TPV: A Tool for Validating Temporal Properties in UML Class Diagrams

Mustafa Al Lail, Antonio Viesca, Hector Cardenas
mustafa.allail@tamiu.edu
antoniorosales@dusty.tamiu.edu
hector_cardenas@dusty.tamiu.edu
Texas A&M International University
Laredo, Texas, USA

Mohammad Zarour
mzarour@hu.edu.jo
Hashemite University
Zarqa, Jordan

Alfredo Perez
alfredoperez@unomaha.edu
University of Nebraska at Omaha
Omaha, Nebraska, USA

ABSTRACT

Software scientists and practitioners have criticized Model-driven engineering (MDE) for lacking effective tooling. Although progress has been made, most MDE analysis tools rely on complex, heavyweight mathematical techniques that are not based on UML. Such tools require a steep learning curve and suffer from many accidental complexities. We developed the Temporal Property Validator (TPV) to tackle this issue. TPV allows designers to specify and analyze temporal properties using UML notations, techniques, and tools. We evaluated TPV using the user experience evaluation method and obtained promising results in all aspects of user needs. You can download TPV and view the demo video from https://github.com/mustafalail/TPV-Tool.

CCS CONCEPTS

• **Software and its engineering → Unified Modeling Language (UML)**; **System modeling languages**.

KEYWORDS

Temporal propeties,OCL, UML, tool, model checking, verification

ACM Reference Format:
Mustafa Al Lail, Antonio Viesca, Hector Cardenas, Mohammad Zarour, and Alfredo Perez. 2024. TPV: A Tool for Validating Temporal Properties in UML Class Diagrams. In *2024 IEEE/ACM 46th International Conference on Software Engineering*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Tools are essential for any software development paradigm, and their quality directly impacts its usefulness. Software engineers have confirmed that high-quality, easy-to-use, and robust tools can maximize the benefits of a paradigm while minimizing the difficulties designers face when learning and using it. However, inadequate MDE tools have been identified by several researchers as a significant, recurring obstacle to the industrial adoption of MDE [6, 8, 14, 29–31]. For an MDE approach to be successful, software designers need to incorporate practical techniques and tools into the development process to improve the quality of the models. When models with design flaws are used to generate executable code, errors are passed down and can be expensive to fix. Therefore, detecting defects in the models as early as possible is vital by analyzing them and ensuring that their behavior adheres to the specified requirements. Such requirements can be expressed as safety temporal properties (e.g., ensuring the system never reaches a deadlock) or liveness properties (e.g., preventing starvation).

Verifying a system model's satisfying temporal properties is a common practice involving *model checking* [9]. Many UML-based model-checking techniques rely on UML behavioral models, such as state machines or activity diagrams, to describe a system's behavior, as seen in Moffett's approach [22]. However, a significant challenge with these approaches is the transformation of UML models into specific model-checking frameworks for verification, as argued by France and Rumpe [14]. They emphasize the difficulty of ensuring semantic correctness and hiding the complexities of target model-checking technologies from UML designers. Furthermore, most of these approaches use temporal logic formalisms like LTL [27] and CTL [10] to specify temporal properties, which may be challenging for designers to learn and use effectively [13]. In essence, existing UML-based model-checking approaches face usability, effectiveness, and efficiency issues similar to other MDE tools [15].

To overcome these challenges, Al Lail et al. [1–5] proposes a new methodological framework that allows UML designers to use UML notations, tools, and techniques to specify and analyze temporal properties. This framework's implementation is the cutting-edge analysis tool called the Temporal Property Validator (TPV), specifically designed to cater to UML designers, the envisioned users.

2 THE UML-BASED FRAMEWORK

This section discusses the software engineering challenges TPV addresses, the originality of the techniques it uses to address them, and the methodology it implies for its users. The UML specification defines many types of diagrams to model different aspects of a system. Class diagrams are central and widely used in model-driven design [26]. Although designers usually use class diagrams for structural modeling, they can also specify behavior through operation contracts. Operation contracts can adequately express

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

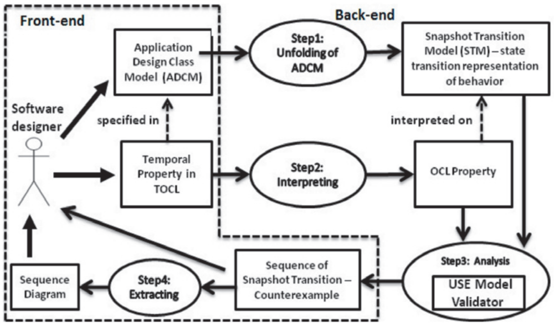

Figure 1: Overview of the technique implemented by TPV [2]

the behavior of a system, as shown by Porres and Rauf (2009) [28]. Designers use the Object Constraint Language (OCL) to specify operation contracts and other constraints [23]. However, OCL does not make it easy to specify temporal properties. To address this shortcoming, Ziemann et al. (2003) defined Temporal OCL (TOCL), an extension to OCL that supports temporal properties [34]. The framework allows for the analysis of TOCL properties on UML class diagrams. The framework consists of the following techniques:

(1) A UML-based analysis technique that exclusively employs UML notations and tools.
(2) A UML-based method that simplifies the process of temporal property specification for UML designers.
(3) An optimization technique that reduces the time required for analysis, enabling the analysis to be applied to larger UML models.

We elaborate on these techniques in the following subsections.

2.1 The Analysis Technique

TPV implements the analysis technique depicted in Fig. 1. Unlike other related work, the novelty of the technique lies in its exclusive use of UML notations and tools– eliminating the accidental complexities and difficulties of related techniques. The first step in the analysis workflow is to create a UML diagram for a specific software system and then specify a temporal property in TOCL.

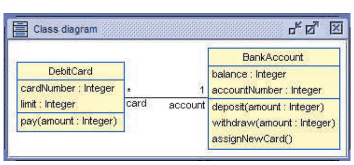

Figure 2: Class diagram modeling accounts and debit cards.

We use the class diagram presented in Fig. 2 to demonstrate the analysis technique. This model comprises two classes, namely *BankAccount* and *DebitCard*. Account objects have a balance that can increase through the *deposit()* operation or decrease through the *withdraw()* operation. Accounts can be assigned a debit card through the *assignNewCard()* operation. A selection of OCL contracts for these operations is displayed in Listing 1.

```
context BankAccount::withdraw(amount:Integer)
post: self.balance = self.balance@pre - amount
context BankAccount::deposit(amount:Integer)
post: self.balance = self.balance@pre + amount
context DebitCard::pay(amount:Integer)
pre: amount < self.limit
post: self.account.balance =
        self.account.balance@pre-amount
```

Listing 1: Operation contracts for the diagram in Fig. 2.

The designer inputs the system requirements as temporal properties that define the system's behavior in TOCL. Listing 2 presents an example of a property for the *BankAccount* class requiring the account balance to be greater than 0 eventually.

```
context BankAccount
inv: sometime balance > 0
```

Listing 2: Example of a temporal property.

Once a designer specifies a temporal property, they can use TPV to check it. TPV analyzes the system model and identifies any counterexamples. At the back end, TPV automatically transforms the class diagram into a different form to make it suitable for behavioral analysis. Fig. 3 displays the resultant form in terms of a *Snapshot Transition Model* (STM) of the class diagram in Fig. 2. The initial form of STM was proposed in 2008 by Yu et al. [32] and extended for temporal property verification by Al Lail et al. [1–5].

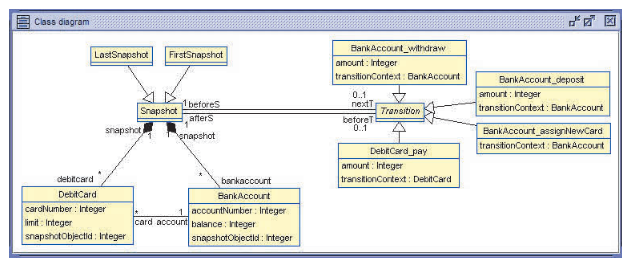

Figure 3: STM representation of the account system.

An STM is a structural representation of a system's behavior through states and transitions. Each state, known as a *Snapshot*, is a structured class that represents an object diagram of the system at a particular moment of execution. *Transition* classes, in Fig. 3, indicate operation calls that cause new system states and result in side effects. The operation pre- and post-conditions are converted into invariants on the transition classes to preserve their constraints, as shown by listing 3. The class invariants remain the same and directly mapped their perspective class in STM. Everything else stays intact.

```
context BankAccount_withdraw
inv post1: (self.transitionContext.getNext().balance = (
      self.transitionContext.balance - amount))
context BankAccount_deposit
inv post2: (self.transitionContext.getNext().balance = (
      self.transitionContext.balance + amount))
context DebitCard_pay
inv pre1: (amount < self.transitionContext.limit)
context DebitCard_pay
inv post3: (self.transitionContext.getNext().account.
      balance = (self.transitionContext.getNext().account.
      getPrevious().balance - amount))
```

Listing 3: Contracts in Listing 1 mapped into STM invariants.

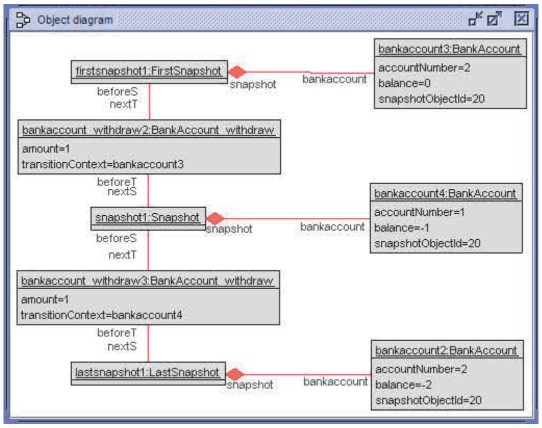

Figure 4: A counterexample as a snapshot-transition chain.

The STMs are automatically used to create object diagrams that describe specific execution scenarios of a system. Fig. 4 displays an example scenario. The scenario is composed of linked *Snapshot* and *Transition* objects. Note the *snapshotObjectId* attribute present in the Fig. 3 STM and in the Fig. 4 counterexample. This attribute is used to identify objects in different snapshots as state representations of the same real-time object. For example, the scenario in Fig. 4 shows three different STM objects with a *snapshotObjectId* value of *20*. These objects represent three states of a single real-time bank account instance. *Transition* elements signify operation calls that affect the system and lead to a new system state. This static representation of system behavior allows the specification and evaluation of TOCL temporal properties and pre- and post-conditions as OCL invariants on STM elements.

To analyze a system's behavior, TPV uses the UML-based Specification Environment (USE) tool and its Model Validator plugin [20]. The USE Model Validator automatically identify any object diagram that violates the TOCL specification within a certain search range. If the analysis detects a system state in which a temporal property is violated, the validator generates a counterexample to demonstrate that the system does not meet the respective property. The designer can use this counterexample to improve their design. For example, the counterexample displayed in Fig. 4 reveals that the account model's specification is too lenient, allowing the violation of the temporal property listed in Listing 2. To fix this flaw, the model's behavior can be restricted by specifying additional constraints.

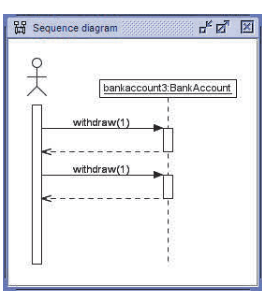

Figure 5: Counterexample scenario as a sequence diagram.

When there are numerous snapshots and transitions in a counterexample scenario, the verification results can become complicated and challenging to analyze. To simplify this process, TPV generates a sequence diagram from the counterexample scenario to assist with debugging. The extracted sequence diagram from the counterexample scenario in Fig. 4 is displayed in Fig. 5.

2.2 The Property Specification Technique

Formally specifying system requirements as temporal properties is a challenging task for many designers [13]. This specification technique streamlines the process of property specification for UML designers. To achieve this goal, the specification patterns of Dwyer et al are defined in TOCL. A user determines a pattern that best fits the requirement and then uses the corresponding TOCL pattern to obtain the intended property. The OCL property is then systematically generated for verification. You can refer to Tables in Fig. 6 for templates of the Response and Universality patterns.

Scope	TOCL Pattern
Globally	context *[Class]* inv: *[P]* implies sometime *[S]*
Before R	context *[Class]* inv: *[R]* implies sometime *[S]* since *[P]*
After Q	context *[Class]* inv: *[Q]* implies always (*[P]* implies sometime *[S]*)
After Q until R	context *[Class]* inv: *[Q]* implies always (*[P]* implies sometime *[S]* before *[R]*)
Between Q and R	context *[Class]* inv: *[Q]* and sometime *[R]* implies always (*[P]* implies sometime *[S]* before *[R]*)

Table 1: The response pattern in TOCL

Scope	TOCL Pattern
Globally	context *[Class]* inv: always *[P]*
Before R	context *[Class]* inv: *[R]* implies alwaysPast *[P]*
After Q	context *[Class]* inv: *[Q]* implies always *[P]*
After Q until R	context *[Class]* inv: *[Q]* implies always *[P]* until *[R]*
Between Q and R	context *[Class]* inv: *[Q]* and sometime *[R]* implies always *[P]* until *[R]*

Table 2: The universality pattern in TOCL

Figure 6: Examples of pattern specifications in TOCL.

2.3 The Optimization Technique

State explosion is a known challenge in model checking [9]. This issue arises when the state space becomes too large to be feasibly checked. TPV employs two strategies to mitigate state explosion: lightweight analysis of a restricted search space and search space optimization. The lightweight analysis strategy reduces the search space by constraining the search scope and search depth [18]. A search scope defines the number of objects created for each class in a snapshot, while a search depth specifies the number of transitions considered in an analysis task. When analyzing a property related to a class diagram, the optimization process identifies relevant elements and returns an optimized diagram. Algorithm 1 outlines the steps to identify affected elements by a temporal property (*Tp*), taking pre and post-conditions (*PcList*), a list of class invariants (*InvList*), and the class diagram (*CD*) as inputs.

The algorithm uses a function, *elements()*, that takes an OCL expression as input and returns the set of affected elements in the expression. The algorithm consists of two loops identifying the set

Algorithm 1 Optimization to identify affected elements (AE)

```
1: Input: CD, PcList, InvList, and Tp
2: Output: AE
3: AE = ∅
4: PcAE = ∅
5: TpAE = elements(Tp)
6: for Pc ∈ PcList do
7:     PtAE = elements(Pc)
8:     if PtAE ∩ TpAE ≠ ∅ then
9:         PcAE = PcAE ∪ PtAE
10:    end if
11: end for
12: InvAE = ∅
13: for Inv ∈ InvList do
14:    PtAE = elements(Inv)
15:    if PtAE ∩ TpAE ≠ ∅ then
16:        InvAE = InvAE ∪ PtAE
17:    end if
18: end for
19: AE = TpAE ∪ PcAE ∪ InvAE
20: Return AE
```

of affected elements from the pre and post-conditions of operations (lines 6-11) and invariants (lines 13-18).

3 TOOL IMPLEMENTATION

We utilized MDE technologies to define, implement, and package the code of the TPV tool as a USE plugin. The plugin includes a user-friendly GUI and conceals the back-end processing from the user. Additionally, it improves USE's functionality by allowing it to open and store models in XMI files which is the standard format for UML models. This feature enhances the interoperability of TPV by allowing models to be exported and imported from other tools. To create TPV, we followed these steps:

(1) We developed a new modeling language for the STM by using UML-based class diagrams to represent behavior.
(2) We created transformation rules that automatically converting UML class diagrams into the new modeling language. We formally defined these rules using QVTo [24] and implemented them using the EMF implementation of QVTo [16].
(3) We started with the original TOCL specification and developed a formal EBNF grammar, a parser for TOCL using the ANTLR 4 parser generator [25], and a metamodel of TOCL.
(4) We established transformation rules from TOCL to OCL to enable a more accessible analysis of TOCL properties. This transformation involved the formal specification of QVTo rules and their implementation.
(5) We implemented the optimization technique.

4 TPV EVALUATION

To assess TPV, we used a survey based on the innovative user needs experience (NX) method by Zarour (2020) [33]. The survey includes 44 questions grouped into 4 sections focusing on usefulness, pleasure, aesthetics, and trust. We collected feedback from student participants to gain insight into the user experience of TPV. The study was made as realistic as possible for actual TPV users. Students downloaded TPV and received an evaluation guide with basic training on software specification and validation. They completed two specification and validation case studies before responding to the survey questions. A total of 19 students participated in the study. The questions, student responses, and detailed analysis can be found in the online repository provided in the abstract.

Table 1 shows student survey results on four criteria. TPV is helpful but needs improvement in user experience categories. UI is good but could be better. Students found TPV dependable.

Table 1: TPV Overll Evaluation

Criteria	Satisfied	Dissatisfied
Usefulness	90.15%	9.85%
Pleasure	91.24%	8.76%
User Interface Aesthetics	93.63%	6.37%
Trust	91.65%	8.35%

5 RELATED WORK

A recent article provides an overview of the latest model-based formal verification techniques and tools, discussing the current state-of-the-art and outlining possible future research directions [15]. This section only covers recent and relevant approaches. In [7], the MADES approach is presented, which combines several heavyweight formalisms and techniques to verify embedded systems. These techniques require steep learning and mathematical skills, making the tool hard to use by UML designers. Unlike this approach, by examining TPV's input and output, one can see that it exclusively uses notations and techniques familiar to UML designers. Combemale et al. [11] use a temporal extension to OCL based on process states to specify temporal constraints. These constraints are then translated to Petri nets for verification. Designers are, therefore, required to learn Petri nets to understand the verification results. Similarly, ProMoBox [21] supports verifying temporal properties in the context of domain-specific modeling. ProMoBox defines a family of five languages that are required to support property specification and verification. Properties are specified in LTL, and models are translated to the Spin model checker for verification. In [17], an approach similar to the technique used in TPV is described. The approach performs verification using UML class diagrams and OCL expressions by transforming them into the so-called filmstrip model that is verified by the USE model finder. However, compared to TPV, the approach offers a limited set of temporal operators based on non-UML notation LTL and lacks support for specification patterns. The research work described in [12, 19] discusses pattern-based specification approaches. [19] implements the patterns on top of Eclipse but does not provide a verification tool. On the other hand, [12] describes a new property, a model-based testing approach using UML/OCL models to evaluate the quality of test suites. However, the approach uses heavyweight techniques.

ACKNOWLEDGMENTS

This work received partial support from NSF grant awards 1950416 and 2308741 and the University Research Grant from TAMIU.

REFERENCES

[1] Mustafa Al-Lail. 2013. A Framework for Specifying and Analyzing Temporal Properties of UML Class Models.. In *MoDELS (Demos/Posters/StudentResearch)*. 112–117.

[2] Mustafa Al Lail. 2018. *A Unified Modeling Language Framework for Specifying and Analyzing Temporal Properties*. Ph.D. Dissertation. Colorado State University.

[3] Mustafa Al-Lail, Ramadan Abdunabi, Robert B France, and Indrakshi Ray. 2013. Rigorous Analysis of Temporal Access Control Properties in Mobile Systems. In *2013 18th International Conference on Engineering of Complex Computer Systems*. IEEE, 246–251.

[4] Mustafa Al-Lail, Ramadan Abdunabi, Robert B France, Indrakshi Ray, and F Boulanger. 2013. An Approach to Analyzing Temporal Properties in UML Class Models.. In *MoDeVVa@ MoDELS*. Citeseer, 77–86.

[5] Mustafa Al-Lail, Wuliang Sun, and Robert B. France. 2014. Analyzing Behavioral Aspects of UML Design Class Models against Temporal Properties. In *2014 14th International Conference on Quality Software, Allen, TX, USA, October 2-3, 2014*. IEEE, 196–201. https://doi.org/10.1109/QSIC.2014.56

[6] Omar Badreddin, Rahad Khandoker, Andrew Forward, Omar Masmali, and Timothy C Lethbridge. 2018. A decade of software design and modeling: A survey to uncover trends of the practice. In *Proceedings of the 21th acm/ieee international conference on model driven engineering languages and systems*. 245–255.

[7] Luciano Baresi, Gundula Blohm, Dimitrios S Kolovos, Nicholas Matragkas, Alfredo Motta, Richard F Paige, Alek Radjenovic, and Matteo Rossi. 2015. Formal verification and validation of embedded systems: the UML-based MADES approach. *Software & Systems Modeling* 14, 1 (2015), 343–363.

[8] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19 (2020), 5–13.

[9] Edmund M Clarke. 2008. The birth of model checking. In *25 Years of Model Checking*. Springer, 1–26.

[10] Edmund M Clarke and E Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*. Springer, 52–71.

[11] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and Francois Vernadat. 2007. A property-driven approach to formal verification of process models. In *International Conference on Enterprise Information Systems*. Springer.

[12] Frédéric Dadeau, Elizabeta Fourneret, and Abir Bouchelaghem. 2019. Temporal property patterns for model-based testing from UML/OCL. *Software & Systems Modeling* 18, 2 (2019), 865–888.

[13] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1999. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*. 411–420.

[14] Robert France and Bernhard Rumpe. 2007. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*. IEEE.

[15] Sebastian Gabmeyer, Petra Kaufmann, Martina Seidl, Martin Gogolla, and Gerti Kappel. 2019. A feature-based classification of formal verification techniques for software models. *Software & Systems Modeling* 18, 1 (2019), 473–498.

[16] Richard C Gronback. 2009. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education.

[17] Frank Hilken and Martin Gogolla. 2016. Verifying linear temporal logic properties in UML/OCL class diagrams using filmstripping. In *2016 Euromicro Conference on Digital System Design (DSD)*. IEEE, 708–713.

[18] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

[19] Bilal Kanso and Safouan Taha. 2012. Temporal constraint support for OCL. In *International Conference on Software Language Engineering*. Springer, 83–103.

[20] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. 2011. Extensive validation of OCL models by integrating SAT solving into USE. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 290–306.

[21] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. 2014. ProMoBox: a framework for generating domain-specific property languages. In *International Conference on Software Language Engineering*. Springer, 1–20.

[22] Yann Moffett, Juergen Dingel, and Alain Beaulieu. 2013. Verifying protocol conformance using software model checking for the model-driven development of embedded systems. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1307–13256.

[23] 2014. *Object Constraint Language 2.4.1*. Object Management Group (OMG). https://www.omg.org/spec/OCL/2.4/About-OCL/

[24] OMG. 2016. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. https://www.omg.org/spec/QVT/1.3/

[25] Terence Parr. 2013. The definitive ANTLR 4 reference. *The Definitive ANTLR 4 Reference* (2013), 1–326.

[26] Marian Petre. 2013. UML in practice. In *2013 35th international conference on software engineering (icse)*. IEEE.

[27] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. https://doi.org/10.1109/SFCS.1977.32

[28] Ivan Porres and Irum Rauf. 2009. Generating class contracts from deterministic UML protocol statemachines. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 172–185.

[29] Charlotte Verbruggen and Monique Snoeck. 2021. Model-driven engineering: A state of affairs and research agenda. In *International Conference on Business Process Modeling, Development and Support, International Conference on Evaluation and Modeling Methods for Systems Analysis and Development*. Springer, 335–349.

[30] Thomas Weber, Alois Zoitl, and Heinrich Hußmann. 2019. Usability of development tools: A case-study. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 228–235.

[31] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. 2013. Industrial adoption of model-driven engineering: Are the tools really the problem?. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 1–17.

[32] Lijun Yu, Robert B. France, and Indrakshi Ray. 2008. Scenario-Based Static Analysis of UML Class Models. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5301)*, Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter (Eds.). Springer, 234–248. https://doi.org/10.1007/978-3-540-87875-9_17

[33] Mohammad Zarour. 2020. A rigorous user needs experience evaluation method based on software quality standards. *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 18, 5 (2020), 2787–2799.

[34] Paul Ziemann and Martin Gogolla. 2003. OCL extended with temporal logic. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 351–357.