram, like State Diagram or Activity Diagram [Forster et al. 2007, Kaliappan and Konig 2012]. Other studies formalize two or more types of diagrams [Konrad et al. 2004, Broy et al. 2006], where can be cited more specifically the proposals of [Graw et al. 2000] and [Baresi et al. 2012] that formalize four and five diagrams, respectively. However, these proposals assume that the user can read and understand the formal results. mas All cited studies can have their formalization of UML semantics processes divided into three distinct stages: (i) **Modeling**, wherein the system to be developed is modeled using UML and the semantic concepts of each proposal; (ii) **Transformation**, wherein the UML model (and the assigned semantics) are transformed into a formal model according to the syntax and semantics of the formal language chosen by the proposal; and (iii) **Verification**, wherein the formal model is analyzed by a model checker that shows the formal results of this verification.

The difference between these studies and the MDA proposed in this paper is that the latter seeks a viable way to represent the information of the formal results (Traceability stage) regardless of which UML diagram was formalized. Also, there are two major details: the MDA represents the formal results in the semi-formal system model (UML model) itself without creating other UML diagrams for the results; and it might be used with any kind of formalization of UML semantics.

3. MDA Tracing Approach

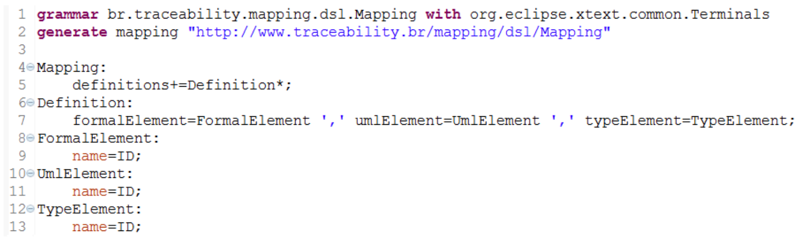
Initially, to make it possible to represent the formal results within the UML model, two artifacts are required: (i) UML Model itself; and (ii) Formal Results, created by the model checker responsible for analyzing the formal model. Using this two artifacts, the approach has available both the formal environment and the UML semi-formal graphical environment. However, the Transformation stage (see Section 2) is very particular to each type of formalization. To overcome this problem, we defined a third artifact called UML Mapping [Pereira et al. 2015] that properly connect both environments.

Our UML Mapping is compose of three elements: (i) ID-UML, a unique identifier to each UML element; (ii) ID-Formal, the identifier which is given to each UML element when the UML model is transformed to a formal model; and (iii) typeElement, which holds the element type at the UML model (e.g. Class, State, Message, etc.) Using the ID-UML, it is possible to identify the equivalent formal element. Similarly, using the ID-formal, it is also possible to identify the UML element by their respective ID-UML. We argue that as any UML graphical element owns an ID-UML in the UML Editor, then this mapping can be applied to any formalized UML diagram.

The UML Mapping is a file generated by an interface that need to be used by the transformation tool of a formalization of UML semantics process. The interface creates a ".mapping" file which respect a Domain-specific Language (DSL) called Mapping DSL. Our MDA then uses another DSL (named as Trace) to parse the Formal Results – together with the Mapping DSL – to represents the information inside the UML Model. In the next subsections we discuss both DSL.

3.1. Mapping DSL

The Mapping DSL grammar defines the elements that need to exist inside the mapping file. Each line of the file contains a tuple with three elements: ID-UML, ID-Formal, and typeElement. The Figure 1 shows the grammar defined for the Mapping DSL. The grammar shows that a Mapping might have N Definition (line 5). Each Definition holds a tuple with the three elements – *one* of each – separate by a comma (line 7). Finally, each element – FormalElement, UMLElement, and TypeElement – has its name (lines 9, 11, and 13).



```
 1 grammar br.traceability.mapping.dsl.Mapping with org.eclipse.xtext.common.Terminals
 2 generate mapping "http://www.traceability.br/mapping/dsl/Mapping"
 3
 4 Mapping:
 5     definitions+=Definition*;
 6 Definition:
 7     formalElement=FormalElement ',' umlElement=UmlElement ',' typeElement=TypeElement;
 8 FormalElement:
 9     name=ID;
10 UmlElement:
11     name=ID;
12 TypeElement:
13     name=ID;
```

Figure 1. Mapping DSL Grammar

By using the Xtext framework[1], our grammar generate a plug-in to Eclipse IDE[2]. Then, the interface builds the mapping file with the tuples every time that a Transformation stage is performed, and the Eclipse checks if the mapping file respect the Mapping grammar.

3.2. Interface

Our MDA depends of an interface that create the mapping file during the Transformation stage. This stage is the only moment in the formalization process where both environments – UML and formal model – exists together. Our interface is a piece of Java code

[1]https://eclipse.org/Xtext/
[2]https://eclipse.org/

which intercepts the UML element being transformed and write on a file its ID-UML, its typeElement (Class, State, Message, Transition, etc.), and the ID-Formal that is given to it. Algorithm 1 shows a pseudo code of our interface.

```
Input: UMLModel umlModel, File mapping
1  begin
2      foreach $StateDiagram\ std$ in $umlModel.getStateDiagrams()$ do
3          foreach $State\ state$ in $std.getStates()$ do
4              FormalState fstate = new FormalState(state);
5              Predicate pred = fstate.getPredicate();
6              mapping.write(pred.getName() + “,” + state.getUmlId() + “,” +
               state.getType());
7          end
8          foreach $Transition\ trans$ in $std.getTransitions()$ do
9              FormalTransition ftrans = new FormalTransition(trans);
10             Predicate pred = ftrans.getPredicate();
11             mapping.write(pred.getName() + “,” + trans.getUmlId() + “,” +
               trans.getType());
12         end
13     end
14 end
```

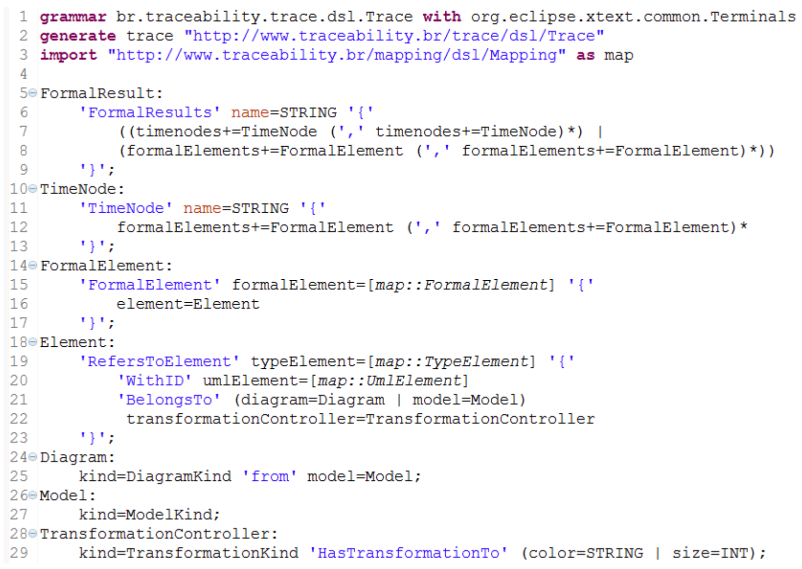
Algorithm 1: Interface - Getting data for Mapping

The core idea at Algorithm 1 is collecting the required data from State Diagrams for our mapping. During the Transformation stage, our method gather the three required data for each element inside the State Diagram (Lines 2 to 13). At Line 2, the method iterate with each State Diagram present in the UML model being formalized. In Lines 3 to 7, our method manipulate all states inside a State Diagram. At Line 3, our method gather each State in the State Diagram. Then, in Line 4 the method instantiate a $FormalState$ – a semantic version of State – with the given state. $FormalState$ depends on the type of formalization being used. Line 5 instantiate a $Predicate$ which holds the formal information for the state. Finally, the data are written in the mapping file, as can be seen in Line 6, in order to be used later in our MDA. A similar process is done with all transitions inside a State Diagram (Lines 8 to 12). The same logic is applied for Class, Object, Sequence, and Interaction Overview Diagrams.

3.3. Trace DSL

This DSL imports the Mapping DSL to use its tuples grammar inside the Trace grammar. Different from the Mapping DSL, which is not used directly by the user, the Trace DSL is written by the user. The Trace grammar defines what happens to each Formal Element present at the Formal Result. The Figure 2 shows the Trace grammar without the enum types.

The FormalResult might have a name and be a collection of TimeNode or FormalElement (lines 5 to 9). A TimeNode has its own name and a collection of FormalElement (lines 10 to 13). At FormalElement occurs the first use of the Mapping DSL. At line 15, after the use of a keyword, the formalElement variable receives a $map :: FormalElement$. The map makes reference to the Mapping DSL imported at line 3 and $FormalElement$ is the reference to the formal element defined at Mapping DSL. Then at line 16, the grammar shows that a FormalElement has one Element.



```
1  grammar br.traceability.trace.dsl.Trace with org.eclipse.xtext.common.Terminals
2  generate trace "http://www.traceability.br/trace/dsl/Trace"
3  import "http://www.traceability.br/mapping/dsl/Mapping" as map
4
5  FormalResult:
6      'FormalResults' name=STRING '{'
7          ((timenodes+=TimeNode (',' timenodes+=TimeNode)*) |
8          (formalElements+=FormalElement (',' formalElements+=FormalElement)*))
9      '}';
10 TimeNode:
11     'TimeNode' name=STRING '{'
12         formalElements+=FormalElement (',' formalElements+=FormalElement)*
13     '}';
14 FormalElement:
15     'FormalElement' formalElement=[map::FormalElement] '{'
16         element=Element
17     '}';
18 Element:
19     'RefersToElement' typeElement=[map::TypeElement] '{'
20         'WithID' umlElement=[map::UmlElement]
21         'BelongsTo' (diagram=Diagram | model=Model)
22          transformationController=TransformationController
23     '}';
24 Diagram:
25     kind=DiagramKind 'from' model=Model;
26 Model:
27     kind=ModelKind;
28 TransformationController:
29     kind=TransformationKind 'HasTransformationTo' (color=STRING | size=INT);
```

Figure 2. Trace DSL Grammar

The Element has the other two element of the Mapping tuple. First, at line 19 the typeElement variable receives the $TypeElement$ from the map – the import alias for the Mapping DSL. Then, at line 20 the $UmlElement$ from map is also assigned to a variable. At lines 21-22 the Element is associated to a Diagram or a Model and the TransformationController is called. For the UML, all elements must have a Diagram associated to it (lines 24-25). On the other hand, a BPMN (Business Process Model and Notation) element do not need a Diagram, so it's associated with a Model. The Trace DSL has validators to verified if the correct elements are associated with their right Diagram or Model. Finally, then enum types are: (i) DiagramKind – all the diagrams supported by the MDA; (ii) ModelKind – all models supported by the MDA; and (iii) TransformationKind – all transformation model-to-model supported by the MDA.

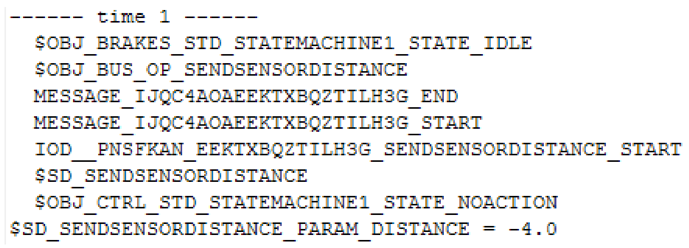
By using the Trace DSL, the user might defines which formal elements he wants to see inside the UML. One could also defines the type of transformation of each element such as color transformation (background, line, and font).

4. Example

The CCAS example is used to illustrate the use of our MDA. In this case, the UML model has the MADES UML semantic [Baresi et al. 2012] and it was converted to a LISP[3] model – the formal model to the MADES approach. We used the MADES UML because it is the approach that formalize most UML diagrams so far. The CCAS examples is one example from the MADES Project[4]. Given a property that holds (or not) at the formal model, then a formal result is generated by Zot Model Checker [Pradella 2009]. Figure 3 shows a short example of a formal result from Zot.

As can be seen in Figure 3 the formal result is not easy to read and understand. One need to identify the problem in the formal result, find the UML element equivalent to the LISP code, and then re-factor the UML model to try to fix the problem. Our MDA

[3]https://common-lisp.net/
[4]http://www.mades-project.org/

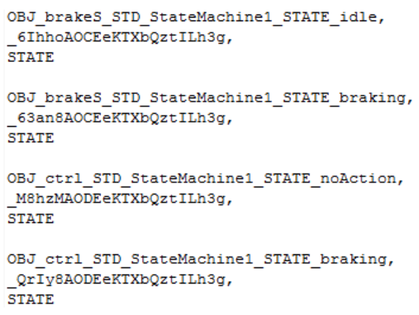


```
------ time 1 ------
  $OBJ_BRAKES_STD_STATEMACHINE1_STATE_IDLE
  $OBJ_BUS_OP_SENDSENSORDISTANCE
  MESSAGE_IJQC4AOAEEKTXBQZTILH3G_END
  MESSAGE_IJQC4AOAEEKTXBQZTILH3G_START
  IOD__PNSFKAN_EEKTXBQZTILH3G_SENDSENSORDISTANCE_START
  $SD_SENDSENSORDISTANCE
  $OBJ_CTRL_STD_STATEMACHINE1_STATE_NOACTION
$SD_SENDSENSORDISTANCE_PARAM_DISTANCE = -4.0
```

Figure 3. Formal Result from Zot

approach aims facilitate the user, by "executing" the formal results inside the UML model. In this context, user might see the step-by-step execution and find more easily and rapidly where to re-factor the model.

Our MDA approach uses both DSL as the Platform Independent Model (PIM) and an ATL (ATL Transformation Language)[5] to perform a Model-to-Model (M2M) transformation. The Figures 4 and 5 show both DSL – Mapping and Trace, respectively – instances for the CCAS example.



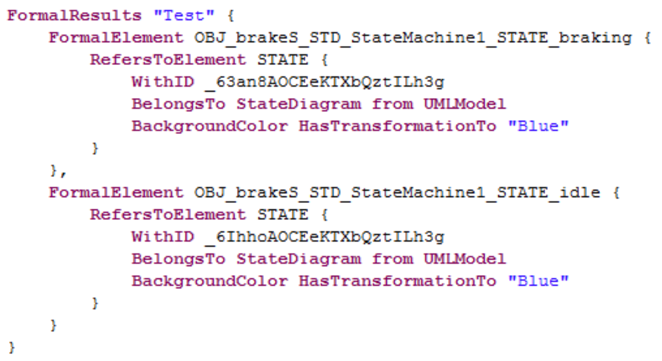
```
OBJ_brakeS_STD_StateMachine1_STATE_idle,
_6IhhoAOCEeKTXbQztILh3g,
STATE

OBJ_brakeS_STD_StateMachine1_STATE_braking,
_63an8AOCEeKTXbQztILh3g,
STATE

OBJ_ctrl_STD_StateMachine1_STATE_noAction,
_M8hzMAODEeKTXbQztILh3g,
STATE

OBJ_ctrl_STD_StateMachine1_STATE_braking,
_QrIy8AODEeKTXbQztILh3g,
STATE
```

Figure 4. Mapping DSL generated by interface



```
FormalResults "Test" {
    FormalElement OBJ_brakeS_STD_StateMachine1_STATE_braking {
        RefersToElement STATE {
            WithID _63an8AOCEeKTXbQztILh3g
            BelongsTo StateDiagram from UMLModel
            BackgroundColor HasTransformationTo "Blue"
        }
    },
    FormalElement OBJ_brakeS_STD_StateMachine1_STATE_idle {
        RefersToElement STATE {
            WithID _6IhhoAOCEeKTXbQztILh3g
            BelongsTo StateDiagram from UMLModel
            BackgroundColor HasTransformationTo "Blue"
        }
    }
}
```

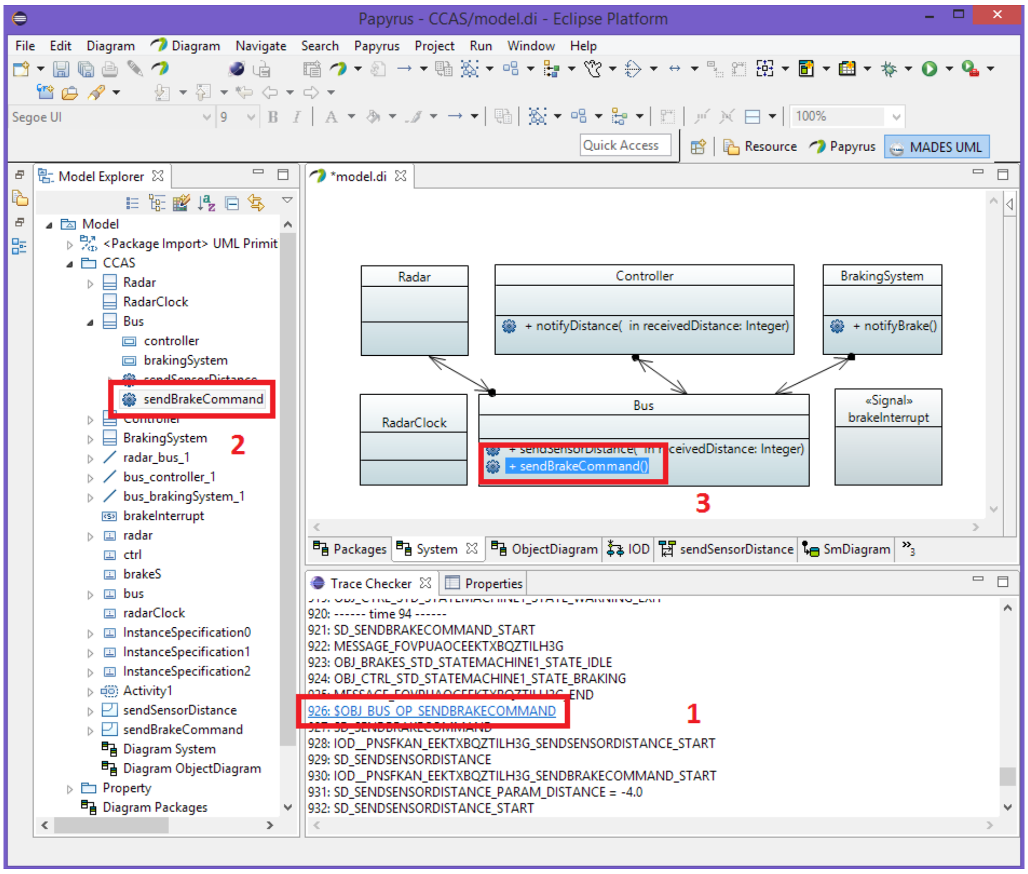
Figure 5. Trace DSL written by user

After obtain the PIM instance, a M2M transformation is performed following the ATL code written to parse the PIM through a Platform Specific Model (PSM). The PSM reflects the modeling environment (e.g. UML, BPMN, etc.). So the PIM that compose our MDA is generic enough to represent any type of formal result – given the formalization process uses our interface. The use of a PSM for each modeling environment allows the transformation of a generic formal result to a specific graphic model.

Finally, with the PSM our MDA code is generated as a Eclipse plug-in and the user could use it for execute the formal result and see the data-flow inside the original model – the UML in this example. The Figure 6 shows an Eclipse plug-in running our MDA with the Zot formal results and the MADES UML model.

Figure 6 highlights the following boxes: (1) Formal Results View – that presents to user the formal result and where the user could click at each formal element to see its correspondent UML element; (2) Model View – where the user could see the hierarchy of the UML element; and (3) Editor View – the graphical representation of the model where the user might edit the UML element.

[5]https://eclipse.org/atl/



```
920: ------ time 94 ------
921: SD_SENDBRAKECOMMAND_START
922: MESSAGE_FOVPUAOCEEKTXBQZTILH3G
923: OBJ_BRAKES_STD_STATEMACHINE1_STATE_IDLE
924: OBJ_CTRL_STD_STATEMACHINE1_STATE_BRAKING
925: MESSAGE_FOVPUAOCEEKTXBQZTILH3G_END
926: $OBJ_BUS_OP_SENDBRAKECOMMAND
927: SD_SENDBRAKECOMMAND
928: IOD__PNSFKAN_EEKTXBQZTILH3G_SENDSENSORDISTANCE_START
929: SD_SENDSENSORDISTANCE
930: IOD__PNSFKAN_EEKTXBQZTILH3G_SENDBRAKECOMMAND_START
931: SD_SENDSENSORDISTANCE_PARAM_DISTANCE = -4.0
932: SD_SENDSENSORDISTANCE_START
```

Figure 6. Plugin example

5. Conclusion

This paper presented a generic model-driven approach that is able to represent formal results back – from model checkers – inside the UML model. The MDA manage to do this by using a PIM which composed by two DSL. The DSLs define the structure of our mapping and a language to write which formal element the user might want to see at UML and which transformation it will have. The paper describes how both DSL were defined, the requirements to use it, as well as an example using MADES UML formalization as the test environment. Our MDA approach aims to fill the gap in how to trace formal results back into UML model. We achieve this by guiding the user through the analysis of formal results generate by model checkers. This enhances the users' understanding level about the results and thereby one can find possible defects more easily, fixing them and improving the UML model.

Although we have presented a example using MADES UML, our approach is generic enough to work with different formalization types of UML semantics, due to the way that our Mapping DSL works. In addition, the our approach works with every type of UML diagram since the transformation tool uses our mapping interface and the UML Editor provides access to ID-UML. As a future work, we aim to: ((i)) improve our approach and its plugin; (ii) carry out case studies and experiments; and (iii) analyze the possibility of using the our approach with other modeling languages, such as SysML and BPMN.

Acknowledgements

Vinicius Pereira would like to thanks the financial support provided by CAPES (DS-7902801/D) and CNPq (245715/2012-6)

References

Baresi, L., Morzenti, A., Motta, A., and Rossi, M. (2012). Towards the UML-based formal verification of timed systems. In *FMCO'12*, volume 6957 of *LNCS*, pages 267–286. Springer Berlin/Heidelberg.

Bouabana-Tebiel, T. (2009). Semantics of the interaction overview diagram. In *IRI'09*, pages 278–283, Piscataway, NJ, EUA. IEEE Press.

Broy, M., Crane, M. L., Dingel, J., Hartman, A., Rumpe, B., and Selic, B. (2006). 2nd UML 2 semantics symposium: formal semantics for UML. In *MoDELS'06*, pages 318–323. Springer-Verlag.

Eshuis, R. (2006). Symbolic model checking of UML activity diagrams. *ACM TOSEM*, 15:1–38.

Forster, A., Engels, G., Schattkowsky, T., and Straeten, R. V. D. (2007). Verification of business process quality constraints based on visual process patterns. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 197–208.

Graw, G., Herrmann, P., and Krumm, H. (2000). Verification of UML-based real-time system designs by means of cTLA. In *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 86–95.

Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011). Empirical assessment of MDE in industry. In *ICSE'11*, pages 471–480.

Kaliappan, P. and Konig, H. (2012). On the formalization of UML activities for component-based protocol design specifications. In *SOFSEM'12*, volume 7147 of *LNCS*, pages 479–491. Springer Berlin-Heidelberg.

Konrad, S., Cheng, B. H. C., and Campbell, L. (2004). Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992.

Lund, M. S. and Stolen, K. (2006). A fully general operational semantics for UML 2.0 sequence diagrams with potencial and mandatory choice. In *FM'06*, volume 4085 of *LNCS*, pages 380–395.

Micskei, Z. and Waeselynck, H. (2011). The many meanings of UML 2 sequence diagrams: a survey. *Software and Systems Modeling*, 10:489–514.

Pereira, V., Baresi, L., and Delamaro, M. E. (2015). Mapping formal results back to uml semi-formal model. In *Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS'15)*, volume 2, pages 320 – 329. SCITEPRESS – Science and Technology Publications.

Pradella, M. (2009). An user's guide to zot. Disponível em: *http://home.dei.polimi.it/pradella/Zot/*.