ACM Copyright Notice

Published in: ***Proceedings of ACM/IEEE International Conference on Model Driven Engineering Languages and Systems - Demo Track (MODELS'19),*** September 2019

"UCAnDoModels: A Context-based Model Editor for Editing and Debugging UML Class and State-Machine Diagrams"

Cite as:

P. Pourali and J. M. Atlee, "UCAnDoModels: A Context-Based Model Editor for Editing and Debugging UML Class and State-Machine Diagrams," *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, Munich, Germany, 2019, pp. 779-783.

BibTex:

```
@INPROCEEDINGS{8904513,
author={P. {Pourali} and J. M. {Atlee}},
booktitle={2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)},
title={UCAnDoModels: A Context-Based Model Editor for Editing and Debugging UML Class and State-Machine Diagrams},
year={2019},
pages={779-783},
doi={10.1109/MODELS-C.2019.00122},
month={Sep.},}
```

DOI: https://doi.org/ 10.1109/MODELS-C.2019.00122

UCAnDoModels: A Context-based Model Editor for Editing and Debugging UML Class and State-Machine Diagrams

Parsa Pourali
Department of Electrical and Computer Engineering
University of Waterloo, Canada
ppourali@uwaterloo.ca

Joanne M. Atlee
David R. Cheriton School of Computer Science
University of Waterloo, Canada
jmatlee@uwaterloo.ca

ABSTRACT

Practitioners face cognitive challenges when using model editors to edit and debug UML models, which make them reluctant to adopt modelling. To assist practitioners in their modelling tasks, we have developed effective and easy-to-use tooling techniques and interfaces that address some of these challenges. The principle philosophy behind our tool is to employ cognitive-based techniques such as Focus+Context interfaces and increased automation of modelling tasks, in order to provide the users with valid, relevant and meaningful *contextual* information that are essential to fulfil a *focus* task (e.g., writing a transition expression). This paper presents our approach, which we call **U**ser-**C**entric and **A**rtefact-Ce**n**tric **D**evelopment **o**f **Models** (*UCAnDoModels*), and discusses two use-case scenarios to demonstrate how our tooling techniques can enhance the user experience with modelling tools.

KEYWORDS

User-Centric Software Development, UML Modelling Tools, Modelling Challenges, Focus+Context User Interfaces.

ACM Reference Format:
Parsa Pourali and Joanne M. Atlee. 2019. UCAnDoModels: A Context-based Model Editor for Editing and Debugging UML Class and State-Machine Diagrams. In *MODELS '19: EEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), September 15–20, 2019, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Model-Driven Engineering (MDE) has been proposed to reduce the complexity of designing software systems by providing tool supports that facilitate the code generation and early verification of a system's design [9]. Key barriers to the adoption of MDE are challenges in using tools [8][13]. This is partly because tool developers 1) do not systematically analyse users and their tasks to identify user difficulties, and 2) do not employ human-factors engineering to improve the tools' usability. In particular, tools' features do not address the modellers' cognitive challenges, instead usability features are based mostly on the model, the meta-model, and their properties. This leads to a huge gap between the support that tools provide and the support that users expect.

Our prior formative study [15] identified *information overload* as a major difficulty that modellers face when using model editors. Modelling languages propose different diagrams to model cross-cutting concerns of a system. Modellers expect model editors to assist in dealing with the relevant information that are separated into different diagrams; but the tools mostly offer navigability features rather than understandability features [10]. According to our formative study [15], an effective model editor should provide features to help users with their challenges of *remembering and editing the relevant contextual information (referred to as Context) when performing editing and debugging tasks (referred to as Focus).*

To alleviate the modellers' challenges, we took a user-centric approach and developed an Eclipse-based tool to support Feature-Oriented and Object-Oriented Modelling. Our approach, called **U**ser-**C**entric and **A**rtefact-Ce**n**tric **D**evelopment **o**f **Models** (*UCAnDoModels*), employs various tooling features (e.g., Focus+Context [2] interfaces) to edit and debug Feature Models, Feature Class Diagrams, and Feature Modules (which are enhanced UML State-Machine Diagrams) according to the Feature-Oriented Requirements Modelling Language (FORML)[20]. Please note that our tool is not limited to FORML; our tool allows UML modellers to develop *precise and analysable* UML Class and State-Machine diagrams without needing to be constrained by the requirements of Feature-Oriented Modelling. In this paper, we present our model editor and explain how our tooling features improve the users' experience of editing and debugging Feature Models, Class diagrams, and State-Machine diagrams.

We conducted two user studies to evaluate the effectiveness of our tool's features to edit and debug models [16]. The results showed significant reductions in users' efforts to edit and debug Class and State-Machine diagrams, and significant improvements in users' experience of using modelling tools.

The rest of this paper is organized as follows. Section 2 provides an overview of our UCAnDoModels model editor and a description of our Focus+Context editors. We explain our tool's capabilities in the context of illustrative scenarios in Section 3. Section 4 describes existing tools that are similar to ours and explains how our tool is different from them. We conclude our work in Section 5.

2 OVERVIEW OF UCANDOMODELS EDITOR

As shown in Fig. 1, our tool is developed using Eclipse-based frameworks such as the Eclipse Modelling Framework (EMF) [1], the Graphical Modeling Framework (GMF)[6], Sirius [21], and Xtext[3]. The tool provides graphical editors to edit the model (i.e., Feature-Model, Class and State-Machine diagrams) based on our meta-model

MODELS '19, September 15–20, 2019, Munich, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

Table 1: List of focus element with their contextual information.

ID	Diagram	Focus of User Action	Contextual Elements	Allowed Actions on Contextual Elements
FC1	State-Machine	State-Machine Elements (States, Transitions, Regions)	Elements in the corresponding Class diagram	Search, View, Modify
FC2		Editing transition expressions	Elements in the Class diagram	Search, View, Modify
FC3		Editing transition expressions	Other Transitions in all of the State-Machine diagrams	Search, View, Reuse
FC4		Selecting states in State-Machine diagrams	Sub State-Machine diagrams	View, Open
FC5	Class Diagram	Editing classes in the Class Diagram	Corresponding State-Machine diagram	View, Open
FC6		Editing elements in the Class Diagram	All the cross-references (i.e., usages) in the model	View, Modify
FC7		Editing feature classes in the (Feature) Class Diagram	Feature-Model (feature-tree)	View, Modify
FC8	Feature Model	Editing feature nodes in the Feature-Model	Corresponding State-Machine diagram	View, Open
FC9		Editing feature nodes in the Feature-Model	Relevant elements in the Class diagram	View, Modify
FC10	All / Other	Creating new State-Machine diagrams	Existing State-Machines	View, Reuse
FC11		Debugging an erroneous model	All the inconsistencies in the model	View, Modify

(i.e., FORML grammar [20]), which is implemented in Xtext. The Xtext engine generates a feature-rich textual editor that provides useful and usable features such as syntax-highlighting, Content-Assist, and displayed errors and warnings. To improve on these features, we have extended the Xtext engine with our Distance-Based Model-Assist, which reports additional errors and warnings, and enhances the Content-Assist.

Our contributions are highlighted in yellow in Fig. 1, which are Focus+Context Editors, Distance-Based Model-Assist, Model Observer, and Consistency Manager.

2.1 Focus+Context Interfaces

A **Focus+Context** interface, in general, allows users to concurrently perform a *focus* task while viewing and/or editing the relevant information to the task (i.e., *context*) [2]. The main goal is to alleviate user's navigation through several diagrams to locate specific information relevant to her current task.

Accordingly, our Focus+Context interfaces aim to alleviate the challenges of remembering contextual information when editing and debugging models. Table 1 shows a list of focus elements or tasks, the diagrams in which they exist, the contextual information relevant to the focus elements, and the type of actions that the user can perform on the contextual information. For instance, *FC2* refers

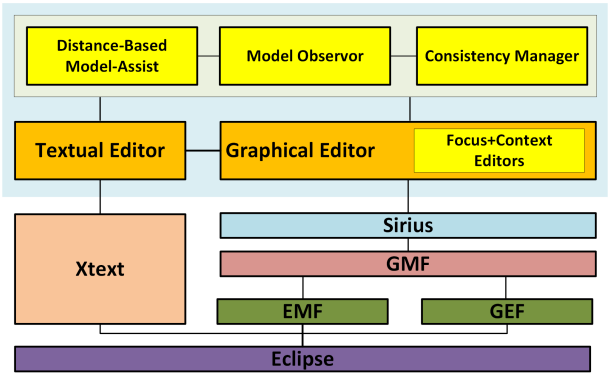

Figure 1: Overall architecture of our UCAnDoModels editor

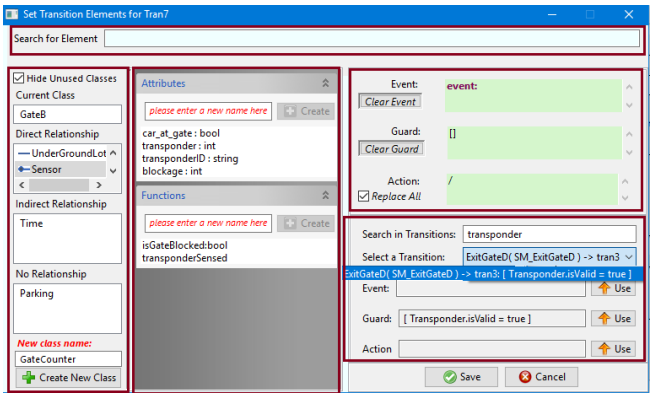

Figure 2: Our Focus+Context Transition Editor

to viewing and modifying the model elements in the Class diagram when editing a transition expression, as these model elements are among the vocabulary used to write precise transitions. The main purpose of *FC2* is to help modellers to recollect the elements to be used in the transition expression without needing to switch back and forth among different diagrams.

For example, we developed a Focus+Context Transition Editor (see Fig. 2) that eases the modeller's task of editing a transition expression, by helping the modeller recall, understand, and edit the related contextual information. The main five sections of the editor (which are outlined by red boundary boxes in the figure) are: 1) **Left section**: displays and allows users to edit (and create) the classes in a Class diagram as well as their relationships (i.e., associations), 2) **Middle section**: displays and allows users to edit the properties (i.e., attributes and operations) of a selected class from the Left section, 3) **Top section (search-bar)**: allows users to search for an intended element, 4) **Upper-Right section**: the focus section where the user expresses the event, guard, and actions of the transition currently being edited, and 5) **Bottom-Right section**: allows the user to search and reuse the event, guard, or actions from other transitions when editing the transition in the focus section.

Table 2: List of Head and Tail Actions

Head Action	Tail Action(s)
Create a new model element	Generate/Set a unique name for it
Create new Region r_1	Create new Initial Pseudo-State in r_1
Create new Initial Pseudo-State s_i in r_1	Create new State s_1 in r_1 and create a transition from s_i to s_1
Use an undefined operation or attribute in a State diagram	Define the missing Operation or Attribute in the Class diagram
Move a Class attribute or element to another Class	Update moved element's navigation paths to the new navigation paths.

2.2 User-Interactive Consistency Management

The Model Observer continuously listens to the model and notifies the Consistency Manager if any of the consistency rules (predefined in the tool) are violated as a result of a model edit. The Consistency Manager uses three types of error-resolution techniques, namely **Auto-Fix**, **Quick-Fix**, and **Interactive-Fix**. Auto-Fix is used when there is only one predefined fix for an inconsistency (e.g., renaming an element should be propagated to all its uses). Quick-Fix is used when the Consistency Manager finds a small number of predefined fixes for an inconsistency (e.g., if state *S1* in a State-Machine is not connected to any other state, the user is prompted to either delete *S1*, or select another existing state to connect with *S1*). Interactive-Fix is used in more complex situations that require the modeller to intervene, such as the inconsistency introduced by deleting an attribute that is referenced in other parts of the model.

2.3 Auto-Completion

The Model Observer not only helps to maintain model consistencies, but also performs a modeller's next edit in predefined situations, either automatically or by offering a set of options to select from. This limited form of auto-completion is performed based on well-formedness and semantic-correctness rules. For each auto-completion rule, we define an atomic Head Action followed by one or more Tail Actions. The Head Action is a change to the model that is initiated by the modeller. The Model Observer listens for Head Actions; when one occurs, it automatically triggers one or more corresponding Tail Action(s). For example, if the modeller moves an operation from a class to another class, the tool automatically updates all the navigation paths (e.g., guard expressions) that use the element to the new navigation path expression. In Table 2, we present a few Head and Tail Actions as examples of our rules.

3 ILLUSTRATIVE SCENARIO

In this section, we present scenarios showing a user's experience in editing and debugging of a model for a parking lot system. Due to the lack of space, this paper discusses only two scenarios that cover the most-important interfaces of our tool, namely FC2, FC3, and FC11. Interested readers can refer to our paper [16] for a more comprehensive description of our tool's features.

3.1 Illustrative Application

Consider that a model of the parking policies at a gated Parking Lot. The parking lot provides surface-level parking for customers (for a fee) and a free underground parking level for employees. Four gates (labelled A,B,C,D) control access to both levels of parking. Employees' cars have transponders that the gates can sense. Some gates open only for cars that have valid transponders. Each of the gates closes 5 seconds after it opens, unless a blockage is detected, in which case it waits for another 5 seconds. A partial Class diagram of the parking lot system is shown in Fig. 3.

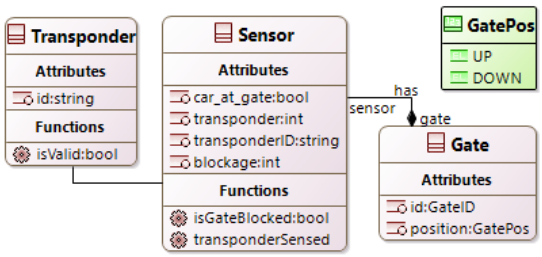

Figure 3: Partial Class diagram for the Parking Lot system

3.2 Scenario A

In Scenario A, the modeller needs to set the event, guard condition, and actions of a transition expression based on the following description:

To open Gate B, the system checks if the approaching car has a transponder and that the transponder is valid (e.g., not expired). If the conditions are met, the gate's position should be set to open. Moreover, the system needs to keep track of the number of cars that have passed through the gate. For this purpose, the gate should have a counter variable which is incremented by one every time the gate opens. To specify the action requirements, the modeller needs to create a GateCounter class which has an increaseCounter operation.

The following is a step-by-step explanation of how a user performs the above edit.

Step (1). Suppose that the modeller wants to specify the transition's triggering event using the Sensor class's operation that checks whether an approaching car has a transponder, namely *Sensor.transponderSensed()*. Our interface offers two ways of doing this to accommodate different user preferences (e.g., "keyboard-" vs. "mouse-" oriented preferences):

i. The modeller edits the *Event* text-field in the Upper-Right section: 1) user presses the *Control+Space* keys to invoke Content-Assist, 2) user starts typing the name of the intended element (i.e., *Sensor...*), 3) Content-Assist proposes (and continuously updates) valid model elements that can be used as a triggering event, and 4) user selects the intended element from the proposed list.
To help modeller find an intended element faster, our Distance-Based Model-Assist [16] employs an Association-based ranking mechanism to sort the proposed elements based on their association distance to the class whose State-Machine model is being edited. The elements are ranked as follows: 1) elements of the current class, 2) elements of the super-classes, 3) elements of the directly associated classes, 4) elements of the indirectly associated classes, 5) elements of indirectly associated classes, and 6) elements of non-related classes. Furthermore, if the modeller

types in only a part of the element's name, the editor uses the Levenshtein edit-distance algorithm [11] to find the elements throughout the model that have similar names to what the user has typed, and the user selects among the listed elements. This feature is especially useful when the user is unable to recall the exact name of an element, such as when modelling a large complex system.

ii. Alternatively, the user can select the *Sensor* class from the list of available classes shown in the Left section, to view its attributes and functions in the Middle section. By doing so, the user can then see the *transponderSensed* operation in the Middle section. This assumes that the modeller recalls that the *Sensor* class offers *transponderSensed* operation. If the modeller cannot recall the *Sensor* class, then she can use the Top section (search bar) to search for the *transponderSensed* element in the model, and if found, the tool displays and highlights the element in the Middle section. The user then drags the *transponderSensed* operation from the Middle section and drops it into the *Event* text-field. The tool automatically parses the dropped element and assigns it to the transition's event. *The user can use analogously drag-and-drop to set other parts of the transition (i.e., guard or actions), as well.*

The tool offers multiple capabilities to keep the modeller from having to search through an extensive list of all available classes: 1) the search bar finds potential intended elements based on a partial name, and 2) Distance-Based Model-Assist ranks the model's classes in the Left section based on their relationship to the current class. The intuition behind this ranking is that the modeller is more likely to use the elements of an associated class than the elements of a non-associated class. In this example, the Sensor class has a (direct) composition relationship with the Gate class, so the user easily finds the Sensor class in the list of Direct Relationship classes.

Step (2). Next, the modeller sets the guard expression to check that the car's transponder is valid. In addition to the discussed above ways of setting a transition's event, guard, or action segments, our interface allows the user to reuse text from other transitions in the model. In the Bottom-Right section of the interface, the user looks for an existing transition based on its name, containing StateMachine's name, triggering event, guard expression, or action. In our scenario, the modeller can simply search for *"transponder"*. As shown in Fig. 2, the name is found in transition *tran3* in the StateMachine *SM_ExitGateD*. Finally, the user clicks on the *Use* button that corresponds to the *Guard* text box, and the tool automatically copies *tran3*'s guard expression to the current transition's guard.

Step (3). The action includes two sub-actions: 1) set the gate's *position* to *up*, and 2) increase the *counter*. For the first sub-action, a modeller simply drags the gate's *position* attribute and drops it into the *Action* text-field. The editor then automatically creates an assignment statement and places the attribute into the Left-Hand-Side (LHS) of the statement (i.e., *"Gate.position="*). The editor then uses the Distance-Based Model-Assist to provide the user with a prioritized list of possible elements and values that can be assigned to the Right-Hand-Side (RHS) of the assignment based on the type of the *position* attribute. *GatePos* enumeration literals *UP* and *DOWN*

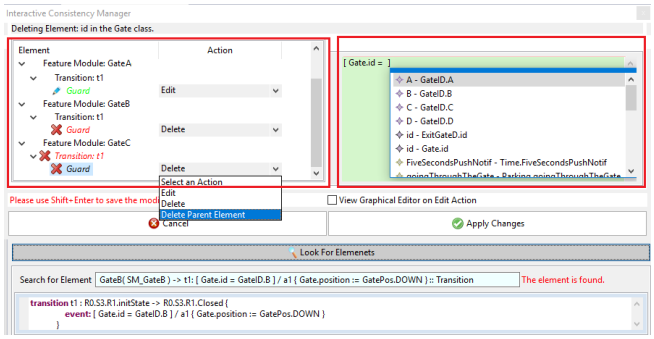

Figure 4: Our User-Interactive Fix Interface

top the list, and the user selects *UP. The editor analogously assists the user when they drop an attribute or operation into the guard text-field, to complete a conditional expression.*

With respect to the second sub-action, the modeller needs to create a *GateCounter* class and *increaseCounter* operation in order to specify an action that increments a counter. For this, the user creates a new class by entering the name *GateCounter* in the text box located on the Bottom-Left side of the editor and clicking on the *Create New Class* button. By default, the newly created class is assigned to the *No Relationship* category; the user drags the newly created class and drops it into the list of *Direct Relationship* classes in order to create an association link between the current and new classes. Afterwards, the user creates the *increaseCounter* operation in the *GateCounter* class by using the text box and the create button in the Middle section. *Using the same steps a user can create a new attribute for a class.*

3.3 Scenario B

In Scenario B, the modeller needs to resolve inconsistencies that are introduced into the model as a result of her edits.

The user removes the Gate's id attribute. As a result of this deletion, the modeller is expected to resolve the inconsistencies that are introduced to various diagrams that use the id attribute.

We discuss the user experience in debugging models as follows.

Step (1). The user deletes the *id* attribute from the *Gate* class. As the *id* attribute is used in other diagrams, the Model Observer detects the uses of an undefined id attribute and notifies the Consistency Manager to handle the inconsistencies.

Step (2). The Interactive-Fix pops up our Interactive-Fix Consistency Management interface. In the Top Left section of the interface shown in Fig. 4, the editor lists all of the inconsistent elements in a hierarchical (tree-style) view, where the leaf nodes of the hierarchy identify all of the inconsistent elements and the root nodes identify the diagrams containing the inconsistent elements. For instance, Fig. 4 shows that deleting the *id* attributes introduces inconsistencies to three guard elements in three different StateMachines diagrams: *Feature-Module GateA*, *Feature-Module GateB*, and *Feature-Module GateC. The main goal of using a hierarchical view is to inform the modeller about all of the model elements that play a role in the inconsistency.*

Step (3). The modeller selects a resolution action for each inconsistency by using the *Action-To-Do* menu. *The modeller can select*

to either edit the elements that use the id attribute (e.g., rewrite the guard conditions), delete the erroneous guard elements, or just delete the guards' parent transitions. Based on the type of action, an icon of ***Delete*** *or* ***Edit*** *will be shown beside the edited lead node.*

Step (4). The user chooses to *edit* the first inconsistent guard, which bring up a feature-rich textual editor on the right side of the interface. *In addition to the textual representation, the modeller can view the graphical diagrams and edit the elements graphically by selecting the "View Graphical Editor on Edit Action" located below the textual editor.*

Step (5). After addressing all of the inconsistencies, the user clicks on the *Apply Changes* button to apply all the resolutions. *If the user clicks on the Cancel button, the model will revert back to latest correct state (i.e., before deleting the id attribute).*

4 SIMILAR TOOLS

Many UML model editors [4, 5, 7, 12, 14, 17–19] ease the editing and debugging of models by proposing artefact-centric techniques, rather than by considering the human user in the loop (i.e., user-centric techniques). For example, Capella [19], MagicDraw [7], and VisualParadigm [14] are proficient model editors which (in general) aggregate and augment the most useful features of other similar editors and adequately address modellers' concerns regarding expected usefulness. Although these tools provide various graphical and textual widgets that improve usability, they do not explicitly design their editors to target and alleviate the users' cognitive challenges. For instance, MagicDraw and VisualParadigm only allow the modeller to write a transition's guard expressions as plain text, which is very error-prone when referring to model elements with complex navigation path names. Capella provides a Content-Assist to help the modeller choose among the available model elements when developing guard expressions; however, its Content-Assist proposes an exhaustive list of all the model elements, which is cumbersome to search through.

MDE editors should balance artefact-centric approaches (i.e., techniques that mostly consider the properties of the model and the meta-model) required by MDE tools and user-centric approaches that are needed and expected by modellers. That is, tool vendors need to pay more heed to modellers, and avoid designing tools based only on the written standards and specifications. For instance, based on UML specifications, the structural and behavioural aspects of a system should be specified in separate diagrams (for the sake of separation of concerns). Most of tools do not tackle the problem of information that is separated in different diagrams, requiring the modeller to search back and forth amongst different diagrams to find a piece of information. However, the relationships between diagrams also need to be captured in the separate diagrams. It is crucial that the tools assist modellers in fetching and understanding these inter-related information.

5 CONCLUSION

We described our tool which facilitates the development of Feature-Oriented and UML-like models by employing and balancing the user-centric and artefact-centric techniques. The rationale behind our tool is to alleviate the cognitive challenges of modellers when remembering the contextual information that is relevant to performing a particular task. Furthermore, our tool generally aims to reduce the cognitive efforts of developing models of formal languages by applying semi-automated repairs and auto-completions in the course of (formal) modelling. We believe that the proposed techniques provide insight for tool vendors into how to improve the usability and utility of model editors, by employing human-centric features that reduce users' cognitive challenges.

REFERENCES

[1] Stefan Berlik. 2007. *Eclipse Modeling Framework.* Addison-Wesley. 1–14 pages.

[2] Andy Cockburn, Amy Karlson, and Benjamin B Bederson. 2009. A review of overview+ detail, zooming, and focus+ context interfaces. *ACM Computing Surveys (CSUR)* 41, 1 (2009), 2.

[3] Sven Efftinge and Markus Völter. 2006. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, Vol. 32. 118.

[4] Miguel a. Garzon, Hamoud Aljamaan, and Timothy C. Lethbridge. 2015. Umple: A framework for Model Driven Development of Object-Oriented Systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* 494–498. https://doi.org/10.1109/SANER.2015.7081863

[5] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. 2007. Papyrus: A UML2 tool for domain-specific language modeling. In *Proceedings of the International Dagstuhl Conference on Model-Based Engineering of Embedded Real-Time Systems (MBEERTS'07).* Springer-Verlag, 361–368. https://doi.org/10.1007/978-3-642-16277-0_19

[6] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. 2009. Language evolution in practice: The history of GMF. In *International Conference on Software Language Engineering.* Springer, 3–22.

[7] No Magic Inc. 2013. Magicdraw, UML. (2013).

[8] Rodi Jolak, Truong Ho-Quang, Michel RV Chaudron, and Ramon RH Schiffelers. 2018. Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.* ACM, 213–223.

[9] Stuart Kent. 2002. Model Driven Engineering. In *International Conference on Integrated Formal Methods.* Springer, 286–298.

[10] Mik Kersten. 2007. *Focusing knowledge work with task context.* Ph.D. Dissertation. University of British Columbia.

[11] V. Levenshtein. 1965. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission* 1 (1965), 8–17.

[12] Andreas Muelder. 2011. Yakindu Statechart Modeling Tools.

[13] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty HC Cheng, Philippe Collet, Benoit Combemale, Robert B. France, Rogardt Heldal, James Hill, et al. 2014. The relevance of model-driven engineering thirty years from now. (2014), 183–200.

[14] Visual Paradigm. 2013. Visual paradigm for uml. *Visual Paradigm for UML-UML tool for software application development* (2013), 72.

[15] Parsa Pourali and Joanne M Atlee. 2018. An Empirical Investigation to Understand the Difficulties and Challenges of Software Modellers When Using Modelling Tools. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.* ACM, 224–234.

[16] Parsa Pourali and Joanne M Atlee. 2019. A Focus+Context Approach to Alleviate Cognitive Challenges of Editing and Debugging UML Models. In *Proceedings of the 22th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems.* IEEE.

[17] Steffen Prochnow and Reinhard von Hanxleden. 2007. Statechart Development Beyond WYSIWYG. In *Model Driven Engineering Languages and Systems.* Lecture Notes in Computer Science (LNCS), Vol. 4735. Springer Berlin Heidelberg, 635–649. https://doi.org/10.1007/978-3-540-75209-7_43

[18] Jason E. Robbins and David F. Redmiles. 2000. Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Information and Software Technology* 42, 2 (2000), 79–89.

[19] Pascal Roques. 2016. MBSE with the ARCADIA Method and the Capella Tool. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016).* Toulouse, France. https://hal.archives-ouvertes.fr/hal-01258014

[20] Pourya Shaker, Joanne M. Atlee, and Shige Wang. 2012. A feature-oriented requirements modelling language. In *20th IEEE International Requirements Engineering Conference, RE 2012.* IEEE, 151–160. https://doi.org/10.1109/RE.2012.6345799

[21] Vladimir Viyović, Mirjam Maksimović, and Branko Perišić. 2014. Sirius: A rapid development of DSM graphical editor. In *INES 2014 - IEEE 18th International Conference on Intelligent Engineering Systems, Proceedings.* IEEE, 233–238. https://doi.org/10.1109/INES.2014.6909375