Designing, Animating, and Verifying Partial UML Models

Frédéric Jouault
ERIS, ESEO-TECH
Angers, France
frederic.jouault@eseo.fr

Valentin Besnard
ERIS, ESEO-TECH
Angers, France
valentin.besnard@eseo.fr

Théo Le Calvar
LERIA, Université d'Angers
Angers, France
theo.lecalvar@univ-angers.fr

Ciprian Teodorov
Lab-STICC UMR CNRS 6285,
ENSTA Bretagne
Brest, France
ciprian.teodorov@ensta-bretagne.fr

Matthias Brun
ERIS, ESEO-TECH
Angers, France
matthias.brun@eseo.fr

Jerome Delatour
ERIS, ESEO-TECH
Angers, France
jerome.delatour@eseo.fr

ABSTRACT

Models have been shown to be useful during virtually all stages of the software lifecycle. They can be reverse engineered from existing artifacts, or created as part of a system's execution, but in many cases models are created by designers from informal specifications. In the latter case, such design models are typically used as means of communication between designers, and developers. They can also in some cases be validated by simulation over test cases, or even by formal verification. However, most existing model simulation or verification approaches require relatively consistent and complete models, whereas design models often start small, incomplete, and inconsistent. Moreover, few design models actually reach the stage where they can be simulated, and even fewer the stage where they can be formally verified. In order to address this issue, we propose a partial modeling approach that makes it possible to animate incomplete and inconsistent models. This approach makes it possible to incrementally improve testable models, and can also help designers reach the stage where their models can be formally verified. A proof-of-concept tool called AnimUML has been created in order to provide means to evaluate the approach on several examples. They are all executable, and some can even undergo model-checking.

CCS CONCEPTS

• **Software and its engineering** → *Model-driven software engineering*; *Unified Modeling Language (UML)*; *Formal software verification*; *Interpreters*; *Software prototyping*; • **Computing methodologies** → *Model verification and validation.*

KEYWORDS

partial models, model simulation, validation and verification

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410967

ACM Reference Format:
Frédéric Jouault, Valentin Besnard, Théo Le Calvar, Ciprian Teodorov, Matthias Brun, and Jerome Delatour. 2020. Designing, Animating, and Verifying Partial UML Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3365438.3410967

1 INTRODUCTION

Models exist in many forms, and for many purposes. Some are barely more than drawings, whereas others can be formally verified. To support validation of design models before they are actually implemented, execution semantics must be defined in some way. In the case of UML models [17], this is notably done through the fUML [16]. and associated standards: PSCS [18] for composite structure, and PSSM [15] for state machines. All tools supporting these standards can execute UML models in a consistent way.

However, achieving model executability according to these standards is actually not an easy task. Virtually all aspects of the model must be consistent, and complete. This is rarely the case for actual design models, and extensive work is then required to make them executable. In most cases, this extra work is not performed, and models remain untested. Designers and developers each have their own understanding of how they should behave, and basically understand them by simulating them in their minds. This clearly contrasts with the way code behavior understanding is typically supported by modern debuggers showing how it actually executes.

In this work, we propose an approach that relaxes several UML constraints to make incomplete models executable in some sense. The objective is not to execute models as if they were code. This can already be achieved with fUML, for instance. On the contrary, the main goal is to help designers and developers understand, and test the behavior of their models by *playing* with them. This can notably help them improve their model, and possibly achieve enough consistency and completeness to make them executable with fUML, for instance.

The approach presented in this paper has been implemented in a proof-of-concept tool that has been used to animate multiple models at various stages of completeness and consistency. This tool also supports connection to a model-checker to perform more advanced validation once models become more complete. A more complete evaluation with actual users has not been performed at this stage, and this work is therefore presented as a *new idea* paper.

The remainder of the paper is organized as follows. Section 2 gives an overview of the proposed approach. Application to a single active object is explained in Section 3. Section 4 extends this by showing how multiple active objects can interact, and Section 5 focuses on the representation of passive objects. Section 6 presents a proof-of-concept tool that supports the approach. Some related work are discussed in Section 7, and Section 8 gives some concluding remarks.

2 APPROACH OVERVIEW

The three main guiding principles behind the presented approach are: 1) models should be executable as early as possible during the design process, 2) designers should be assisted in understanding execution, and 3) designers should be able to control execution. This section starts by focusing on the notion of executable partial models before considering how their executions may be represented and controlled.

2.1 Executable Partial Models

Near the end of the design process, produced models should comply with the UML standard. However, intermediate versions will be incomplete, and often not executable without relaxing compliance. Because UML is relatively permissive, and already has semantic variation points, it already accepts some partial models. To better specify the execution semantics of UML models, additional standards like fUML enforce more constraints, and will consider many legal UML models as invalid. Instead of extending UML with more precision, as fUML does, this approach extends it towards making less precision possible.

For this purpose, we have defined a list of *semantic relaxation points*, which make it possible to consider more partial models as valid when compared to the UML standard. In this paper, we introduce the following semantic relaxation points:

- **Instances before classifiers.** When designing a system it is often convenient to be able to think in terms of objects, before thinking in terms of classes [24, Section 13.5]. Some research has, for instance, already been conducted on inferring classes from sketchy object models [12]. This is similar to the notion of gradual abstraction in programming [14]. It is also useful to allow adding attributes and operations to objects.
- **Receiving undeclared messages.** Even though we allow adding operations to objects, we also allow calling undeclared operations, and for state machines to receive corresponding messages. This makes it possible to test a protocol before specifying it more rigorously with operation signatures.
- **Messages may transit through the *ether*.** In UML, calling an operation or sending a signal must be performed on a specific target instance, except for signal broadcast. However, broadcasted messages may be received by multiple objects. Received messages are placed in the event pool of each receiving object. This requires knowledge about which object will accept a given message, or making it acceptable by multiple possible recipients. Instead, we propose to allow sending individual messages without specifying a target object. Such messages end-up in what we call an *ether*, which is a kind of global event pool from which any object can retrieve a message.
- **Message triggers may *find* messages.** To be able to animate state machines that communicate with objects that are yet to be specified, it is necessary to allow them to find messages when needed to make progress. This means that transitions can be fired even though their triggers do not match any already sent message.
- **Time triggers may always match.** During animation, there is no fixed notion of time. It is therefore useful to allow transitions with time triggers to be fireable at all times.
- **Actions and expressions may be non-executable.** Because all attributes, operations, and parameters may not always be already modeled, we allow actions (e.g., transition effects, operation methods) to be non-executable. Calling an operation that no object can receive results in a lost message.
- **All guards may be considered true.** As a special case of non-executable expressions, guards may evaluate to false, or not evaluate at all (e.g., have an undefined result). Therefore, we allow transitions to fire even if their guards do not evaluate to true.

Designers do not need to use all of these semantic relaxation points. The objective is to gradually improve models so that they no longer need all this permissiveness. For instance, once the targets of all messages are known, the *ether* is no longer necessary. Similarly, once each received message is sent by an object, it is possible to stop allowing state machines to find messages that have not been sent yet. The main consequence of all these relaxations is that models do not need to be statically valid, as long as some kind of relaxed execution is possible.

Some relaxation points make it possible to consider a model that is incomplete because parts of the system it models are still missing (e.g., receiving undeclared messages). Other points make it possible to consider models that lack precision (e.g., guards accessing otherwise unused attributes may be considered true). Finally, we have not considered all possible kinds of model incompleteness yet, and the list of relaxation points could be extended in the future.

2.2 Execution Representation and Control

Partial models may be so incomplete as to prohibit traditional execution. For instance, what should happen for non-executable actions? Should they just be completely ignored? The results and effects of such a partial model execution cannot be the same as those of an actual program's execution for instance. This is achievable for complete fUML models, but not for partial models.

However, the UML standard offers several mechanisms to represent executions, such as sequence or timing diagrams. Therefore, we leverage these mechanisms, with some extensions, to show the result of a partial model execution. An *ether* lifeline may be added to sequence diagrams to explicitly show that messages have transited through it. Lost and found messages can already be represented on sequence diagrams, but require additional *virtual* timelines that do not correspond to actual objects.

Execution of partial models cannot generally proceed fully automatically. The main reason is that partial models can be highly non-deterministic. They may, for instance, find any message at

any time. Designers must be able to control the execution. As an example, if execution scenarios have been defined during a specification phase before starting the design phase, designers may want to attempt to reproduce them as sequence or timing diagrams by animating their models. This can be achieved by letting designers decide which transition to fire at each step. Moreover, it should be possible to explore alternative scenarios by restoring an earlier configuration, and pursuing animation from there.

Finally, the approach presented here also supports complete models that do not need semantic relaxation. Such models may be obtained by incrementally refining partial models. Complete models are also executable, but even more so. It may for instance make sense to perform automated analysis on them, such as state space exploration, or model-checking.

3 MODEL OF A SINGLE ACTIVE OBJECT

Let us consider Figure 1, which represents the state machine of a lamp. It starts in the `Off` state, where it waits for an `onButton` message. Upon reception of such a message, it sends the `turnOn` message, and enters the `On` state. It then waits for either an `offButton` message, or for 10 minutes to elapse. In both cases, it sends the `turnOff` message before returning to the `Off` state.

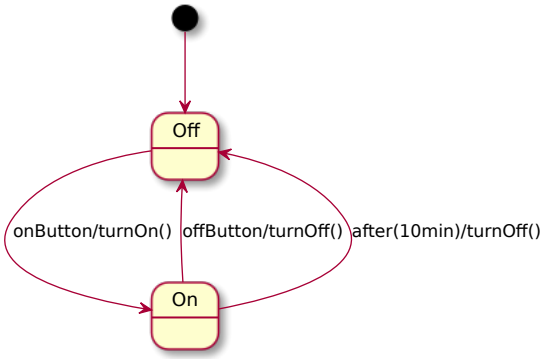
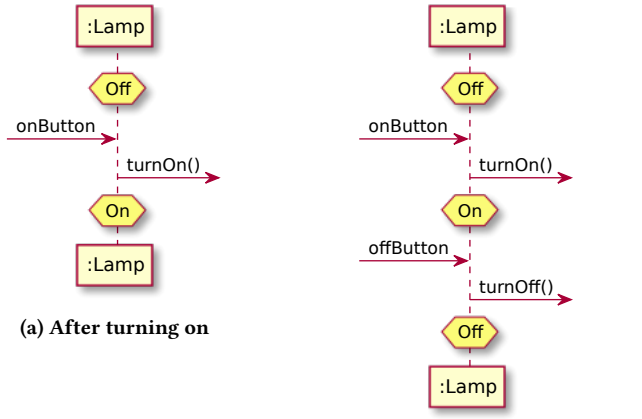


Figure 1: State machine of a lamp

Such a single standalone state machine is considered as executable even though no other modeled object can send the messages the lamp receives, or can receive the messages it sends. A possible execution is represented on Figure 2, as a sequence diagram with a single lifeline for the lamp. Current states at various points are shown as state invariants. Between the `Off` and `On` states, the lamp has received an `onButton` message from an unspecified source, and sent a `turnOn` message to an unspecified target. Something similar happens before returning to the `Off` state.

The sequence diagrams from Figure 2 can be produced by animating the state machine from Figure 1. For instance, when the lamp is in the `On` state (Figure 2a), the designer may be presented with the representation of the state machine given in Figure 3. On this figure, the current state is shown in a darker shade (green when colors are visible). The labels of fireable transitions are shown in the hyperlink style: underlined, and blue when colors are visible. The designer may then click on any one of the fireable transitions.



(a) After turning on

(b) After turning on then off

Figure 2: Sequence diagrams of an execution of the lamp

Clicking on the transition triggered by the `offButton` message results in the diagram from Figure 2b.

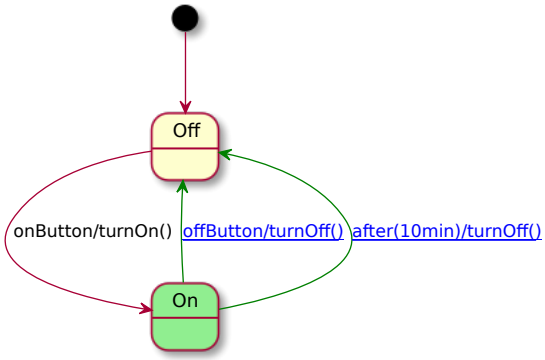


Figure 3: State machine of the lamp in the `On` state

4 INTERACTIONS BETWEEN MULTIPLE ACTIVE OBJECTS

Now that we have seen how an active object may be animated, let us consider interactions between active objects. The state machine of a button is represented on Figure 4. The button is always in the `Waiting` state from which it may either send the `onButton`, or the `offButton` messages. The transitions of this state machine have no trigger, and are therefore fireable at anytime. Such a state machine is typical to represent environment objects external to the modeled system.

Figure 5 shows how the button connects to the lamp, as well as the possible messages they can exchange as label of the link. The *«active»* stereotype denotes that both button and lamp are active objects. The small triangle to the right of the label indicates the direction of messages (from the base to the tip). Representing possible messages in this way is not necessary for state machine animation, but is shown here to illustrate the possibility. It is still

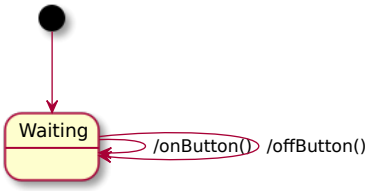


Figure 4: State machine of a Button

Figure 5: Architecture model of a button, and a lamp

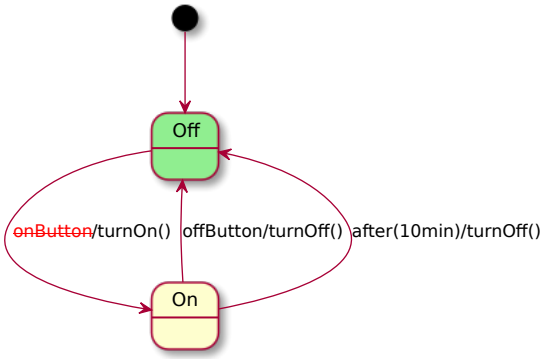


Figure 6: State machine of the lamp in the `Off` state

possible to animate the lamp and the button independently, by finding `onButton` and `offButton` messages.

However, because the button can actually send these messages, it becomes possible to restrict the animation by forbidding the triggering of transitions that do not match a previously sent message. Once this restriction is activated, the lamp in the `Off` state may be represented as shown on Figure 6. The current state is still denoted in the same way as on Figure 3, but no transition is shown as fireable. Instead, the `onButton` trigger of the transition from `Off` to `On` is stroked-through (and colored in red). The objective is to show the modeler that the transition is activated, because it goes out of the current state, but disabled because its trigger does not match any previously sent event. The designer can then fire the button transition that sends the `onButton` message, which makes the corresponding transition of the lamp fireable.

Figure 7 represents a possible execution of the button and lamp model as a timing diagram. Logical time steps are shown at the bottom of the figure. Messages sent by the button to the lamp are shown as slanted arrows between their timelines. They are not shown vertically to denote that some amount of time may have passed between their emission and reception. Messages sent by the lamp have no recipient, and end up in the *Unknown target* virtual timeline. Similarly, the *Unknown source* timeline could be used as the source of found messages. These virtual timelines are one possible

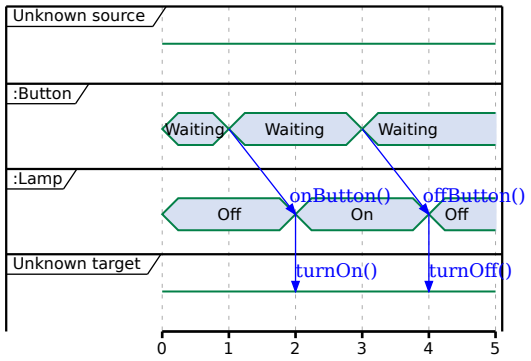


Figure 7: Timing diagram of an execution of lamp & button

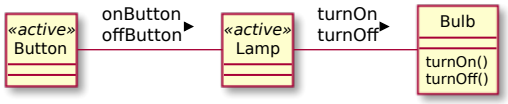


Figure 8: Architecture model of a button, a lamp, and a bulb

way to represent found and lost messages. Alternatively, found messages could be shown as coming from the top of the diagram, and lost messages as going to the bottom. This would be similar to their sequence diagram representation in Figure 2.

If the state machine of the button had not been modeled yet, a timing diagram like the one from Figure 7 could still be produced, by letting the designer click on the `onButton`, and `offButton` messages shown on Figure 5. Conversely, a timing diagram could also be produced without having modeled the connection between button and lamp.

5 ADDITION OF PASSIVE OBJECTS

Not all objects are active, and some simply have operations that are executed by the caller at any time, without state machine constraints. Figure 8 shows how one can add a bulb that may receive the `turnOn`, and `turnOff` messages sent by the lamp. Adding such a passive object makes it possible to produce executions, as well as corresponding sequence or timing diagrams, in which these messages are actually received.

Additionally, operations may have methods, such as those shown on Figure 9 for the bulb. Such methods could also send messages, but are here only logging some text to a console. Actual operation execution may be set to automatic for non-ambiguous models, but may be under the designer's control for partial models. In the latter case, operations may be shown in a hyperlink style, like fireable transitions. The designer may then choose to execute them at any time, or to restrict their execution to situations where corresponding messages have been previously sent. This is similar to the way transitions behave, and makes it possible to start animating the model, and producing sequence or timing diagrams relatively early during the modeling process.

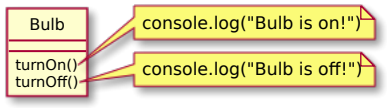


Figure 9: Bulb with methods

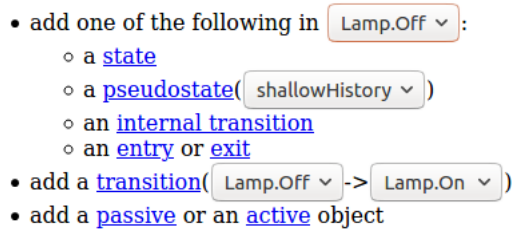


Figure 10: Edition actions on the lamp

6 TOOL SUPPORT

To make it possible to evaluate the approach, we have implemented a proof-of-concept tool, called AnimUML, supporting it. AnimUML has been used to animate more than thirty relatively small models, including multiple variants at different levels of completion, such as those presented as examples in this paper. A few slightly larger models inspired from the literature [4, 11, 13, 21] have been successfully loaded as well, notably to test the model-checker connection. However, this is still a very early version of the tool, and evaluation has thus far been limited to testing that everything mostly works as intended. It has nonetheless been quite useful in refining the approach, more especially regarding the kinds of interactions offered to users. The tool is available on GitHub[1].

Tool Implementation. This tool is implemented in HTML, JavaScript, and SVG, and based on an *ad hoc* implementation of a simplified version of the UML metamodel. Model actions (e.g., operation methods, transition effects) must be specified in JavaScript, but may be only partially executable. AnimUML leverages JavaScript's Proxy mechanism [8, Section 26.2] in order to capture calls to undefined operations, or accesses to undefined attributes. These may then be represented on a sequence or timing diagram. It uses PlantUML[2] to generate diagrams with automatic layout.

AnimUML Settings. Editing models is possible via a list of possible actions, as illustrated on Figure 10. Elements can be added to the model (e.g., state, transition, object). The diagram is then updated by reserializing the model to the PlantUML format, and asking it to produce a new picture. Additional actions are shown directly on the model when edition mode is enabled. For instance, clicking on a state name, a transition trigger, guard, or effect, makes it possible to edit it. It is also possible to create or edit models in JSON, and load them in AnimUML. Moreover, we have implemented a conversion from Eclipse UML models to AnimUML-compatible JSON format to make it possible to load existing models.

A screenshot of the tool is shown on Figure 11. The top-left corner shows the general settings. Built-in models can be selected using a drop-down list, here showing that the *ButtonLampBulbWithMethods* model is loaded. Specific active objects can be selected using another drop-down list, here showed with all objects selected. More specific settings, such as requiring triggers to match previously sent messages or not, can be accessed by clicking on the *Settings* link. Model edition is enabled by clicking on the *Edit* hyperlink, which opens the list of actions from Figure 10. PlantUML diagrams may be exported by clicking on the *PlantUML* hyperlink, with or without annotations (e.g., current state and fireable transition). Finally, the *export* hyperlink lets users get a link with a JSON serialization of the currently loaded model embedded. This link may be bookmarked, or sent by chat or email to collaborators, who can then load the same model in their web browser. The model is shown on the bottom left corner. There are multiple display settings (e.g., with or without classes, state machines, or methods), but it is fully displayed (although cropped) here, with active object state machines shown in notes. The current execution history is shown on the right-hand side of the figure after a small list of actions. The model may be *Reset* into its initial configuration, the diagram may be exported in *PlantUML*, and a drop-down list lets users select whether to show a timing or a sequence diagram. The *ether* is represented explicitly on this figure, with messages exchanged between objects passing explicitly through it. Additionally, it can be seen that the lamp's time-triggered transition (see Figure 1) has been fired. This is shown on the sequence diagram as an interruption of all lifelines across which *after(10min)* is written to show that some time is supposed to have elapsed. State invariants are represented as hyperlinks because clicking on them makes it possible to rollback to an earlier configuration.

All figures in this paper have been exported from AnimUML, either as screenshots (Figures 10 and 11), or as PlantUML exports (all others). PlantUML codes exported from AnimUML have however been slightly hand-edited for simplification reasons.

Connection to Third-party Tools. AnimUML can be connected to external tools by leveraging the TCP protocol from OBP2[3] [3, 6]. This mostly only requires proxying the TCP connection over a WebSocket [10] to make it possible to establish a connection from a web page. We have thus far successfully tested remote simulation, state space exploration, deadlock detection, as well as model-checking of an LTL property. This often requires limiting the number of messages in event pools or the *ether*. We have, for instance, experimented with allowing at most one unconsumed message of each kind (as identified by its name). In general, such relatively advanced verification only makes sense for models that are relatively close to completion rather than for highly incomplete ones. It has nonetheless proven useful to detect typos in models, or bugs in the animation engine. Moreover, it is not only possible to connect AnimUML to analysis tools such as simulators, or model-checkers, but it is also possible to connect it to other execution engines, by reversing the protocol commands and responses. We have thus successfully tested connecting AnimUML to OBP2's UML engine[4] [3]. This makes it possible to use the same AnimUML interface to animate early-stage design models, or to control actual execution on an embedded platform.

[1] https://github.com/fjouault/AnimUML
[2] https://plantuml.com/
[3] http://www.obpcdl.org/
[4] http://www.obpcdl.org/bare-metal-uml/

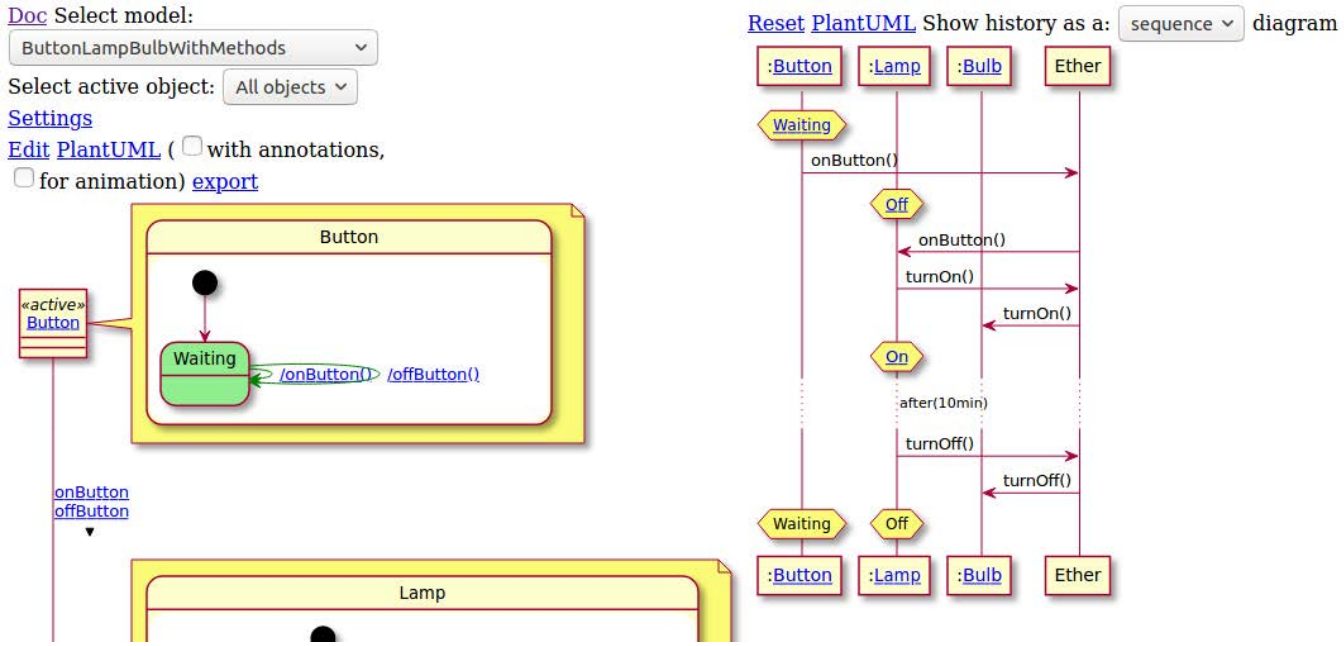


Figure 11: Tool screenshot

7 RELATED WORK

AnimUML provides a JavaScript execution engine for partial execution of UML models. Our work is in line with ongoing work on Eclipse Che and Theia [20], and Sirius [23] that aims at providing online programming and modeling environments. The notion of "partial models" has also been used in [9] in relation with model uncertainty. This is a different notion of partial models from ours, although it might be interesting to consider the execution of their kind of partial models. In [7], a systematic review gives an overview of current state-of-the-art UML execution engines. The Gemoc Studio [5] and the Embedded Model Interpreter (EMI) [3] are two execution frameworks that enable to execute and analyze execution of complete models but they do not provide facilities for executing partial models online in a web browser. Some approaches already support semi-automatic model improvement, such as [12] that infers classes given some sketchy object models. This is complementary to our approach, which requires designers to perform these improvements, but helps them by animating their models. The notion of gradual abstraction in programming [14] is similar to the notion of partial model execution as presented in this paper, however our approach supports models that are even less complete. For instance, it is possible to call undeclared operations. Partial models are also used in live modeling approaches [1, 19, 22] that allow to change the runtime model to get immediate feedback while avoiding the tedious "edit-compile-run" cycle. Such approaches seem complementary to our work but they have not yet been used for executing and analyzing models online. Last but not least, the work in [2] defines a tool called PMExec that supports the execution of Partial UML-RT models. Using static analysis, the tool adds decision points where information is missing such that the user or an automated script provides the missing data during model execution. The tool supports model debugging and has code generation facilities but does not go as far as model verification. As a result, to the most of our knowledge, AnimUML is the first web application that provides an execution engine for partial UML models with animation, simulation, and formal verification support.

8 CONCLUSION

This paper has presented an approach that makes it possible to animate early-stage design models, even though they may be incomplete and inconsistent. Animating such models is actually useful to discover inconsistencies, or missing elements. This kind of tool also brings benefits in assisting designers during the intellectual creation phase of design models, and in informal exchanges between the different designers (i.e., to share ideas) by removing barriers and the need to use formal aspects. A tool supporting the approach has also been described.

The approach could be extended by integrating automated model improvement techniques, such as inferring classes from objects [12]. It would also benefit from being evaluated with actual users. Because it helps understand UML models, it may also be useful for the purpose of teaching UML.

The tool could be improved too, by supporting a larger subset of UML, the export of UML models (import is already partially supported), or some form of code generation. One point not discussed in the paper is that other kinds of model visualisation can also be integrated. For instance, we have implemented the possibility to visualize a flattened version of composite state machines. This also helps understanding their semantics. Finally, an execution semantics for partial models could be formalized.

REFERENCES

[1] M. Bagherzadeh, K. Jahed, B. Combemale, and J. Dingel. 2019. Live-UMLRT: A Tool for Live Modeling of UML-RT Models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 743–747. https://doi.org/10.1109/MODELS-C.2019.00115

[2] Mojtaba Bagherzadeh, Karim Jahed, Nafiseh Kahani, and Juergen Dingel. 2019. PMExec: An Execution Engine of Partial UML-RT Models. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1178–1181. https://doi.org/10.1109/ASE.2019.00131

[3] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. 2018. Unified LTL Verification and Embedded Execution of UML Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*. Copenhagen, Denmark. https://doi.org/10.1145/3239372.3239395

[4] Frédéric Boniol and Virginie Wiels. 2014. Landing Gear System Case Study. In *Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014, Proceedings*. 1–18.

[5] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution Framework of the GEMOC Studio (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (Amsterdam, Netherlands) *(SLE 2016)*. ACM, New York, NY, USA, 84–89. https://doi.org/10.1145/2997364.2997384

[6] Mihal Brumbulli, Emmanuel Gaudin, and Ciprian Teodorov. 2020. Automatic Verification of BPMN Models. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*. Toulouse, France.

[7] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. 2018. Execution of UML models: a systematic review of research and practice. *Software & Systems Modeling* (10 April 2018). https://doi.org/10.1007/s10270-018-0675-4

[8] ECMA International. 2011. *Standard ECMA-262 - ECMAScript Language Specification* (5.1 ed.). http://www.ecma-international.org/publications/standards/Ecma-262.htm

[9] M. Famelis, R. Salay, and M. Chechik. 2012. Partial models: Towards modeling and reasoning with uncertainty. In *2012 34th International Conference on Software Engineering (ICSE)*. 573–583.

[10] I. Fette and A. Melnikov. 2011. *The WebSocket Protocol*. RFC 6455. RFC Editor. http://www.rfc-editor.org/rfc/rfc6455.txt http://www.rfc-editor.org/rfc/rfc6455.txt.

[11] Frank Houdek and Alexander Raschke. 2020. Adaptive Exterior Light and Speed Control System. *Case study for the ABZ 2020 conference* (2020).

[12] Andreas Kästner, Martin Gogolla, and Bran Selic. 2018. From (Imperfect) Object Diagrams to (Imperfect) Class Diagrams: New Ideas and Vision Paper. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3239372.3239381

[13] Alexander Knapp, Stephan Merz, and Christopher Rauh. 2002. Model Checking Timed UML State Machines and Collaborations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Werner Damm and Ernst Rüdiger Olderog (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 395–414. https://doi.org/10.1007/3-540-45739-9_23

[14] Kurt Nørmark, Lone Leth Thomsen, and Bent Thomsen. 2012. Object-Oriented Programming with Gradual Abstraction. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/2384577.2384583

[15] OMG. 2017. Precise Semantics of UML State Machines. https://www.omg.org/spec/PSSM/1.0/Beta1/PDF

[16] OMG. 2017. Semantics of a Foundational Subset for Executable UML Models. https://www.omg.org/spec/FUML/1.3/PDF

[17] OMG. 2017. Unified Modeling Language. https://www.omg.org/spec/UML/2.5.1/PDF

[18] OMG. 2019. Precise Semantics of UML Composite Structures. https://www.omg.org/spec/PSCS/1.2/PDF

[19] Riemer Rozen and Tijs Storm. 2019. Toward Live Domain-Specific Languages. *Softw. Syst. Model.* 18, 1 (Feb. 2019), 195–212. https://doi.org/10.1007/s10270-017-0608-7

[20] R. Saini, S. Bali, and G. Mussbacher. 2019. Towards Web Collaborative Modelling for the User Requirements Notation Using Eclipse Che and Theia IDE. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. 15–18.

[21] Timm Schäfer, Alexander Knapp, and Stephan Merz. 2001. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science* 55, 3 (2001), 357 – 369. https://doi.org/10.1016/S1571-0661(04)00262-2 Workshop on Software Model Checking (in connection with CAV '01).

[22] Yentl Van Tendeloo, Simon Van Mierlo, and Hans Vangheluwe. 2019. A Multi-Paradigm Modelling Approach to Live Modelling. *Softw. Syst. Model.* 18, 5 (Oct. 2019), 2821–2842. https://doi.org/10.1007/s10270-018-0700-7

[23] V. Viyović, M. Maksimović, and B. Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. 233–238. https://doi.org/10.1109/INES.2014.6909375

[24] Rob Williams. 2005. *Real-Time Systems Development*. Butterworth-Heinemann, USA.