Software Quality Journal (2023) 31:575–617
https://doi.org/10.1007/s11219-023-09617-5

Assessing the specification of modelling language semantics: a study on UML PSSM

Márton Elekes[1] · Vince Molnár[1] · Zoltán Micskei[1]

Accepted: 29 January 2023 / Published online: 1 March 2023

Abstract
Modelling languages play a central role in developing complex, critical systems. A precise, comprehensible, and high-quality modelling language specification is essential to all stakeholders using, implementing, or extending the language. Many good practices can be found that improve the understandability or consistency of the languages' semantics. However, designing a modelling language intended for a large audience is still challenging. In this paper, we investigate the challenges and typical issues with assessing the specifications of behavioural modelling language semantics. Our key insight is that the various stakeholder's understandings of the language's semantics are often misaligned, and the semantics defined in various artefacts (simulators, test suites) are inconsistent. Therefore assessment of semantics should focus on identifying and resolving these inconsistencies. To illustrate these challenges and techniques, we assessed parts of a state-of-the-art specification for a general-purpose modelling language, the Precise Semantics of UML State Machines (PSSM). We reviewed the text of the specification, analysed and executed PSSM's conformance test suite, and categorised our experiences according to questions generally relevant to modelling languages. Finally, we made recommendations for improving the development of future modelling languages by representing the semantic domain and traces more explicitly, applying diverse test design techniques to obtain conformance test suites, and using various tools to support early-phase language design.

Keywords UML · Modelling language · Semantics · State machine · Testing · Conformance

✉ Zoltán Micskei
micskeiz@mit.bme.hu

Márton Elekes
elekes@mit.bme.hu

Vince Molnár
molnarv@mit.bme.hu

[1] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Müegyetem rkp. 3., Budapest H-1111, Hungary

1 Introduction

Context In *model-based engineering* (MBE), models play the central role in information exchange and serve as the base of derived artefacts and analysis (Brambilla et al., 2012). Models are expressed using various *modelling languages*. Modelling languages are defined by specifying (1) the possible elements of the language and their connections (*abstract syntax*), (2) how they can be used to construct well-formed models (*well-formedness constraints*), (3) what graphical or textual notation to use to represent these elements (*concrete syntax*), and (4) what a well-formed model means (*semantics*). However, to use models for simulation, verification or generation, defining the *precise semantics* of the modelling language is essential (Broy & Cengarle, 2011).

The Object Management Group's (OMG) *Unified Modeling Language* (UML) (OMG, 2017) is a general-purpose modelling language widely used to describe software systems. UML defines several diagrams for visualising different aspects of systems. The initial release of the language specification concentrated on the syntax and visual notations to use. To relieve the lack of a precise semantic definition of the language, the OMG made a great effort to create the *Semantics of a Foundational Subset for Executable UML Models* (fUML) specification (OMG, 2011), which is an executable subset of UML to define the exact structural and behavioural semantics of systems. The *Precise Semantics of UML State Machines* (PSSM) specification (OMG, 2019) extends fUML with execution semantics for state machines.

Motivation Specifying a general-purpose, complex modelling language for widespread use by thousands of professionals and organisations is highly challenging. Several methods have been proposed for specifying modelling languages with various levels of formalisation, but the prevailing method is still primarily based on *extensive natural language specification* (Bork et al., 2020).

Having a "high quality", precise definition of a modelling language is crucial for all the different stakeholders:

- *Model users*, i.e. professionals using the modelling language, need to understand the language to create meaningful, correct models.
- *Developers* need to implement the models or modelling, simulation and analysis tools to conform with the details of the specification.
- *Language designers* need to carefully design or update each feature and consider its impact on other language features and their joint behaviour.

Organisations specifying general-purpose, widespread modelling languages have realised that more precise definitions of the syntax and semantics of languages are needed. For example, if we observe the history of the modelling languages developed by OMG, we can see several improvements to the process and format of language specification. The abstract syntax is defined using formal languages (context-free grammars or metamodels) extended with well-formedness constraints (e.g. in Object Constraint Language). A machine-readable version of the metamodel is now part of every specification in a standardised interchange format. The fUML and subsequent standards added a semi-formal, operational semantics to refine the semantics defined with natural language in the UML specification. Reference implementations were created during the development of the language. Finally, the PSSM specification provided an extensive test suite to illustrate the semantics and validate that model execution tools conform to the PSSM semantic model.

Having semi-formal semantics, reference implementations and test suites are significant improvements for specifying modelling languages. They offer support for model users to understand some nuances of the semantics, and for validating tool conformance but only for a subset of the semantics. However, even in state-of-the-art specifications, there are still numerous issues regarding the definition of semantics.[1] Important questions such as the followings arise during the development or revisions of complex modelling languages.

- Can model users understand the semantics from the specification?
- Are the different semantics described in the various artefacts consistent?
- What level of tool conformance can we validate using the test suites?

The reason behind these issues and questions could be that less work was directed towards assessing the modelling language specification itself than its usage. We need verification and validation (V&V) techniques and activities for the modelling language, and not just for the derived artefacts (i.e. system models created with UML or modelling tools implementing the specification).

Modelling language specifications are requirement and design documents. Therefore they shall adhere to typical quality criteria: they shall be consistent, complete, verifiable, and understandable (ISO/IEC/IEEE, 2018). Common review techniques could be applicable also to modelling language specifications. However, good practices for modelling language development (Czech et al., 2020) and previous works on testing modelling languages (Ratiu et al., 2018) were more focused on the tooling for the language (editors, code generators) and not specificities of modelling language specification documents.

Therefore the research question that motivated our work is the following.

What are the challenges when assessing the specifications of modelling language semantics?

Method To analyse the specificities of modelling language specifications from a quality point of view, we built on our experience in designing modelling languages' semantics, building verification tools and teaching semantics to model users. We (1) investigated practices from recent modelling language specifications, (2) performed a case study on assessing the artefacts of the PSSM specification, and (3) based on these insights, synthesised general concepts and challenges of modelling language's semantics.

1. We considered the major stakeholders (model users, tool developers and language designers), their motivations and viewpoints with respect to the semantics of the language. We collected the various artefacts supporting the definition of a modelling language, and how each artefact can be used to refine or communicate the semantics of the language.
2. We used the PSSM specification (OMG, 2019) as a case study. We choose PSSM because it is a recent specification containing several of the above artefacts. We reviewed the document along standard review criteria (e.g. consistency, unambiguity); taking into account the different stakeholder views based on our experience in those roles; executed the tests in Cameo Systems Modeler; and analysed the consistency between the various artefacts. As PSSM is quite complex, we focused on selected key areas instead of aiming for a complete review.

[1] Issues for OMG specifications can be browsed publicly: https://issues.omg.org/issues/lists

3. Based on the issues and insights from the case study that can be generalised, we synthesised challenges about specifying semantics and ways to assess and improve the definition and understanding of semantics.

The findings in this paper capture our experience gathered in the course of various modelling-, specification-, and verification-related activities from the last decade. For example, we analysed the semantics of UML Sequence Diagrams (Micskei & Waeselynck, 2011), designed a statechart language (Graics et al., 2020), and built verifiers for UML/SysML (Horváth et al., 2020). Moreover, we taught numerous courses on model-based design for students and professionals, worked with modellers from automotive, railway, and aerospace companies, and recently are contributing to OMG's new SysMLv2 language. Thus, we have first-hand experience with various stakeholder roles, which provides a deep, although potentially biased view of the topic. We intend this paper to facilitate further discussions and provide input for formulating hypotheses and research questions in potential future empirical studies.

Note that we concentrated on the semantics of a behavioural modelling language because we had more experience with the operational semantics of behavioural languages, but a similar process can be carried out for structural languages (e.g. UML composite structures or SysML block definitions).

The insights from the case study helped us to identify typical issues with the semantics, practical methods for cross-checking the various artefacts, and synthesise why these issues and inconsistencies happened in the first place.

Results Our main finding is that as the semantics of complex modelling languages is still mainly described with informal texts (even if there are numerous additional semi-formal artefacts), a key challenge is how the various *stakeholders understand* these fragmented semantic descriptions and how their understanding of the same semantics might misalign. Therefore, assessment of the semantics should concentrate on identifying and decreasing these *misalignments in the understandings* and *inconsistencies in the artefacts*.

Moreover, by performing a detailed analysis of the UML PSSM specification, we identified typical issues in modelling language semantics: unclear representation of execution traces, errors due to manual specification of expected behaviours, or issues with the joint behaviour emerging from the individual specifications of multiple elements. We found that the test suite helped a lot to understand difficult parts of the specification. However, it contained several significant typos, some valid execution traces were missing, and some aspects of the semantics remained undefined even by cross-checking all the relevant semantic descriptions and test cases.

Based on these experiences, we proposed recommendations for specifying future modelling language specifications. The most important ones are (1) the need for a clear representation of the semantic domain, (2) finding the right abstraction level for execution traces, (3) better processes and tools to derive test oracles and list of alternative traces, (4) using additional test methods to design test suites for the semantics, and (5) recommendations about further tooling to support assessment.

Extension of previous work This paper extends our preliminary workshop paper (Elekes & Micskei, 2021), which focused on the PSSM test suite itself. This paper discusses assessing the whole modelling language specification and not just testing the modelling language test suite, and contains a more detailed investigation of the PSSM specification.

Conclusion Having precise semantics and the appropriate tooling for a modelling language is crucial. Semi-formal semantic definitions, detailed test suites and reference implementations for the modelling language increase the understandability and interoperability significantly, as illustrated by the PSSM specification. However, our results highlighted that defining a complex modelling language specification is still error-prone. We hope that revealing the impact of possible misalignments and inconsistencies, the insights from the assessment of PSSM, and our recommendations can support future specifications to be more precise, comprehensible, and easier to use.

Structure of the paper Section 2 gives an overview of UML state machines and the PSSM specification. Section 3 collects the viewpoints of different stakeholders and artefacts, then presents how misalignments and inconsistencies can be assessed regarding the modelling language's semantics. Section 4 details the assessment of the PSSM specification and collects typical issues. Section 5 contains our recommendations for defining and assessing future modelling specifications. Section 6 collects the related work on modelling languages, semantics and testing. Finally, Section 7 concludes the paper.

2 Overview of UML PSSM

This section gives a short overview of the history of UML (Section 2.1), the informal semantics of state machines (Section 2.1), the semantic description in PSSM (Section 2.3), the test suite in PSSM (Section 2.4), and the various tools for PSSM (Section 2.5).

2.1 Short history of UML state machines

The *Unified Modeling Language* (UML) was created in the second half of the 1990s to unify the various notations and approaches used for describing software design. The first version of UML was standardised in 1997 as UML 1.1, which was refined in several versions in the coming years (with releasing UML 1.5 in 2003). UML 1.x included state machines as an object-based variant of Harel statecharts (Harel, 1987). The abstract syntax of the language was defined using class diagrams, additional well-formedness rules with Object Constraint Language (OCL), and semantics using natural language.

The 2.0 version of UML consolidated the various notations in UML (collaboration, state machines...) to a common semantic foundation (Selic, 2004, 2012). Class descriptions were more detailed, but the description of the semantics was scattered in the alphabetically listed individual descriptions of each class. This problem was resolved in version 2.5 with the "UML Simplification" initiative, which merged the separate Infrastructure and Superstructure documents and consolidated the description of the semantics to common sections, where information was listed in a logical order (instead of alphabetical).

In the meantime, OMG issued a Request for Proposal for the "*Semantics of a Foundational Subset for Executable UML Models*" to have more precise semantics for the core of the UML language. The resulting Foundational UML subset (fUML) (OMG, 2011) specification was adopted in 2011. The specification contained a subset of the classifiers, common behaviour (events), actions and activities elements of UML. The fUML specification defined a general execution model and an operational semantics for the language elements in the selected subset. The execution model is itself an executable, object-oriented fUML model containing classes describing the executions of the various UML elements. The resulting specification is a more precise, although not easily understandable definition of

the intended semantics. An important design direction was that fUML would not change the semantics already published. Instead, fUML would only more precisely define the semantics of the previous specifications and explicitly list cases where it restricts the previous behaviour.

Building on the fUML specification, the *Precise Semantics of UML State Machines* (PSSM) specification extended the execution model and operational semantics to state machines (OMG, 2019). PSSM is a direct extension of fUML, e.g. specialising execution classes defined in fUML. A significant part of the specification is a test suite containing 103 test cases that are given as examples for various parts of the semantics and can be used to check the conformance of state machine execution tools.

Cook (2012) provided a detailed description of the history and evolution of the UML language up to 2012, which we recommend for interested readers.

2.2 Informal semantics of UML state machines

For event-driven behavioural modelling, UML defines state machines that model finite automata (OMG, 2017, Clause 14). Since the basics of state machines are known well, in the following, we give only a simplified introduction to the elements we use in this paper based on the official specification.

A StateMachine comprises one or more (orthogonal) Regions, each Region containing a set of Vertices interconnected by Transitions. Vertices can be States, FinalStates and Pseudostates (initial, etc.). States can be simple states or composite states, which can comprise one or more Regions. A State contained in a Region of another composite state either directly or indirectly through other composite states is called a *substate* of the composite state. A State may have *entry* and *exit* Behaviors, which are executed when the State is entered or exited (resp.). A State may have a *doActivity* Behavior, which commences execution concurrently when the State is entered and the *entry* Behavior has completed (if given).

A Transition is a directed arc from a *source* Vertex to a *target* Vertex (can be the same). It may have an *effect* Behavior, which is executed when the Transition is traversed. A Transition is *enabled* if its source State is in the active state configuration and it has a Trigger matching the dispatched Event occurrence. If more than one Transition is enabled within a StateMachine, they may be in conflict with each other. In this case, transitions from direct and indirect substates have priority over containing States. Only transitions in mutually orthogonal Regions may be fired simultaneously. A Transition may have a *guard* Constraint. A Transition with a guard that evaluates to `false` is disabled.

An executing StateMachine instance is in exactly one state configuration at a time, called the *active state configuration*. When an Event occurrence is recognised, it is stored in the *event pool*. When the execution comes to a wait point (stable state configuration) where it needs a trigger to continue, the event pool is examined. If the pool contains an event occurrence that matches one of the Triggers, the occurrence is removed from the pool and dispatched to the state machine. A StateMachine *step* is executed if there is at least one enabled Transition that the Event occurrence can trigger. A step involves executing a transition and terminating in a stable state configuration, i.e. the next wait point. The transition can be a *compound transition,* which chains multiple transitions into a more complex one through Pseudostates, e.g. when entering a composite state via the initial Pseudostate and initial transition. This step is called a *run-to-completion step* (RTC). Run-to-completion

means that a pending Event occurrence is dispatched only after the processing of the previous occurrence is completed and a stable configuration has been reached.

A special kind of Transition is a *completion transition*, which is triggered by a *completion event* of its source state (`CE(source)`). A state generates a completion event if all of its internal activities have completed (e.g. entry and doActivity Behaviors) and all of its orthogonal Regions have reached a FinalState (in case of composite state). Completion events have dispatching priority over regular events.

In UML, a State may specify a set of Events that are *deferred* in that State. These Event occurrences are not dispatched and remain in the event pool until the Events are no longer deferred by any state in the state configuration, or a Transition from the deferring state is triggered by the Events in order to override the deferring constraint.

2.3 Precise operational semantics of fUML and PSSM

As PSSM extends the execution model of fUML, the precise semantics of fUML is presented briefly. The execution of an fUML model is done by an *execution engine*. The specification defines this execution engine using class diagrams specifying the classes and operations of the engine. The details of the operations are specified with detailed activities described using a surface notation similar to the Java language. Hence the specification contains low-level, executable operational semantics for a subset of UML.

The fUML specification uses *visitor classes* to define how a specific behaviour shall be executed. For example, one of those visitor classes defines how and when the activity's nodes should be activated. So-called activation visitor classes define the semantics of the various model elements (e.g. what should happen when a final node is fired). The specification contains additional classes representing elements of the *semantic domain* that are not UML model elements themself (e.g. Token that is an essential concept in the token-game dataflow semantics of activities). *Non-deterministic choices* are modelled in the operational semantics using `ChoiceStrategy` classes (although, as we would see later in the paper, not every choice is modelled explicitly).

The Common Behavior package defines how active objects communicate with each other using signals or operation calls. Active objects can act *asynchronously*. The semantics of an active object defines how the object's event pool is handled (e.g. dispatching events from the pool). Matching events are handled by so-called *event accepter*s.

The PSSM specification defines classes that extend the execution model of fUML. PSSM defines that state machines have two separate event pools, one for "normal" event occurrences and one for deferred ones. The active object representing state machines contains specific methods to guarantee that completion events are dispatched first. A single, special event accepter is used for state machines, which is responsible for collecting matching transitions, calculating priorities, resolving conflicts and selecting the set of transitions to fire in case of orthogonal regions. Visitor classes are used to define the semantics of each state machine element (e.g. state, region, transition).

2.4 Overview of the PSSM test suite

The PSSM specification (OMG, 2019) contains 103 test cases grouped into 18 categories relating to specific parts of state machines. The tests were manually created by experts based on 113 requirements extracted from the normative text of the UML 2.5.1

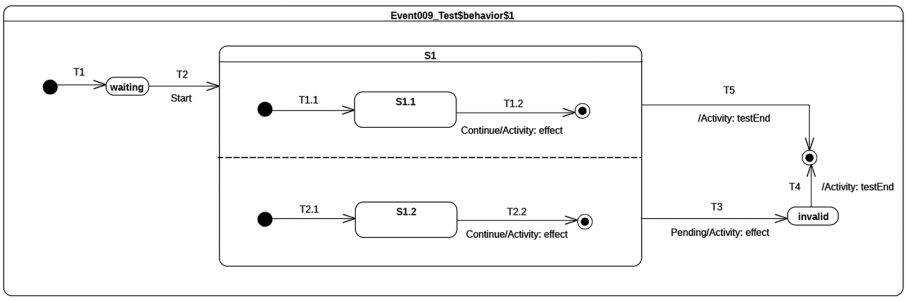


Fig. 1 Target state machine for a test case (OMG, 2019, Event 009). Input events: `Start`, `Continue`, `Pending`

specification. The PSSM specification defines a traceability matrix showing the coverage of the requirements.

A test case in the PSSM test suite consists of the following:

- A *target* state machine is the system under test (SUT). The target receives and dispatches events of the stimulation sequence, which will trigger transitions. Throughout its execution, the state machine generates an execution *trace*, which is used to evaluate the outcome of the test case.
- A *tester* that encodes the stimulation sequence (i.e. a series of event occurrences, e.g. `Start`, `Continue`), which is the input sent to the target.
- A *semantic test* instantiates and controls the tester and the target. After the execution of the SUT, the semantic test compares the execution trace to manually defined expected one(s) and marks the test passed or failed.

A test case is identified by its category, a 3-digit number, and optionally a letter for subtests. Figure 1 shows the target state machine of test case Event 009, which checks the following requirements: multiple transitions (in different regions) can be triggered by the same event occurrence, and their execution order is left undefined As an explanation, PSSM describes the RTC steps realised during the execution (Table 1). Each step contains the status of the event pool, the state machine configuration, and the transition(s) fired during the RTC step. The default dispatching strategy for the event pool is FIFO, and the dispatched event in the front is denoted with **boldface**.

Table 1 RTC steps of test case Event 009 (OMG, 2019)

Step	Event pool	State machine configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(waiting)**]	[waiting]	[]
3	[**Start**]	[waiting]	[T2(T1.1, T2.1)]
4	[Pending, Continue, CE(S1.2),**CE(S1.1)**]	[S1[S1.1, S1.2]]	[]
5	[Pending, Continue, **CE(S1.2)**]	[S1[S1.1, S1.2]]	[]
6	[Pending, **Continue**]	[S1[S1.1, S1.2]]	[T1.2, T2.2]
7	[Pending, **CE(S1)**]	[S1]	[T5]

The execution is the following: the tester sends as the stimulation sequence a `Start` signal when the state configuration is [`waiting`], a `Continue` and a `Pending` signal in state configuration [`S1`[`S1.1`, `S1.2`]]. Once the first state `waiting` is reached, a completion event (`CE(waiting)`) is generated. The completion event is given priority over non-completion events. Therefore it is dispatched first: in lack of completion transition (transitions without an external triggering event, here `T4` and `T5`) to be triggered from state `waiting` the event is discarded. In state `waiting`, a `Start` event is dispatched from the event pool, which triggers transition `T2`, and the state machine enters the orthogonal regions of `S1`. Thus the configuration changes to [`S1`[`S1.1`, `S1.2`]]. The completion events of `S1.1` and `S1.2` are discarded without triggering a transition. The dispatching of `Continue` event simultaneously triggers transitions `T1.2` and `T2.2` in the two orthogonal regions. Both regions and state `S1` complete as the regions reach final states, which generates completion event `CE(S1)`. As the completion event has priority, it triggers the completion transition `T5`. `Pending` will never be dispatched because there is no longer a transition that it can trigger, and the state machine finishes its execution. Some parts of the state machine contain activities (i.e. effects of transitions, entry/exit/doActivity behaviours of states) to generate the execution trace. PSSM specifies the expected trace for every test, in this case `T1.2(effect)::T2.2(effect)` and the alternative trace `T2.2(effect)::T1.2(effect)`, i.e. both orders are possible due to the simultaneous firing of the two transitions on `Continue` in the orthogonal regions, but `T3` cannot be traversed.

A PSSM test case contains many details which illustrate and check some part of the semantics. However, as we would see later in Section 4, this description sometimes still misses key details and concepts that hinders understanding and cross-checking the semantics. For example, as only the firing of transitions `T1.2` and `T2.2` are written in the trace, we cannot be sure about the order of `T1.1` and `T2.1`. If `T1.1` fires first, does it mean that the upper region always steps first and `T1.2` should also come first? Such specific questions about corner cases could only be answered with more directed test cases or more detailed descriptions (e.g. the RTC table could define the internals of an RTC step).

2.5 Tools: Implementations, simulators and verifiers

One of the novelties of fUML specification's development was a *reference implementation*[2] created in parallel with the text of the specification. This software takes an XMI (XML Metadata Interchange) description of a UML model and provides an execution trace for selected activities. The reference implementation prints out only one execution trace, even if multiple possible alternative traces exist when executing an activity. The reference implementation used during the development of the PSSM specification was an open-source component for the Moka plugin[3] of Eclipse Papyrus. The Java code implementing the methods of the execution model's classes is more or less the same as the one in the semantics model XMI of the specification.

Simulator tools usually offer visualisation and debugging capabilities among model execution. As far as we know, Cameo Simulation Toolkit (CST) is the only commercial UML modelling tool stating that it simulates state machines according to the PSSM semantics. CST offers a graphical user interface for simulating state machines, but generally it produces only one simulation trace in case of parallelism or non-deterministic choices.

[2] fUML implementation: https://github.com/ModelDriven/fUML-Reference-Implementation

[3] Eclipse Papyrus Moka: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

Formal *verifier* tools can check some properties of the model (Micskei et al., 2014). These tools usually analyse all or many possible traces for a given model. Thus they can reason whether a specific state configuration is reachable or whether there is a deadlock in the model. There are verifier tools for subsets of fUML (Lima et al., 2020) or SysML (Horváth et al., 2020).

3 Challenges of assessing modelling language semantics

In this section, we examine the specificities of developing modelling language specifications in order to highlight the increased importance of human understanding. While programming languages can be regarded as a form of communication between humans and machines, the use of modelling languages is closer to natural languages in the sense that their primary role is (a more formalised, potentially computer-assisted) communication between different stakeholders of the system development process. In this sense, the equivalent of an execution platform for a programming language is in fact many times a human engineer realising the prescribed model.

Along this insight, we first collect the various human parties involved in different aspects of a modelling language and shortly present their viewpoints. Then we identify the various artefacts a specification may provide to cater to these stakeholders and examine them with respect to the extent and quality of understanding they can provide about the language. Finally, we zoom in on these understandings and analyse the impact of the various ways they may be misaligned.

3.1 Understanding stakeholder viewpoints

Stakeholders interested in the semantics of a modelling language may have very different motivations, which may be addressed differently (Harel & Rumpe, 2004). In order to assess the quality of modelling language specifications, we will consider three distinct groups in terms of their goals and necessities.

The widest and most obvious group is *model users*, composed of end-users of the language, i.e. modellers, engineers, and any other practitioner interested in *communicating* through models (either with others or themselves). Therefore, the primary interest of model users is to efficiently, intuitively and unambiguously convey information with the help of models. However, they mostly work with informal intuitions and rely on patters to move from examples to abstractions and back.

Another group with significantly different concerns is *developers*, including implementers taking the models as specification as well as developers of simulation and analysis/verification tools. The primary interest of developers is a *precise*, often to-the-letter interpretation of a model for some kind of *execution*, as opposed to getting an overall intuition. While implementers are usually close to the modeller and can use other means of communication to clarify issues, tool developers have to rely on the specification only. Furthermore, while simulators generally aim to derive one or some valid executions of the model, analysis and verification tools in theory should be able to consider all of them (and only them).

Last but not least, the *developers of the language* or its specification, as well as *methodologists* creating guidelines for the utilisation of the language are also stakeholders with special requirements. They are distinguished from the previous stakeholders in that they

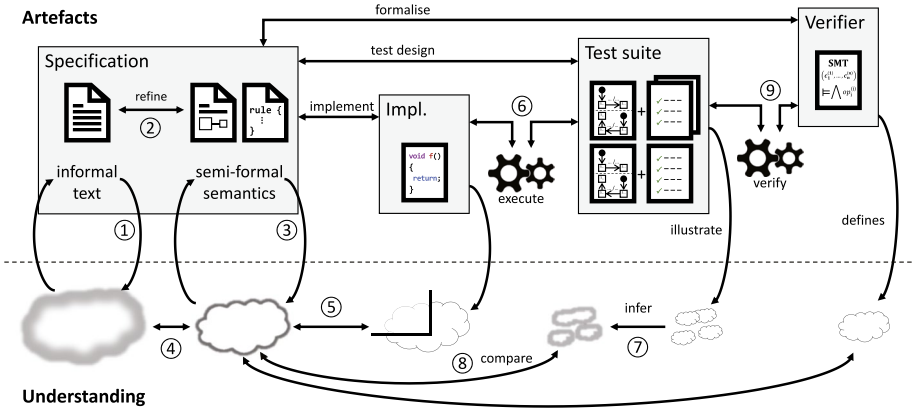


Fig. 2 Possible artefacts connected to a specification process of a modelling language, activities to check them, and understandings gathered from them

need to maintain a holistic picture of all the features and artefacts of the language in order to make it usable and ultimately a successful engineering tool. They are responsible for avoiding inconsistencies inside and between different artefacts, and for providing mappings between common problems of a domain and the constructs of the language.

3.2 Assessing views provided by specification artefacts

As we could see, while the complete and consistent description of the syntax and semantics is a primary concern of language developers, the specification of a successful language needs more than that. As a modelling language is a tool, its success will mostly depend on how well its users can employ it to solve their problems. A language not explained sufficiently to its different audiences will most likely fail.

In this section, we describe the possible artefacts produced in the specification process of a (behavioural) modelling language to facilitate this explanation. Figure 2 shows the possible artefacts, how they are related to each other, and what iterative steps could be used to improve the clear and precise understandings of these artefacts with respect to the language's semantics.

Specifications are usually given as an informal text. Language designers iteratively review and refine the text to harmonise with their understandings and to clarify vague parts and resolve inconsistencies in order to improve the quality of the specification. ① denotes this review-revise cycle. The understanding of the informal specification is the complete definition of the whole language but due to its informal nature it is unclear at certain points. This is illustrated by the blurred edges in the figure. Before submitting new versions of standards third party reviewers also review the specification in most cases.

The specification can be refined by giving a semi-formal semantics, which includes some kind of semi-formal description, e.g. using other languages or structured textual rules. ② Refining the semantics makes the specification more precise and clarifies certain questions. This process can also be used in the opposite direction to complete the informal text where needed and resolve inconsistencies. ③ The understanding based on the semi-formal semantics is more precise than the informal text but it might be limited if it does not cover the whole language (e.g. PSSM does not include time-dependant behaviours from

UML). Language designers revise the semi-formal semantics in order to match it to their intended understandings.

Once the language is defined by more artefacts, their consistency is an essential question. ④ The designers can compare their understandings of both the informal text and the semi-formal semantics. This leads to a bigger review cycle where both artefacts can be revised to decrease their misalignments.

Several other types of artefacts can help define the standard and verify its usage. An implementation developed in parallel with the standard, called a reference implementation, demonstrates the feasibility of the language, e.g. the standardisation of new C++ features usually includes collecting implementers' and users' feedback before adoption.[4] A reference implementation is precise and can clarify ambiguities, since it is defined using a programming language, which is defined quite precisely. However, it is usually limited to a subset of the language, i.e. it produces valid executions but not all possible executions. For example, Moka reference implementation of PSSM is not capable of parallel execution, which is allowed by the specification but not mandatory. ⑤ The implementation can be compared to the semi-formal semantics using reviews.

A test suite can be designed to verify whether implementations execute models conforming to the specification. For this purpose test design techniques can be used, e.g. aiming to cover the requirements imposed by the semi-formal semantics with test cases to verify that the requirements are fulfilled. ⑥ The test suite can be used to verify the reference implementation. This way we can make sure that the reference implementation produces an execution that is included among the possible valid executions defined by the test suite. A reference implementation can also be used to produce one/some of the valid executions if we have a test model without the expected valid executions.

A test suite precisely defines the semantics for the test models in it, as it lists all the possible executions. However, this only covers parts of the whole semantics. ⑦ These test cases can be considered as illustrations of the semantics on selected examples, which can be used to learn or infer the semantics of the language from these examples. The result of this learning process is a more general but still partial view of the language. ⑧ By comparing this inferred understanding with the semi-formal semantics we can discover if some parts of the semantics are not tested by the test cases or if there are inconsistencies (which will primarily concern language designers).

The specification can be extended with formal semantics using mathematical formalisms that define a precise semantics of the language, or more commonly, a subset of it. This can be compared to the specification using review to find inconsistencies and clarify parts that should be clear also in the specification. Furthermore, the formal semantics can be used by a verifier, which can derive proofs from the model considering every possible interpretation allowed by the semantics. This can be used to show that an execution trace conforms to a model, look for traces with a given property (e.g. to generate tests or find requirement violations), or even potentially generate every trace of a given model, e.g. a test model — in which case we get the expected valid executions of that test case. ⑨ This way the test suite can be verified to match the formal semantics.

[4] https://isocpp.org/std/the-life-of-an-iso-proposal

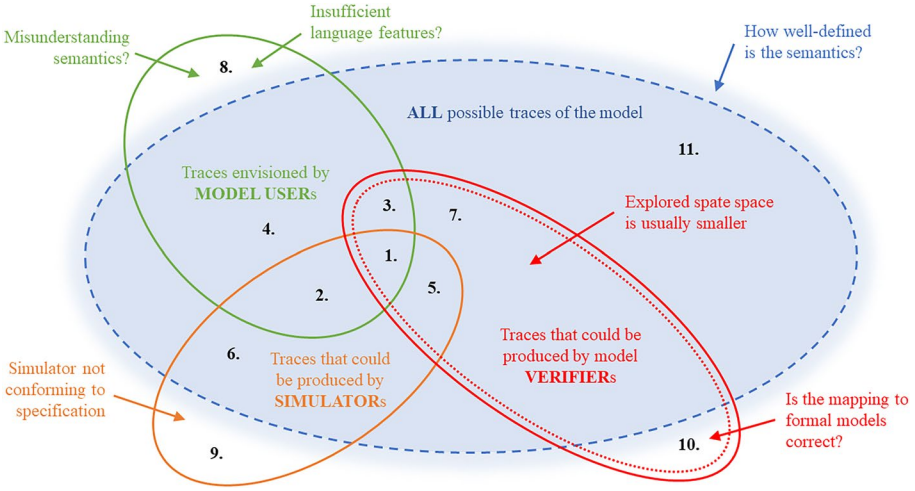


Fig. 3 Subset of traces understood by users and explored by tools

3.3 Misaligned understandings of the semantics

After discussing the several artefacts aiming to refine and cross-check our understanding of the semantics, we will now evaluate the impact of potential misalignments that can result from their quality. Exploring these misalignments is crucial in order to identify what types of issues could be present.

Figure 3 illustrates how the misalignment of the understanding of different stakeholders may affect the successful adoption of a language. Assume we have a complex behavioural model that defines a set of execution traces denoted by the dashed blue set in the Venn diagram (all possible traces). The model has been created by a modeller (a *model user* in the above grouping), whose understanding about the intended traces is denoted by the green set (model users). A *simulator* is available for the language that is able to produce the orange set of execution traces (simulators). Finally, the model is mapped into a formal language and given to a formal *verifier*, which will explore the behaviours in the red set (verifiers). In this last case, we also show with an embedded dotted set that regardless of the mapping, a verifier may be unable to explore the all behaviours of the model.

Ideally, these sets should coincide perfectly. This (very unrealistic) case would assume:

- From the user, a *perfect understanding* of all the language concepts and every detail of the semantics.
- From the simulation, a completely *controllable* simulation down to the smallest non-deterministic steps, perfectly faithful to the specification.
- From the verifier, a perfect, semantics-preserving and complete mapping from the modelling language to the formal language and a sound and complete verification algorithm that is capable of exploring all the behaviours defined by the model.

More often than not, however, the sets will not align perfectly. First of all, no matter how detailed the semantics are, there are often (either deliberately or accidentally) un- or underspecified cases where it is not possible to decide whether a given trace

is defined by the model or not. For example, most modelling language specifications refuse to go into details about memory management (the handling of variables) as it is highly platform-specific, but this makes one unable to reason about the precise effects of concurrently accessing a shared variable. Other times users or tool developers are not familiar with the details of the language well-enough, have different preconceptions which they use to fill in the blanks, or simply misunderstand things for some reason.

In the following we will discuss the causes and effects of the various misalignments, marked by numbers in Fig. 3. Any quality assessment activity should focus on recognising these misalignment and inconsistencies with respect to the various semantics.

1. The ideal case is when an execution trace is envisioned by the user, it is allowed by the specification, can be simulated with a simulator, and formal verifiers can reason about its correctness.
2. Quite frequently, verifiers will not be able to analyse a trace, either because it is theoretically very hard or impossible, or because they are not prepared for a subset of the language or miss the semantics. This is not a significant issue, because we are still in the user-expected subset and therefore confirmation by one tool is usually enough.
3. Other times, verifiers will return traces that are expected by the user, but cannot be simulated. This usually happens when the simulator cuts some corners in the semantics, and, e.g., fixes behaviours that are non-deterministic based on the specification. This case may cause confusion to the user, as simulators are considered more credible — in this case, one would check the specification to confirm that the trace is indeed allowed by the semantics.
4. Sometimes, users are aware of traces that are implied by the semantics, but no tool can handle them. Since they are left alone, there should be an easy way to get confirmation from the specification. Users will often avoid models with such behaviour because they will not take the risk of being wrong — note that the set of valid traces defined by the specification is often fuzzy.
5. A "success story" of tool development is when a verifier shows the user a trace that was not expected, and through detailed simulation users can expand their understanding about their models and the modelling language. However, in a sense, this may also be a shortcoming of the specification — one that can be compensated with good tooling.
6. A similar case is when even though we are outside the capabilities of the verifier, but using the simulator, users can discover new behaviours in their model. In a sense, this is the purpose of testing: to validate if the model indeed describes the intentions of the modeller. There may be questions about whether the trace is indeed allowed by the specification or not (case 6 or 9), but simulators are generally better in explaining their results, so users can employ techniques similar to debugging to decide on a verdict.
7. The other variant, where the verifier returns an unexpected trace without the simulator being able to reproduce it, is similar to case 3. Even worse, users may be more suspicious, because according to their understanding, the trace should not be valid according to the specification. Contrary to case 6, there is generally no possibility for "debugging". Considering whether the trace is indeed in case 7 or in case 10 outside of the behaviours allowed by the specification is a very hard task that requires expertise.
8. There may be cases when users envision traces that are not permitted by the specification. This may come from a misunderstanding of the semantics, but it may also be the consequence of the limited capabilities of the language. The former is a problem that can and should be supported by specification artefacts aimed at the user, i.e. explana-

tions and examples, or tool support to validate models and give warnings about possibly confusing parts of the model. The latter should be discovered by the language designers, who in turn should make a decision whether to support these cases or not, or provide workarounds to change the model for a better alignment. In any case, a too large gap is the responsibility of the designers and addressing it should be in their best interest if they want the language to be successful.

9. Other times, simulators will produce traces that are in fact not allowed by the specification. This is dangerous, because users will more often believe tools rather than textual descriptions of the semantics, and this way they may not be aware that other tools (e.g. code generators) may interpret the model differently. To help avoid this, language specifications may try to define test sets for the evaluation of simulators.
10. Verifiers may also produce invalid traces for a plethora of reasons, mostly due to transformations not maintaining the original semantics. Once again, they can be really confusing to users, as they need to rely on the specification to decide on a verdict. Furthermore, too many false positives can lower the usability of verifiers and erode user-trust.
11. The most dangerous part of the Venn diagram is where neither the user, not the simulator or the verifier is aware of the trace that is otherwise legal according to the specification. Users often miss corner cases (such as the specific firing sequence in the example of Section 2.4), but these will not come up in simulators either, and verifiers will produce false negatives. The problem is that other tools and people, especially implementers or code generators may very well come up with these exact traces, most of which have never been tested or verified or even considered.

A key takeaway from the above list is that misalignments can be dangerous, and it is the responsibility of the specification to help stakeholders align their understanding with the true and intended meaning of models. A language that does not help its users in this is likely to fail. In the rest of the paper, we will present a set of techniques that can be used to assess how well a specification fares in this regard, mostly by detecting consistency problems.

4 Assessing the PSSM specification

To synthesise how modelling languages' semantics can be assessed and what typical issues are present in state-of-the-art specifications, we reviewed and assessed the UML PSSM 1.0 specification (OMG, 2019). We choose PSSM as it is a recent standard applying many of the good practices listed in the previous section (e.g. test suite, reference implementation).

4.1 Steps of assessing the PSSM specification

We found no prior work on evaluating the PSSM specification. Therefore we started with common verification and validation techniques, then we outlined the possible artefacts and cross-validation options for a behavioural modelling language in the assessment workflow in Section 3.2. Figure 4 shows how this workflow is adapted to PSSM. UML defines state machines in an informal text, PSSM gives a semi-formal semantics and a test suite, Moka is the reference implementation for PSSM, Cameo is another implementation we used. Unfortunately, a PSSM-compliant verifier for state machines is not available.

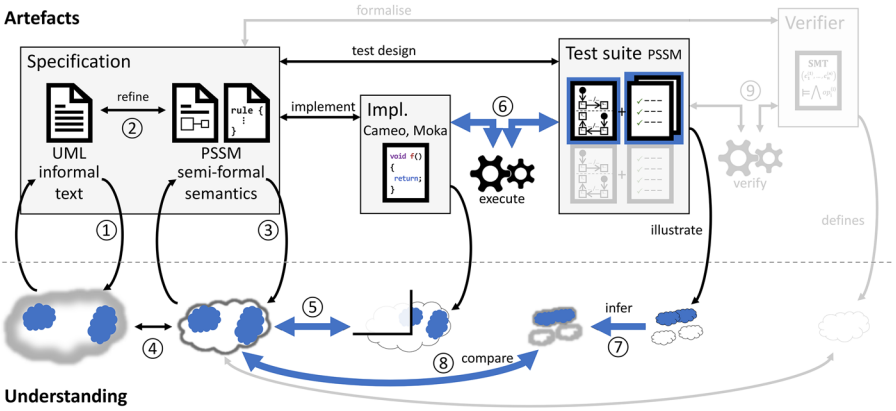


Fig. 4 Artefacts connected to UML/PSSM standard and understandings gathered from them. Highlighted blue parts symbolise questions about certain semantic concepts in the language. Blue lines connect the artefacts and their understandings that we compared during the assessment

The types of assessment activities we performed and the insights we gathered can be mapped to the workflow in Fig. 4 in the following way.

1. We had an initial understanding of the semantics defined in the UML specification from our previous experience (step ① in Fig. 4).
2. We reviewed the text of the specification (Section 4.2).

   (a) We read the standard concentrating on the parts about semantics. We collected the essential concepts that affect the execution ③. We continuously cross-checked the new information with the text of the UML specification and with our understanding of the intended semantics ④.
   (b) We reviewed and categorised the test cases. The behaviour described in the tests helped to infer additional details about the semantics ⑦.
   (c) We identified some major cross-cutting semantic concerns (e.g. scope of atomicity and interleaving in orthogonal region) that are challenging to understand even from all the available artefacts (Section 4.2.2). The highlighted blue parts symbolise these concerns in the figure. The figure stresses the main challenge in defining or assessing the semantics of modelling languages: such semantic concerns are scattered in various parts of multiple artefacts that should be consistent and unambiguous.
   (d) We cross-referenced the textual descriptions of the relevant execution classes and test cases mentioning those concerns (highlighted in blue), and tried to construct a big-picture view of the given concern supported by the evidence in steps ⑦ and ⑧ (Section 4.2.3).
   (e) To gain additional insights, we consulted the Java source code of the Moka reference implementation in step ⑤.
   (f) We collected relevant test cases, categorised them in a test coverage table to find parts of the semantics that are not tested thoroughly enough (Section 4.2.4).

(g) During the review of the tests, we found conceptual limitations and inconsistencies of PSSM's test architecture (Section 4.2.5) and the way the expected results are specified in tests (Section 4.2.6).

3. We executed the test suite available as an XMI document in a UML/SysML simulation tool in step ⑥ (Section 4.3).

   (a) We compared the trace produced by the simulator to the ones given in the specification.
   (b) We manually experimented with different stimuli in case of test cases covering the above-mentioned challenging cross-cutting concerns.

As PSSM is a complex specification, we did not aim for completeness in our review. We concentrated on the most frequently used elements and behaviour. For example, we skipped most of the test cases of the following categories: Junction, History, Redefinition. Nevertheless, we already found several issues, and our findings illustrate general challenges in specifying precisely such an expressive modelling language.

4.2 Reviewing the specification and test cases

During the review process, the first and the last author separately reviewed parts of the PSSM test suite along standard review criteria (e.g. consistency, unambiguity, completeness). Then all authors discussed the issues found. We reported the errors found to OMG through 6 issue tickets on their official portal.[5] Some concerns, e.g. exact representation of semantic concepts such as threading or RTC steps, are more significant than a simple issue, and would require a major rework of the standard. Therefore we recommend some of our findings in Section 5 primarily for language designers of modelling languages.

4.2.1 Revisiting transition firing and orthogonal regions

In the subsequent sections, most questions are concerned with the detailed execution and specific cases of the transition firing sequence or the possible concurrent execution of orthogonal regions. Therefore we shortly summarise their behaviour.

Transition firing sequence The transition firing sequence is as follows.

1. The Transition source is exited. The exit sequence consists of a call to the exit Behavior of the source vertex. This exit sequence can propagate exit calls to parent vertex as long as the least common ancestor Region of the source and the target vertex has not been reached.
2. The effect Behavior of the Transition is executed.
3. The Transition target is entered, and its entry Behavior is executed. This call can lead to a number of nested enter calls to enter parent vertices before the actual target vertex is entered until the least common ancestor is reached (OMG, 2017, 14.2.3.9.6; 2019, 8.5.3).

[5] See https://issues.omg.org/issues/spec/PSSM/1.0

Conflict and priority UML defines that two Transitions are in conflict if they both exit the same State (i.e. the intersection of the States they exit is non-empty). Only Transitions that occur in mutually orthogonal Regions may be fired simultaneously. A partial priority between transitions is defined based on their source State: a Transition originating from a substate has higher priority than a conflicting Transition from any of its containing States (OMG, 2017, 14.2.3.9.3).

Orthogonal regions In the presence of orthogonal Regions, it is possible that multiple Transitions in different Regions can be triggered by the same Event occurrence in an order which is left undefined. Each active orthogonal Region without nested Regions (i.e. a "bottom-level" Region) can fire at most one Transition for an Event occurrence. When all Regions have finished executing the Transition, the current Event occurrence is fully consumed, and the run-to-completion step is completed (OMG, 2017, 14.2.3.9.1).

These concepts seem to be well-defined. However, if we take a closer look and try to understand every possible combination and corner cases, we may have several questions. In the subsequent section, we first describe the identified general question that is relevant for all modelling languages (marked with *General*), then illustrate this question with specific issue(s) in PSSM, then detail concrete examples and evidences from the PSSM text or the test suites that help to characterise these issues, and finally summarise the insights.

4.2.2 Explicit definition of concepts in the semantic domain

The classes of the original UML specification represent those model elements that the engineers use to construct the model of the system under study. The concepts of semantics are described in the accompanying text. The novelty of fUML and PSSM was that they explicitly introduced some of these semantic classes, e.g. `Token` in activities or the deferred event pool in state machines.

General: Defining and including essential semantic concepts Turning informal semantic concepts into explicit classes makes explaining or reasoning about the semantics much easier for model users, and tool developers could implement these concepts similarly. However, during the review we found out that not all semantic concepts are explicitly included in the specifications. For example, the concept of threading is implicit in fUML activities (OMG, 2011, 8.9.1), and the run-to-completion step for state machines is not modelled explicitly in PSSM. Language designers have to make a non-trivial trade-off: specifying everything might make the specification incomprehensible, but leaving out a core concept might raise unending questions.

PSSM: What is an atomic execution step? We illustrate this situation with an example on the definition of the basic execution step. A core aspect of the semantics is the exact definition of what an atomic step of execution means. Atomicity influences what steps can overlap and when an execution can be aborted.

UML State Machines are driven by run-to-completion (RTC) steps. However, the unit of execution is not an RTC step, but the different micro-steps of entry/exit behaviours or effects. The decision on what constitutes an atomic step and when the effects can be seen, e.g. by other components, is a well-studied problem in the early history of statecharts (Pnueli & Shalev, 1991). Various statechart versions use slightly different strategies, but it is crucial to define these basic assumptions explicitly.

Fig. 5 A state machine with an aborted doActivity. Input event: `AnotherSignal`. Based on OMG (2019, Behavior 003-A)

The previously described transition firing sequence seems to define the execution steps in simple cases. How the execution steps interleave with each other is rather complex in case of concurrently firing, possibly compound transitions in orthogonal regions. Moreover, these behaviours are defined with activities composed of numerous actions. Thus besides entry/exit behaviours and transition effects interleaving with each other, they might overlap each other. How internal actions can overlap is especially interesting if, e.g., they depend on the same variable (StructuralFeature), they read or write them parallelly.

Therefore the different stakeholders could raise a number of questions regarding atomicity and interleaving.

- If there are more than one transition firing concurrently, can every step in the different transitions' firing sequences interleave with each other?
- Can entry/effect/exit behaviours of different transitions interleave?
- Are these steps intended as instantaneous? What happens if we add actions with long duration constraints to effect behaviours?
- How is the execution affected if we have an asynchronously running doActivity behaviour executing in the current state configuration?

The specification answers some of these situations with the help of the test cases. However, it provides no systematic evidence to fully answer questions about atomicity or concurrency. Some concurrency aspects are deliberately left undefined to accommodate various implementations and execution platforms, e.g. no actual parallelism or multi-threading is required in a conformant execution tool as long as it produces a subset of the allowed traces. However, the specification needs to define the set of all possible traces, i.e. if there are any restrictions at all on concurrency. It would be beneficial to clarify this in the text and illustrate its consequences with positive and negative test cases.

Example We will present these questions in more detail in the subsequent questions. Now we use the example of doActivities to show why these are important questions that can affect the execution of even basic state machines. A controversial part of the specification is the atomicity of doActivities, i.e. is there any part of a doActivity that is undoubtedly executed before its abort caused by, e.g., an event triggering an outgoing transition.

When state `S1` in Fig. 5 is entered, the entry behaviour is executed, and then the doActivity starts its execution on its own thread. After that, the state machine dispatches `AnotherSignal` from the event pool, which aborts the execution of the doActivity, runs the exit behaviour, traverses the transition and executes its effect. Test case Behavior 003-A specifies that the only possible trace is `entry::do-I::exit::effect`, where the doActivity can print to the trace, waits for an incoming `Continue`, and then the dispatching `AnotherSignal` aborts the doActivity while it is waiting for a signal. On

Table 2 RTC steps of test case Transition 019 (OMG, 2019)

Step	Event pool	Configuration	Fired transition(s)
1	[]	[] - Initial RTC step	[T1]
2	[Start, **CE(wait)**]	[wait]	[]
3	[**Start**]	[wait]	[T2(T1.1, T2.1)]
4	[Continue, CE(S1.1), **CE(S2.1)**]	[S1[S1.1, S2.1]]	[]
5	[Continue, **CE(S1.1)**]	[S1[S1.1, S2.1]]	[]
6	[**Continue**]	[S1[S1.1, S2.1]]	[T1.2, T2.2]
7	[CE(S2.2), **CE(S1.2)**]	[S1[S1.2, S2.2]	[T1.3]
8	[**CE(S2.2)**]	[S1[S2.2]]	[T2.3(T3)]

the other hand, test case Event 017-B allows an alternative execution, where the doActivity is aborted before printing to the execution trace: `entry::exit::effect`. It argues that the doActivity starts its execution asynchronously when its state is entered, thus the doActivity may or may not have the time to contribute to the trace before the state is exited.

Summary This example presented a case where the semantics inferred from different test cases are inconsistent. Without having an explicit definition of what is atomic in the state machine execution, it is hard to decide which version is the intended semantics. Edge cases and contradictions of this kind should be either clarified or explicitly marked as implementation-defined.

4.2.3 Exact representation of semantic concepts

General: Canonical representations for core semantics concepts For model users and tool developers to understand the semantics, it is helpful to have canonical representations for semantic domain concepts. For example, as we discussed previously, some semantic concepts were explicitly introduced in fUML and PSSM (e.g. activations of model elements). However, some fundamental concepts are left implicit, e.g. the RTC steps forming an execution trace. Even if traces, threading, or RTC steps are not explicitly represented in the execution model, the specification should define an unambiguous representation format for them to ease interoperability and communication.

PSSM: How to represent run-to-completion steps unambiguously? The RTC steps are described with a table in the test case definitions. The table contains an ID for the step, the status of the event pool and the state configuration before the RTC step is started, and the fired transitions.

The RTC steps table should be detailed enough to differentiate the alternative executions. The text does not clarify whether an execution trace has only a single possible sequence of RTC steps or whether different RTC steps of the same state machine can also produce the same trace. In our opinion, the RTC table does not contain all the necessary information to uniquely identify the executed behaviour (e.g. interleaving of state and transition activations).

As we discussed above, the exact behaviour of orthogonal regions poses several questions. Thus we examined the RTC steps of those. The RTC steps table defines the fired

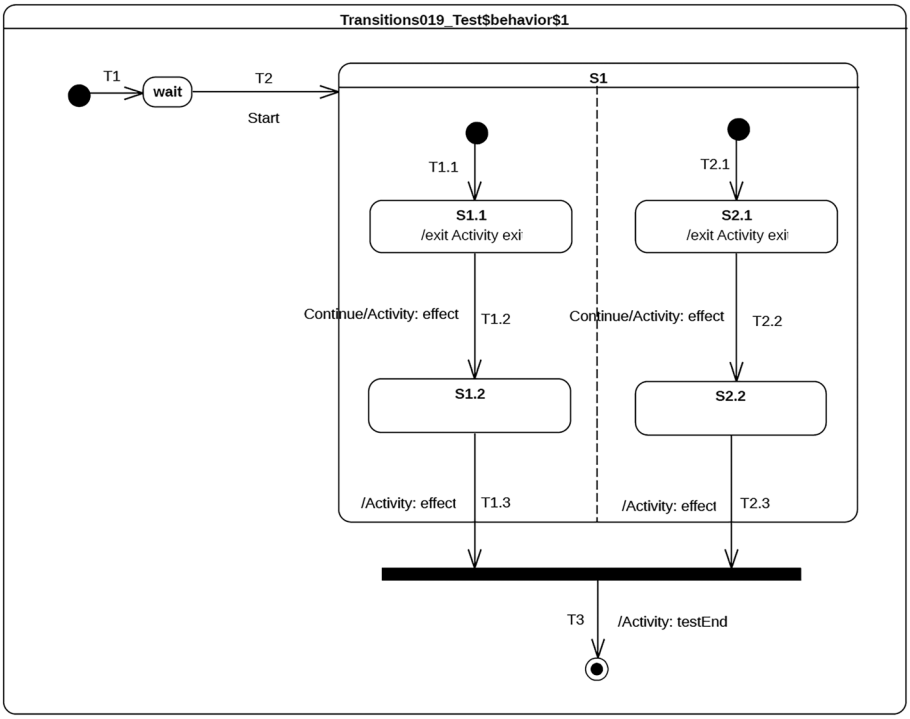


Fig. 6 A test case with compound and simultaneous transitions in orthogonal regions. Input events: `Start`, `Continue`. (OMG, 2019, Transition 019)

transitions in each step, which can include multiple transitions in orthogonal regions and compound transitions.

Question Does the order of the fired transitions in the RTC table define execution order or is it only a set of unordered transitions?

Example The RTC steps of state machine Transition 019 (Fig. 6) are described in Table 2. The first three steps are as follows.

- *Step 1-2*: Initial RTC step, then the completion event of `wait` is discarded.
- *Step 3*: The state machine dispatches a `Start` event which fires [`T2(T1.1, T2.1)`] compound transition to implicitly enter the orthogonal regions via the initial Pseudostates. Entering the states `S1.1` and `S2.1` produce completion events `CE(S1.1)` and `CE(S2.1)` (resp.).

As we can see, a lot can happen in a single RTC step, and only part of the behaviour is visible from the RTC step table. In order to decide whether the text in the fired transition(s) column represents the order in which the concurrent transitions `T1.1` and `T2.1` have been fired, we could cross-check this information with the RTC step table of the alternative traces. In both the main and alternative RTC steps table the firing is written as [`T2(T1.1, T2.1)`], but the order of produced completion events are different ([`CE(S1.1), CE(S2.1)`] and [`CE(S2.1), CE(S1.1)`]). Does this mean that the order in which the fired transitions are irrelevant (i.e. they can be in any order or fire even in a true concurrent

manner)? Or is the firing order the same as the written order, just the firing order of the transitions does not determine the order in which the completion events are produced (see the general question in the previous section about what is atomic and what can interleave)?

Similarly, in test Entering 011, the fired transitions seem to be unordered or not affecting the order of completion events since the compound transition [`T2(T1.1, T2.1)`] can produce multiple traces and CE orders. For example, in trace `T2.1(effect)::S2.1(entry)::T1.1(effect)::S1.1(entry)` the fired transition order in the RTC table is the reverse of the effect order in the trace, and the order of the state entry behaviours does not imply the order of the completion events produced. Furthermore, in an alternative trace the effect order does not imply the order of entry behaviours, i.e. it is in reverse. In Fork 001 the fired transitions are not ordered alphabetically, which would suggest that their order matters but the transition effects in the trace do not follow the same order.

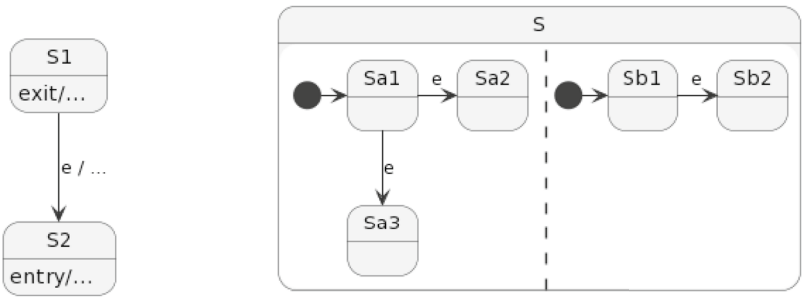
On the contrary, in a similar setting in test Entry 002-B the fired transitions are listed in different orders and the order determines the order of entry behaviours and the order of CEs in the event pool, i.e. [`T2(T1.1, T2.1)`] produces [`CE(S1.1), CE(S2.1)`] and the reversed transition order, [`T2(T2.1, T1.1)`] produces the reversed event order [`CE(S2.1), CE(S1.1)`]. The same applies to tests Terminate 001 and Terminate 002, where transition order in RTC steps determines the order of entry behaviours and completion events (in the former test). Another test from the Fork category, Fork 002 suggests that different transition firing orders imply the transition effect order in the trace, despite the counterexample in Fork 001.

As we experienced, from the partial information in the RTC step table, we could deduce that the order of the fired transitions in RTC steps in some test cases is unordered and can produce several different orders in the trace and the event pool. However, in some tests the fired transitions are ordered and this order not only defines the transition effect order in the trace but can also define the entry behaviour order and the order of the completion events in the event pool, from which it follows that these should be executed as atomic steps, even though transitions in orthogonal regions are fired concurrently. Some examples show that a single trace can be produced by several different RTC step sequences, despite at most one of them is showed in the tests.

Summary If an element of the semantic domain is not modelled explicitly, it is important to have an unambiguous representation for it in the examples and test cases. Users could use the information in the test cases to accept or reject some theories about the working of those elements. But in order to decide such questions, the test cases should not contradict each other, and the representation should contain all the necessary details or it should be explained what is left undecided in it.

4.2.4 Emergent behaviours need explanation and systematic testing

General: Undefined emergent behaviour of well-defined elements An overarching challenge in defining modelling languages' semantics is the identification and explanation of emergent behaviours. Even if some of the language elements and their semantics are clearly explained, combining these elements might result in emergent behaviours. These behaviours might be clearly defined using the composition of individual semantic rules, but in many cases, these behaviours are unexpected or even unintuitive for some users. It is a good practice to explain them with simple examples and later use systematically designed tests to check the behaviours of possible combinations.



(a) Transition effect and entry/exit behaviours. **(b)** Conflicting transitions and orthogonal regions.

Fig. 7 State machines illustrating behaviours that are well-defined separately but their interaction is complicated

PSSM refines how UML modelling elements should behave and defines test cases verifying and illustrating the behaviour of a few of these elements in each. As UML has numerous modelling elements and concepts used in state machines, the elements can have various combinations that alter each other's behaviour. Some of these combined behaviours can be easily deduced by the model users. Other combinations need clarification and explicit specification about the emergent behaviour of those modelling elements when used in combination. To verify the tools, developers need test cases designed systematically to test the intricate behaviour emerging from the combination of elements.

PSSM: Combining transition firing and orthogonal regions The steps of firing a transition, and the selection of transitions in orthogonal regions even in case of conflicts are separately well-defined (see Section 4.2.1), therefore comprehensible. However, it is considerably harder to understand the emergent behaviour of their interaction. The concurrent firing of transitions in orthogonal regions is a hard-to-understand part of PSSM, e.g. how the exit–transition effect–entry behaviour sequences can interleave with each other in orthogonal regions, and which parts should be executed in one RTC step (we mentioned this question previously in Section 4.2.2).

Question Are the possible combinations of the firing sequence of multiple transitions explained in the specification and illustrated with test cases?

Simple state machines for exit, transition effect, and entry behaviours and for conflicting transitions in orthogonal regions are shown in Fig. 7a and b. Exit–transition effect–entry sequence and orthogonal regions separately are explained in the relevant execution model classes in PSSM and some test cases verify these aspects. (Although the whole exit–transition effect–entry sequence, illustrated in Fig. 7a, is only present in quite complex test cases.[6])

[6] Test cases: Event 018, Deferred 002, History 002-A, History 002-B

Table 3 PSSM test cases where transition firing behaviours interleave with each other in orthogonal regions

	exit			transition effect		entry
Test case	exit	transition effect	entry	transition effect	entry	entry
Transition 011-D*	X					X
Transition 017*				X		
Transition 019*	X	X		X		
Transition 023						X
Event 009				X		
Event 016-B				X		
Event 019-E						X
Entering 010					X	X
Entering 011				X	X	X
Exiting 001	X					
Exiting 003	X					
Entry 002-A						X
Entry 002-B						X
Exit 002				CE		
Junction 005				X	X	
Fork 001				X	X	X
Fork 002				X	X	
Join 001	CE	CE		CE		
Join 002				X		
Join 003				CE		
Terminate 001						X
Terminate 002						X
History 001-C*			CE			CE
History 002-B*	CE	CE	CE	CE	CE	CE

Notation: X — the two behaviours interleave with each other in the test, CE — the two behaviours can precede and succeed each other depending on the order of completion events in the event pool, * — the test case has errors. *N*ote: duplicate test cases in Standalone category are excluded.

However, we were not able to find an explanation in UML and PSSM about their joint behaviour and how the steps can interleave with each other.

Example To reveal the subtle details of transition firing in orthogonal regions, we systematically reviewed all the 34 PSSM test cases which contained orthogonal regions. Table 3 lists the 24 test cases whose traces showed concurrency in transition firing steps, i.e. exit/entry behaviours and transition effects. Only selected states and transitions in the test cases are extended with activities that print to the trace therefore comparing the traces can only verify the order of these steps. Concurrency between exit/entry behaviours and transition effects is denoted with X. In certain tests (denoted with CE) alternative order of the steps is present as the transitions are triggered by completion events that are put into the event pool after their states, which are active concurrently, complete their own behaviour. However, as the completion event of a state is processed in separate RTC step, the transition firing steps in one region form an atomic step and they are not concurrent and cannot interleave with transition firing steps of other regions. In this case only the completion of

the previous states, which triggers the steps, is concurrent. Test cases marked with * have errors in the traces.[7]

Evidence There is no test case which shows the concurrency of all the 3 steps (i.e. a row with 6 X-es in the table). The 6 test cases with X marks in exit–transition effect, exit–entry, or transition effect–entry columns can partially answer the question about interleaving steps: Steps in another region (exit and effect in the example) can be executed between exit behaviour and transition effect in one region (Transition 019). Steps in another region (effect and entry in the examples) can be executed between transition effect and entry behaviour in one region (Entering 010, Entering 011, Junction 005, Fork 001, Fork 002). Since there is no X in the exit–entry column, there is no test case to show whether the whole exit–transition effect–entry sequence in one region can precede the same in another region, despite UML specifies that transitions in orthogonal regions may be fired simultaneously.

Summary Test cases should be designed systematically to illustrate and verify how such behaviours can interleave. It would be beneficial to elaborate on the emergent behaviour in all relevant artefacts, since the individual specifications might not be enough for many stakeholders.

4.2.5 Test architecture for conformance test suites

General: Observability and controllability trade-offs for the tests As we have seen, having a test suite is essential for supporting understanding (model users) and checking tool conformance (developers). The authors of PSSM spent a great amount of effort on designing a test architecture where the test suite is automated, tool-independent, and can be traced to the requirements derived from the normative text of the UML specification.

The decision to perform the testing on the model level without direct interaction with the execution engine influenced what can be logged in the test trace used to evaluate the outcome of the tests. RTC steps, the content of event pools, etc. are described as supporting information, but they are not part of the machine-readable test suite. However, as we will see, this is not the only controllability and observability challenge with the test architecture and traces.

Observability: What information should the test trace contain? Based on the test's purpose, certain state machine states and transitions are extended with entry/exit/doActivity behaviours and transition effects that trace the execution to make the behaviour of the target state machine observable and verify that the execution tool under test runs correctly. State changes and transitions are expected to happen as a consequence of stimulation signals. However, the traces do not contain the received input signals. Therefore the test suite cannot enforce the causality and the order in which the input and output signals interleave. The text also describes the intended RTC steps for each test case, but the tests have no means to verify that these were the executed RTC steps.

[7] E.g. Transition 011-D fails to list traces in both order that the concurrent entry of the initial states of two regions can produce. The order of concurrent transitions in the traces of Transition 019 determines the order of their target states' completion events in the event pool (as if transition effect and the target state producing CE is an atomic step), despite transitions in orthogonal regions may be fired simultaneously. Completion transitions in History 001-C and History 002-B incorrectly precede the initial entry of the other region.

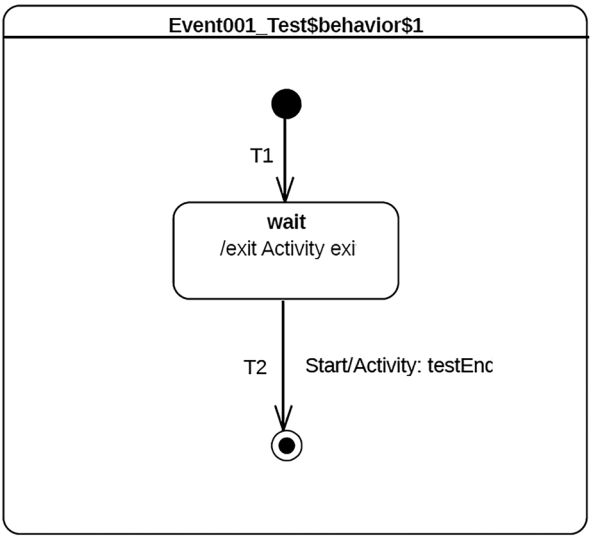


Fig. 8 A test case with a single part as expected trace. Input event: `Start`. (OMG, 2019, Event 001)

Example PSSM states that comparing only the execution traces, which are intentionally incomplete, is always sufficient to evaluate conformance.[8] However, we found that this is not necessarily true for every test case. For example, the purpose of test case Event 001 in Fig. 8 is to check that upon its creation, the state machine immediately starts its execution. The expected execution trace only contains a single part, the exit behaviour of the initial state `wait`. `Start` event triggers the outgoing transition `T2`, but this transition is not included in the trace, nor is the received `Start` event. Therefore we were not convinced that the test could actually verify that the execution was started immediately and the exit behaviour was executed when the state exited and not before that since only the exit behaviour prints to the trace.

Summary Some of these observability questions can be addressed by adding more logging calls to state and transition behaviours. This will increase the length and number of alternative traces, but with the help of automation in test design, this could be managed. However, adding the received signal to the trace is not trivial.[9] We revisit these questions in the discussions.

Controllability: Exact order and timing of test input sequence In each PSSM test case, the tester encodes the test input, i.e. a stimulation sequence, and sends it to the target. A test case enumerates the received event occurrences in an ordered list worded as "`SignalName` — received when in configuration `StateConfiguration`". This

[8] "Although the trace built during the execution is not complete, it is always sufficient to evaluate if the state machine was executed in way that conforms to the semantics specified for UML state machines" (OMG 2019, 9.3.1).

[9] Specific tools might have extra support to get such information, e.g. the Action Language Helper API in Cameo https://docs.nomagic.com/display/CST190/ALH+APIs

suggests the order of the input signals is determined, and their timing depends on the actual state configuration.[10]

Example Contrary to the quote above, the tester component disregards the state configuration and may send the stimulation signals right after the test starts.[11] For example, the tester implementation of Exiting 002 sends two `Continue` signals (parallelly). Still, the textual specification expects that the reception of the second signal should be after an `AnotherSignal` sent to the state machine (by a doActivity in the state machine itself), and the second `Continue` signal should be received only in the next state. Other testers send the signals after each other, but the specification also prescribes in which state configuration the events are received, which the test suite still does not ensure.

The tester has no means to delay sending a signal. Moreover, the tester cannot query the actual state configuration of the target state machine, thus cannot wait for a state configuration of another state machine or other events received by the state machine. Unless, of course, the target state machine sends back signals to the tester. As one of the authors of PSSM pointed out in an active OMG issue, ensuring that tester implementations send the signals in the specified order is still insufficient because there is no guarantee that the state machine will receive them in the same order as they can get reordered.[12]

Summary There is a non-trivial trade-off in the test architecture design balancing controllability/observability, the complexity of tests, and tool independence. The chosen test architecture constrains the part of the semantics that the test suite can potentially check.

4.2.6 Test design: specifying expected results in tests

General: How to derive expected results in tests efficiently? Creating a detailed, complex test suite is, again, a challenging task. Standard test design techniques could help to select relevant test inputs and scenarios. However, specifying the expected results for each test is quite resource-intensive. Without tool support or models helping to derive tests, manually specifying the tests can easily introduce issues into the test suite.

PSSM: Minor issues due to manual specification of expected result We found more than a dozen cases when the PSSM test suite had minor inaccuracies, e.g. in the RTC steps of tests. This is due to the error-prone manual specification of the test suite. In most cases, the specification references non-existing states or transitions, but some refer to existing ones that can cause confusion. For example, in test case Transition 020, the RTC steps table refers to a non-existing state `S1.1` instead of state `S1` of the model.

PSSM: Specifying alternative traces completely and consistently As PSSM permits multiple execution orders in certain cases (e.g. in orthogonal regions), alternative execution traces can also be valid. In most cases, the specification also lists these alternative traces.

[10] "test descriptions includes: … The event sequence that is received by the tested state machine. The order in which the event occurrences are enumerated is the order in which the event occurrences will be received. Each received event occurrence is related to a specific state machine configuration" (OMG 2019, 9.3.1).

[11] Except for test Deferred 006-A, in which the target state machine sends a signal to the tester that causes the tester to emit the next stimulation signal.

[12] "Tests that send multiple signals are not correct", https://issues.omg.org/issues/PSSM11-3

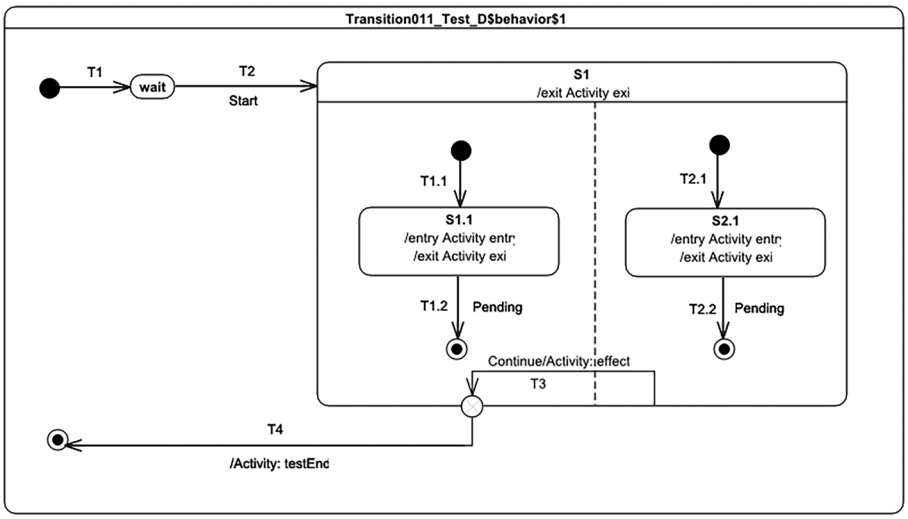


Fig. 9 A test case with missing alternative traces. Input events: `Start`, `Continue`. (OMG, 2019, Transition 011-D)

Example For example, Fig. 9 shows a test case with an orthogonal region. In state configuration `[S1[S1.1, S2.1]]` the state machine receives a `Continue` event which fires local transition `T3`, then reaching an exit point causes state `S1` to exit, and `T4` is traversed. Since the orthogonal regions are exited concurrently, the exit behaviours can print `S1.1(exit)` and `S2.1(exit)` to the trace in either order. PSSM lists the following two possible traces:

```
S1.1(entry)::S2.1(entry)::T3(effect)::S1.1(exit)::S2.1(exit)::S1(exit)
S1.1(entry)::S2.1(entry)::T3(effect)::S2.1(exit)::S1.1(exit)::S1(exit)
```

In this case, there is a more severe mistake in the specification, i.e. certain alternative execution traces are missing. Missing traces are problematic for several stakeholders. Model users might not consider all possible executions of their model, thus they might design faulty systems. Simulation and verification tool developers might build wrong tools by following the test cases, or their valid execution tools would be declared non-conformant by wrong test cases.

In Fig. 9, the substates in the orthogonal region have not only exit but also entry behaviours. Although PSSM specifies that the regions are entered concurrently

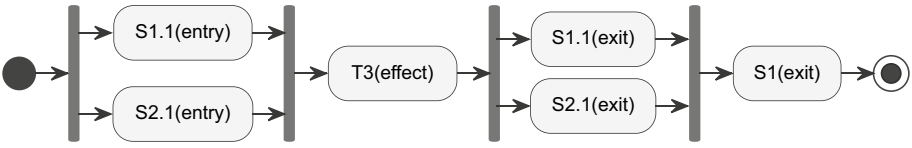


Fig. 10 The 4 valid execution traces for Fig. 9 as an activity diagram

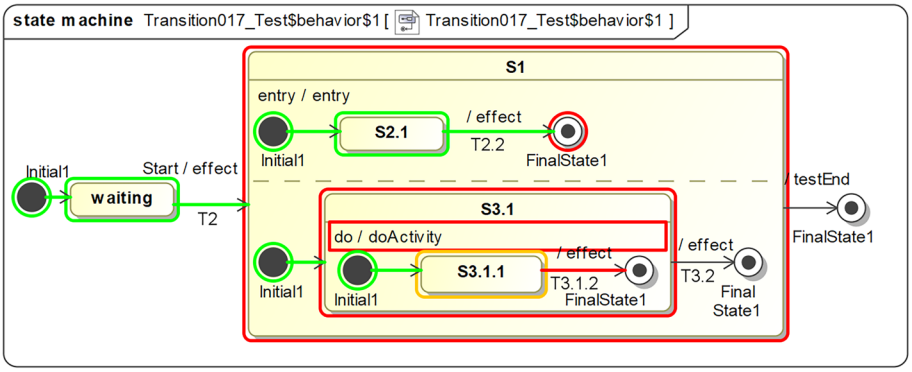


Fig. 11 Test case Transition 017 while simulated in Cameo (extended with transition labels). Visited (green), last visited (orange), and active states/transitions (red) are highlighted with colours. Input event: `Start`. Generated trace during a particular run: `T2(effect)::S1(entry)::S3.1(doActivity)::T2.2(effect)::T3.1.2(effect)`. Next part of trace: `T3.2(effect)`

(OMG, 2019, 8.5.2), the test does not list the entry behaviours in the reversed order, i.e. `S2.1(entry)::S1.1(entry)`. By contrast, test case Entering 011 correctly allows both orders of the entry behaviours.

As a compact representation for alternative execution traces, we introduce a notation similar to activity diagrams. Figure 10 shows all possible alternative traces produced by Transition 011-D in Fig. 9. The actions in the diagram print their label and a separator to the trace with the restriction that printing is atomic, i.e. each name is printed after the other and cannot overlap.

*S*ummary A more compact, readable representation for alternative traces proved to be really useful for us when reviewing complex test cases with numerous possible scenarios. Such exact representations for important semantic concepts could be essential for all kinds of modelling languages.

4.3 Executing the test cases in a simulator

From the available simulators (Section 2.5), we chose Cameo Systems Modeler 19.0 SP4 and executed the test cases in it. We also tried Moka, but it could not run state machines reliably. Cameo is an MBSE environment for SysML models, which is stated to support PSSM.[13] Importing the official test suite from the XMI format and resolving the missing dependencies was complicated, and the import displayed several warnings. Therefore we had to send the stimulation signals manually to the target state machine and inspect the execution trace in the console log instead of executing the test suite automatically. Consequently, we refrain from evaluating the conformance of the tool.

The high-level summary of our experiences is the following.

- Basic tests from, e.g., Behavior, Transition, Event categories ran successfully, generated valid traces, and showed correct behaviour during the step-by-step execution using the debugging tools of Cameo.

[13] https://docs.nomagic.com/display/CSM190/19.0+LTR+Version+News

- A few advanced modelling elements are not supported in execution: e.g. fork/join pseudostates, evaluating guards with properties (*Choice 001*).
- Some elements or not recommended constructs are wrongly executed:
  - local/external transitions (*Transition 011-A–D*);
  - when the execution of a region is to be ignored because it misses the initial state (*Entering 004*);
  - the static analysis phase of junction pseudostates, which should precede the RTC step (*Choice 005*);
  - event deferred by a state in a region which blocks the other orthogonal regions accepting it (*Deferred 004-A*).

A more important observation regarding the semantics is that since Cameo is a simulation tool and not a model checker, it cannot be used to produce all possible execution traces. However, it is not explicitly defined how the simulator handles non-deterministic choices in the semantics (or even if every choice point is recognised by the simulator). Without knowing such implementation details, users of a simulator might get the wrong impression that their model is deterministic or there is no concurrency in it.

For example, in case of the test case *Event 015* having two conflicting (completion) transitions, Cameo seems to always traverse the same transition first. On the other hand, Cameo can produce different execution traces non-deterministically in other tests, e.g. in orthogonal regions (Fig. 11).[14]

The PSSM specification permits that conformant tools produce only a subset of the possible traces. Concurrency or time are intentionally not constrained by the fUML execution model (OMG, 2021, 2.3). However, handling concurrency is not an explicit semantic variation point, therefore there is no standardised way to report on how these non-determinism resulting from concurrency is handled internally in the simulator. Standardised templates of conformance statements for such tool-specific choices would be beneficial.

5 Discussion

Based on the collected specification aspects, existing good practices, and the experiences when assessing the PSSM specification, in this section we make recommendations to improve the specifications of future modelling languages.

- The section starts with recommendations for the language (Section 5.1):
  - *Representation of semantic domain*: the core semantic concepts of the language should be explicitly represented as language elements and not just in the informal text.
  - *Abstraction level of execution steps*: an essential semantic decision is what is observable from the execution of the model.
- Then it continues with recommendations for test suites (Section 5.2):

[14] The following 2 of 8 possible traces were observed while simulating Transition 017 in Cameo: `T2(effect)::S1(entry)::S3.1(doActivity)::T2.2(effect)::T3.1.2(effect)::T3.2(effect)`, `T2(effect)::S1(entry)::T3.1.2(effect)::S3.1(doActivity)::T2.2(effect)::T3.2(effect)`.

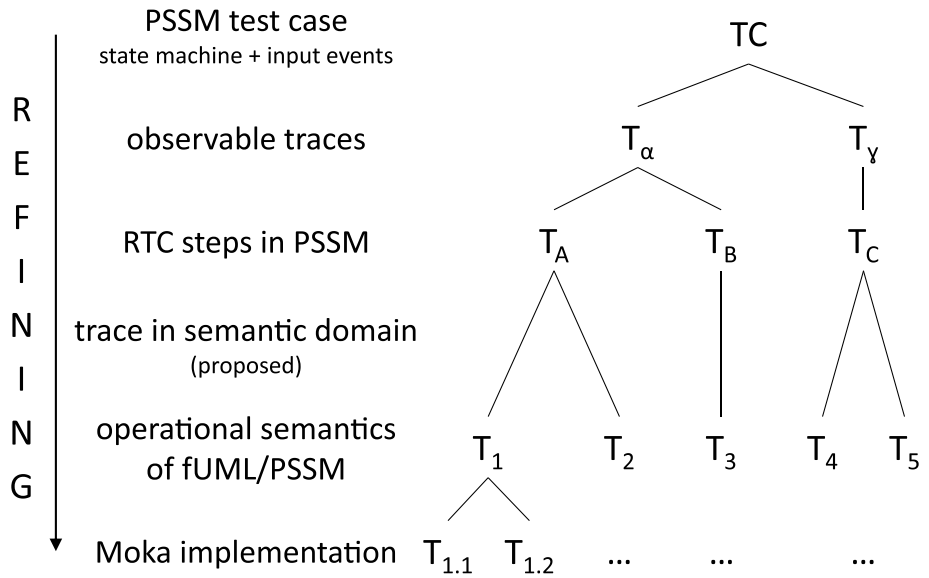


Fig. 12 The observable traces (T) produced by a test case (TC) in different abstraction levels from high-level (top) to implementation-level (bottom)

- *Derived test oracles*: automatically deriving all valid execution traces for a model would reduce inconsistencies in the test suite.
- *Rigorous check of execution*: checking only a high-level, partial trace could hide several issues in the implementations.
- *Representation of alternative execution traces*: a graph-based representation could be useful to capture many traces compactly.
- *Creating tests using more diverse test design techniques*: after fulfilling requirement coverage, the test suite could be extended with other test design techniques (e.g. combinatorial testing of model elements).
- *Separate tests for conformance testing and for readers*: start with simple tests explaining only the semantics of basic elements, then gradually include further elements or specific cases useful for conformance checking.

- Finally, recommendations for tool support are given (Section 5.3).

5.1 Recommendations for language specifications

5.1.1 Representation of semantic domain

The precision and unambiguity of arguing about the semantics of a language are heavily influenced by how its semantic concepts are defined. The fUML and PSSM standards make the reasoning easier and more precise by explicitly defining some concepts of the semantic domain with classes (e.g. event pool). Concepts of the semantic domain are defined in the same metamodel as the elements of the language. Another possibility would be to give the language translational semantics to another language.

During reviewing PSSM we found that some concepts of the semantic domain missed their explicit definition, e.g. run-to-completion step of state machines. Consequently, several questions (e.g. concurrency) are hard to answer and the PSSM test suite contains contradictions since we do not have the concepts to specify the "micro-steps" within an RTC step.

5.1.2 Abstraction level of execution steps

Defining the abstraction level of execution is a crucial decision, because it limits what can be observed from the semantics by the various users and tools. Figure 12 shows an overview of the hierarchy of traces with different levels of details. If a state machine receives an input, it could produce traces on the following abstraction levels during its execution.

1. Selected states and transitions in the state machine are extended with activities that print to the trace. These traces (T_α and T_γ) are on the first trace level. They are the only behaviours that can be observed from outside if the state machine is treated as a black box.
2. The next level contains the RTC step representation used in PSSM test suite. The specification uses them only for illustration and they are not used during verification as there is no means to examine the current state of another state machine. However, several such traces ($T_A, \ldots, T_C$) can produce the same observable trace (T_α).
3. The next level of traces ($T_1, \ldots, T_5$) present the operational semantics as defined in PSSM with the methods of the execution engine.
4. The most detailed traces ($T_{1.1}, T_{1.2}, \ldots$) are the traces produced by logging the steps, e.g. the method calls, of a concrete implementation, the Moka reference implementation in this case.

As we see in Section 4.2.3, different RTC step sequences of a state machine can produce the same trace despite at most one RTC sequence is listed for each test. Furthermore, the RTC steps representation used in PSSM is not detailed enough to unambiguously define the next step, e.g. how the concurrent steps produce completion events or what observable traces the state machine produce. Thus, the figure shows an idealistic view, and the RTC steps in PSSM are not detailed enough for the verification purposes of tool developers and neither sufficient for model users to conceive the observable traces based on them. On the other hand, the operational semantics level with the method call traces is overly detailed and it is closer to the abstraction level of Moka's traces than the concepts of state machine execution. Using that to represent the execution steps of a state machine restricts implementations to use the same class hierarchy and methods, and even the body of the methods in fUML. Although a concrete implementation is detailed enough to catch all the details, PSSM and fUML reference implementations are not complete, e.g. for concurrent behaviours as they are single-threaded.

We recommend having a trace level described using the concepts of the semantic domain. This level is in the middle between the RTC steps and the current operational semantics in PSSM and is detailed enough to describe the execution steps unambiguously but does not restrict the implementations since only concepts of the semantic domain are used.

Currently to check tool conformance only the observable traces can be used but they proved to be ambiguous for some testing purposes (e.g. a trace with a single part in Fig. 8). This could be improved if the observable trace also includes the stimulation sequences sent to the test target (Section 5.2.2). To treat the state machine execution more like a white box, we need to access the internals of the state machine execution, e.g. event pool, activated elements, and boundaries of RTC steps. There is no means in the specification to observe these while the state machine is executed, e.g. in a simulator. In order to facilitate debugging and verification of the operational semantics, future specifications may include concepts to observe runtime information in the metamodel, e.g. a trace model to observe state changes. A similar extension to fUML was proposed by Mayerhofer et al. (2012).

5.2 Recommendations for test suites

5.2.1 Derived test oracles

Most of the issues we found (missing alternative executions, inconsistencies, and typos) could be avoided if the expected output and the execution steps are automatically derived from the specified test model and its inputs. This way the generated output could be used as test oracle to decide whether a test passes or fails. PSSM has a proof-of-concept implementation, Eclipse Papyrus/Moka,[15] which supports execution and debugging of UML models. An execution framework is typically able to realise a certain execution trace but not all variants of them. A single execution can generate the RTC steps of that specific execution. Generating all possible executions is an open question. For this purpose model checkers (Horváth et al., 2020) could be used instead of manual enumeration.

5.2.2 Rigorous check of execution

Only partially verifying the execution trace helps the tests to be targeted. However, we suggest that the order in which the stimulation signals are received should also be recorded during the execution. Verifying the order of both the stimulations and the trace is needed to check the causal relation. Moreover, the execution tools should support a way to monitor the RTC steps and compare them to the expected ones.

5.2.3 Representation of alternative execution traces

Some test cases can produce alternative execution traces, which are represented as lists. If there are more than a few or even more (e.g. 85 traces for test Fork 001), it is hard to understand the list of valid traces. Partially ordered sets can be used to specify which executions should precede which, and leave the order of the others undefined. A graphical representation would help language designers to specify the traces, and would also help model users and developers to understand them more easily. We suggest a notation similar to activity diagrams to represent alternative execution traces (Fig. 10). This notation has the ability to represent conditional trace segments which is needed in certain tests where some parts can be omitted, e.g. due to abort.

[15] https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

5.2.4 Creating tests using more diverse test design techniques

As general-purpose modelling languages cover a wide range of use cases, they might comprise numerous modelling elements and semantic rules. It is a good practice that PSSM describes at least some details of the test design process, i.e. the test design technique used. The test suite was designed using requirement-based black-box test design technique, i.e. the tests aimed to cover the requirements extracted from the normative text of the UML specification and a traceability matrix shows which requirement is covered by which test.

While this coverage might be sufficient to cover the text of the specification, when using the modelling elements jointly a substantial number of combinations are possible, and the language designers need to systematically cover the significant ones. To this end, combinatorial test design techniques can be used (Kuhn et al., 2013). In addition to the requirement-based categorisation, test cases can be categorised into equivalence partitions, e.g. tables showing which modelling elements and behaviours are covered in each test (e.g. which tests contain simple, composite, or orthogonal states). This could be useful for both the language designers during test design to increase the test coverage on model elements and also for model users and developers looking for illustrative test cases. As an example, we categorised the test cases which combine transition firing steps and orthogonal regions (Table 3).

If the reference implementation is available, its code can be used to measure test coverage and design additional tests to cover missing functionality.

5.2.5 Separate tests for conformance testing and for readers

Model users most likely use the specification to learn the behaviour by looking at the relevant examples in the test suite rather than reading the normative text of UML and PSSM thoroughly. To make this learning easier, test cases should be ordered from simplest to more complex ones and from the ones using basic modelling elements to more advanced ones. This would help the readers to understand the specification gradually, and tool developers would be able to provide partial implementations without the need to support all modelling elements from the beginning. To help model users to comprehend the language, examples for certain corner cases should be included. Additionally, tables about the modelling elements used in each test could be provided as an index for navigating in the document.

More rigorous testing increases the number and complexity of the test cases. The test cases should be separated to both allow thorough testing of the tools and keep the document's readability. One group could contain the tests selected for readers to easily understand the basic and advanced modelling elements then the emergent behaviour of a few combinations. The other could extend this to reach higher coverage when testing the conformance without the constraint of readability, especially testing the combinations of elements.

5.3 Recommendations for tooling

Evaluating current practices UML made a major step forward with fUML and PSSM. Extending the specification with a reference implementation and a test suite is crucial for both helping to understand the semantics and assess the correctness of other tools. Starting to create these tools in the early phases of the specification development can greatly improve the specification's quality.

Having the full test suite available to download and execute is convenient. However, importing UML models from XMI is notoriously problematic, therefore some official guidelines would be helpful (e.g. which other XMIs are needed, which tools were able to import it).

Future tools to ease cross-checking the semantics of artefacts If we revisit Fig. 4, we can see that there are still many ways to support the development and assessment of a modelling language. We list a few recommendations that could be worth exploring.

- Comparing the semantics implemented in a simulator to the specification is limited by the fact that the simulator usually returns only one trace for a given input. Simulators could be extended to select outcomes of non-deterministic choices during execution, or at least report such choices.
- A specific verifier tool could be used to check whether a given trace is a valid execution of a model.
- An advanced verifier could be developed to enumerate all or most of the possible valid traces for a given model.

5.4 Threats to validity

Internal validity As the primary goal of this paper was not to assess the PSSM standard per se, but to devise a methodology to assess it along with other similar specifications, we allowed ourselves to diverge from standard empirical protocols.

Most importantly, the workflow was created in parallel with the assessment, iteratively refining both as we discovered newer and newer aspects of the problem. This approach may threaten internal validity of both the workflow definition process and the assessment of PSSM:

1. The workflow may be too specific to PSSM (which may seem like an external validity issue, but in fact, it is related to the validity of our method of devising the workflow). We may have failed to discover relevant aspects that were either hidden by the flaws of PSSM or were not present in it at all.
2. The assessment of PSSM influenced how we assessed PSSM, which is not recommended in an empirical evaluation process. However, as stated above, this is not the primary contribution and was a conscious decision.

A more abstract threat is the fact that this paper is also based on the (subjective) understanding of the authors about the artefacts illustrated in Fig. 2. One of our key takeaway messages was that this source of uncertainty has to be acknowledged during the specification process, and one way to address it is the presence of multiple artefacts that can be cross-validated.

External validity Although our goal was to assess modelling languages in general, our experience is mostly based on discrete, software-based systems. We expect that our findings on semantics, representation of traces, and different types of inconsistencies are also valid for other types of modelling language families (e.g. continuous models like Matlab/Simulink, Modelica).

Our experience with stakeholder roles, tools, and model-based methods originates from specific domains (mostly automotive, railway, and aerospace). Other domains using model-based techniques might rely on different practices or have different requirements.

To better assess the validity of our results, it would be beneficial to further explore the discussed issues in the context of other subdomains and language specifications. With the accumulation of such studies, we hope the way towards more precise methodological and empirical studies will eventually open.

6 Related works

This section collects the related work on 1) common practices in modelling language specifications, 2) specification and assessment of modelling languages, 3) experiences in defining semantics for UML, 4) works on using fUML and PSSM, and 5) conformance testing in other domains.

Common practices in modelling language specification At the conception of new language specifications, there are generally two ways to approach: either *1)* start with a use case and syntax-oriented, intuitive, informal description and later define the semantics more precisely; or *2)* start from a mathematically precise formalism and add language features as syntax sugar. Examples for the former are UML (OMG, 2017) and the Form-L specification language (Nguyen, 2019), where the core of the language in both cases was built from pre-existing approaches and practical user requirements. In case of UML, the extension with (semi-)formal semantics came a lot later with the fUML (OMG, 2011) and later the PSSM (OMG, 2019) specifications. Examples for the latter approach include the Common Requirement Modelling Language (CRML, Bouskela et al., 2022), which in a sense is a redesign of the Form-L language bottom-up from mathematical logic and Modelica, as well as the Property Specification Language (PSL, IEEE, 2012) which is an extension of temporal logics to provide a richer syntax for formal property specification.

One of the latest large-scale general-purpose language specification efforts currently in progress is the standardisation of SysMLv2. The SysMLv2 Submission Team includes a lot of experts with many different backgrounds, including many of the people previously working on fUML, PSSM and SysMLv1, as well as representatives of many different stakeholders (users and experts from several domains, tool vendors, formal method

experts). Building on the vast amount of experience these people bring, the standardisation process aims to ship a very diverse set of artefacts already in the first release. This includes textual and formal descriptions of the core of the language, KerML, as well as explanatory and reference manual-style textual descriptions of language elements, but also a standardised API, a reference implementation including the first prototype of a simulator, guidelines for industrial tool developers, and a rich set of examples and test cases. This should serve not only the future adopters of the standard, but also the language designers in validating their concepts from the very start. Friedenthal (2018) summarises the goal of the SysMLv2 submission process in more detail.

Specifying and assessing modelling languages Bork et al. (2020) surveys techniques used in the specification of 11 well-known visual modelling language specifications. The paper focuses more on the methods to specify the syntax and details structural modelling concepts. They found that most specification use the combination of metamodels and natural language text descriptions.

Czech et al. (2020) collected good practices for domain-specific modelling. They classified the identified good practices with respect to the language development lifecycle phases: planning, analysis, design, implementation, test, and maintenance. However, they found only a few advises for testing (Ratiu et al., 2018), and reported ones were more focused on testing the language implementation and associated tooling (editors, code generators, etc.).

Bork and Roelens (2021) discuss the definition of visual modelling language notations. They recommend a technique to evaluate and improve the *semantic transparency* of a notation. Semantic transparency is "the extent to which a novice reader can infer the meaning of a symbol from its appearance alone" (Moody, 2009). A semantically transparent notation is intuitive and it helps understanding the model by shifting some of the cognitive tasks to perceptual tasks. In our experience, this intuitiveness and readability are also important for the semantics of the language.

As we reported in the discussion section, specifying the execution trace format of a language is critical for the semantics, as it impacts understandability and conformance testing. Hojaji et al. (2019) conducted a systematic mapping study of 64 primary studies on model execution tracing. They reported that the surveyed approaches mainly use translational and operational approaches for semantic definitions. Collection of traces was recommended for various reasons (debugging, testing or model checking). The authors discuss the need for a common execution trace format. Zschaler et al. (2023) present a framework for executable domain-specific modelling languages (xDSMLs) to integrate execution semantics with concurrency models of atomic and parallel steps. The framework allows exploring and debugging possible execution traces along concurrency strategies. They argue that exploring the concurrency model is useful for language designers to specify the right semantics and for modellers to ensure their models capture the intended concurrent behaviour.

There are several user studies and controlled experiments (Budgen et al., 2011), where some aspects of UML are evaluated with the help of human participants. However, those studies mostly investigated the effect of the graphical layout (Purchase et al., 2000) or model size and complexity (Störrle, 2014), and did not concentrated on the semantic details. Wiesmayr et al. (2021) performed a user study to evaluate the usefulness of visual programming IDEs.

Semantics for UML There is a long history of works about the semantics of modelling languages and specifically for UML. Harel and Rumpe (2004) explain what does and what does not semantics mean for models, and argue that the explicit definition of the semantic domain is crucial. They also mention that different kinds of representations are needed for the various audiences (e.g. users or developers).

Seidewitz (2003) gives an overview about what models mean in the context of UML: how models are specified with metamodels, and how is the model interpreted (i.e. the mapping of the model to the system under study). The concepts presented in the paper appeared later in the overview sections of the fUML specification (OMG, 2011). Broy and Cengarle (2011) report experiences in defining formal semantics for UML, and point why a solid semantic foundation is needed for a formal modelling language that is to be used for code generation or simulation.

There are many approaches that extend UML with some kind of formal semantics to support execution and analysis of UML models. Ciccozzi et al. (2019) collect 63 papers and 19 tools on executing UML models. They found out that state machines are frequently used, but covering all UML concepts is still an open challenge. Most of the approaches use a translational approach, and there are only a handful of model interpreter approaches.

Verification of UML model is a large field of research. Gabmeyer et al. (2019) presents a survey on formal verification of software models. However, most of the techniques are for the model and not for the language level, i.e. methods to verify user models and not the specification of the language itself.

Finally, specifically the semantics of state machines were studied extensively in the literature. Relevant surveys (Crane & Dingel, 2005; Lund et al., 2007; André et al., 2023) present more than 40 approaches that recommend various types of formal semantics to UML State Machines. Moreover, many variants of state machines emergent targeted towards real-time (Posse & Dingel, 2016) or component-based systems (Graics et al., 2020).

fUML and PSSM There is a limited amount of work on fUML, and there are very few papers mentioning PSSM. Seidewitz (2014) gives an overview about fUML and its recommended textual action language, Alf. Seidewitz and Tatibouet (2015) discuss how design in fUML and execution can be supported by tools.

Craciun et al. (2013) introduces a rewrite-based executable semantic framework for fUML that will enable model-based testing of fUML models. Romero et al. (2014) verified the base UML (bUML) subset of fUML using theorem proving and detected inconsistencies and incomplete parts in the specification. Mayerhofer et al. (2012) introduced an extension of the fUML virtual machine with a dedicated trace model, and a command API to support debugging. Abdelhalim et al. (2013) implement a formal verifier framework for fUML.

Pham et al. (2017) implemented a state machine code generator, and used the PSSM test suite to test its conformance. We are not aware of any other paper investigating the PSSM specification or analysing its semantics.

Conformance testing Conformance testing has well-defined methodologies originated from the telecommunication domain (ISO/IEC, 1994). These methodologies also recommend a detailed design review for validating test specifications. The European Telecommunications Standards Institute developed the Test Description Language (TDL) (Makedonski et al., 2019) to support defining test objectives.

Conformity of programming languages is also tested, e.g. Java Specification Requests contain Technology Compatibility Kits to assess conformance of the implementations (Søndergaard et al., 2017). In the Java 8 Language Specification (Gosling et al., 2014) they present several example Test programs with expected outputs in each chapter after larger language feature specifications.

Großer et al. (2022) studied different types of traceability relations and dependencies between requirement documents to support requirement reviews and conformance analysis in large space-engineering projects. They focused on requirement reuse, e.g. tailoring requirements from a standard to specific projects, and analysed requirement graphs automatically to reveal document inconsistencies and missing tailorings in a case study from the space industry.

7 Conclusion

Succeeding in building complex, safe, and reliable systems is hard to achieve without using models and modelling. Building accurate, helpful, collaborative models is hard to achieve without having precise, understandable modelling language specifications. In this paper, we investigated why specifying precise and comprehensible semantics is a crucial but highly challenging goal due to the possibly conflicting understandings of the various stakeholders using the modelling language.

We identified the viewpoints of model users, developers, and language designers concerning the specification of the semantics of a modelling language. We collected the various artefacts that support the definition of the language's semantics. We analysed how the model execution traces envisioned by model users could differ from the traces produced by simulators or considered by model verifiers and why each type of difference is relevant but needs different verification and validation techniques to identify or resolve. Furthermore, we summarised how these differences and inconsistencies could be identified while assessing the semantics.

To investigate the typical challenges and issues in the specification of semantics in more detail, we assessed a state-of-the-art, general-purpose modelling language specification, the Precise Semantics of UML State Machines (PSSM). We categorised and presented an overview of the issues found while reviewing the PSSM specification and executing its test suite. Issues ranged from minor errors in test descriptions to unanswered questions about fundamental semantic concepts like atomicity or interleaving of executions. Based on these experiences, we refined the general challenges and assessment activities.

We recommended improving the specification of future modelling languages using explicit or more exact representations of the semantic domain, more systematic approaches for designing the test suite, and tooling support, e.g. exploring all possible traces for the expected behaviour.

As future work in an ongoing collaboration on developing pragmatic formal verification methods for systems engineering models, we plan to contribute to the semantic definition of the upcoming SysMLv2 language, and to partially automate generating and validating such test suites for modelling languages using novel graph generation (Semeráth et al., 2021).

Funding Open access funding provided by Budapest University of Technology and Economics. This work was partially supported by the National Research, Development and Innovation Fund of Hungary, financed

under the 2019-2.1.1-EUREKA-2019-00001 funding scheme, and by the European Union's Horizon 2020 program under the Marie Sklodowska-Curie grant agreement No. 823788.

Data availability All data generated or analysed during this study are included in this published article. The specifications from the Object Management Group are available from its website: https://www.omg.org/spec.

Declarations

Competing interests The authors declare no competing interests.

References

Abdelhalim, I., Schneider, S. A., & Treharne, H. (2013). An integrated framework for checking the behaviour of fuml models using CSP. *International Journal on Software Tools for Technology Transfer, 15*(4), 375–396. https://doi.org/10.1007/s10009-012-0243-0

André, E., Liu, S., Liu, Y., et al. (2023). Formalizing UML state machines for automated verification - A survey. *ACM Computing Surveys*. https://doi.org/10.1145/3579821

Bork, D., & Roelens, B. (2021). A technique for evaluating and improving the semantic transparency of modeling language notations. *Software and Systems Modeling, 20*(4), 939–963. https://doi.org/10.1007/s10270-021-00895-w

Bork, D., Karagiannis, D., & Pittl, B. (2020). A survey of modeling language specification techniques. *Information Systems, 87*. https://doi.org/10.1016/j.is.2019.101425.

Bouskela, D., Falcone, A., Garro, A., et al. (2022). Formal requirements modeling for cyber-physical systems engineering: an integrated solution based on FORM-L and modelica. *Requirements Engineering, 27*(1), 1–30. https://doi.org/10.1007/s00766-021-00359-z

Brambilla, M., Cabot, J., & Wimmer, M. (2012). Model-Driven Software Engineering in Practice. Morgan & Claypool.

Broy, M., & Cengarle, M. V. (2011). UML formal semantics: lessons learned. *Software and Systems Modeling, 10*(4), 441–446. https://doi.org/10.1007/s10270-011-0207-y

Budgen, D., Burn, A. J., Brereton, O. P., et al. (2011). Empirical evidence about the UML: a systematic literature review. *Software: Practice and Experience, 41*(4), 363–392. https://doi.org/10.1002/spe.1009

Ciccozzi, F., Malavolta, I., & Selic, B. (2019). Execution of UML models: a systematic review of research and practice. *Software and Systems Modeling, 18*(3), 2313–2360. https://doi.org/10.1007/s10270-018-0675-4

Cook, S. (2012). Looking back at UML. *Software and Systems Modeling, 11*(4), 471–480. https://doi.org/10.1007/s10270-012-0256-x

Craciun, F., Motogna, S., & Lazar, I. (2013). Towards better testing of fUML models. In: *ICST. IEEE Computer Society*, pp. 485–486. https://doi.org/10.1109/ICST.2013.67

Crane, M. L., & Dingel, J. (2005). On the semantics of UML state machines: Categorization and comparision. In: Technical Report 2005-501, School of Computing, Queen's University, Canada.

Czech, G., Moser, M., & Pichler, J. (2020). A systematic mapping study on best practices for domain-specific modeling. *Software Quality Journal, 28*(2), 663–692. https://doi.org/10.1007/s11219-019-09466-1

Elekes, M., & Micskei, Z. (2021). Towards testing the UML PSSM test suite. In: 10th Latin-American Symposium on Dependable Computing, LADC 2021, Florianópolis, Brazil, November 22-26, 2021. IEEE, pp 1–4, https://doi.org/10.1109/LADC53747.2021.9672570

Friedenthal, S. (2018). Requirements for the next generation Systems Modeling Language (SysML® v2). *Insight, 21*(1), 21–25. https://doi.org/10.1002/inst.12186

Gabmeyer, S., Kaufmann, P., Seidl, M., et al. (2019). A feature-based classification of formal verification techniques for software models. *Software and Systems Modeling, 18*(1), 473–498. https://doi.org/10.1007/s10270-017-0591-z

Gosling, J., Joy, B., Steele, G. L., et al. (2014). The Java Language Specification, Java SE 8 Edition, 1st edn. Addison-Wesley Professional.

Graics, B., Molnár, V., Vörös, A., et al. (2020). Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling, 19*(6), 1483–1517. https://doi.org/10.1007/s10270-020-00806-5

Großer, K., Riediger, V., & Jürjens, J. (2022). Requirements document relations. *Software and Systems Modeling, 21*(6), 1–37. https://doi.org/10.1007/s10270-021-00958-y

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming, 8*(3), 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

Harel, D., & Rumpe, B. (2004). Meaningful modeling: What's the semantics of semantics? *Computer, 37*(10), 64–72. https://doi.org/10.1109/MC.2004.172

Hojaji, F., Mayerhofer, T., Zamani, B., et al. (2019). Model execution tracing: a systematic mapping study. *Software and Systems Modeling, 18*(6), 3461–3485. https://doi.org/10.1007/s10270-019-00724-1

Horváth, B., Graics, B., Hajdu, Á., et al. (2020). Model checking as a service: towards pragmatic hidden formal methods. In: *MODELS Companion. ACM*, pp. 37:1–37:5, https://doi.org/10.1145/3417990.3421407

IEEE. (2012). Standard for Property Specification Language (PSL). https://doi.org/10.1109/IEEESTD.2012.6228486. IEC 62531:2012(E) (IEEE Std 1850-2010).

ISO/IEC. (1994). Conformance testing methodology and framework. ISO/IEC 9646.

ISO/IEC/IEEE. (2018). Systems and software engineering – Life cycle processes – Requirements engineering. 29148-2018.

Kuhn, D. R., Kacker, R. N., & Lei, Y. (2013). Introduction to Combinatorial Testing. Chapman and Hall/CRC.

Lima, L., Tavares, A., & Nogueira, S. C. (2020). A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. *Science of Computer Programming, 197*(102), 497. https://doi.org/10.1016/j.scico.2020.102497

Lund, M. S., Refsdal, A., & Stølen, K. (2007). Semantics of UML models for dynamic behavior - A survey of different approaches. In: Model-Based Engineering of Embedded Real-Time Systems, LNCS, vol 6100. Springer, pp 77–103. https://doi.org/10.1007/978-3-642-16277-0_4

Makedonski, P., Adamis, G., Käärik, M., et al. (2019). Test descriptions with ETSI TDL. *Software Quality Journal, 27*(2), 885–917. https://doi.org/10.1007/s11219-018-9423-9

Mayerhofer, T., Langer, P., & Kappel, G. (2012). A runtime model for fUML. In: 7th Workshop on Models@run.time, Innsbruck, Austria. ACM, pp 53–58. https://doi.org/10.1145/2422518.2422527

Micskei, Z., & Waeselynck, H. (2011). The many meanings of uml 2 sequence diagrams: a survey. *Software and Systems Modeling, 10*, 489–514. https://doi.org/10.1007/s10270-010-0157-9

Micskei, Z., Konnerth, R., Horváth, B., et al. (2014). On open source tools for behavioral modeling and analysis with fUML and Alf. In: OSS4MDE@MoDELS, vol 1290. CEUR-WS.org, pp 31–41.

Moody, D. L. (2009). The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering, 35*(6), 756–779. https://doi.org/10.1109/TSE.2009.67

Nguyen, T. (2019). Formal requirements and constraints modelling in form-l for the engineering of complex socio-technical systems. In: IEEE 27th Int. Requirements Engineering Conference Workshops (REW), pp 123–132. https://doi.org/10.1109/REW.2019.00027

OMG. (2011). Semantics of a Foundational Subset for Executable UML Models (fUML). Formal/11-02-01.

OMG. (2017). OMG Unified Modeling Language (UML). Formal/17-12-05.

OMG. (2019). Precise Semantics of UML State Machines (PSSM). Formal/19-05-01.

OMG. (2021). Semantics of a Foundational Subset for Executable UML Models (fUML). Formal/21-03-01.

Pham, V. C., Radermacher, A., Gérard, S., et al. (2017). Complete code generation from UML state machine. In: MODELSWARD. SciTePress, pp 208–219. https://doi.org/10.5220/0006274502080219

Pnueli, A., & Shalev, M. (1991). What is in a step: On the semantics of statecharts. In: *Theoretical Aspects of Computer Software, International Conference TACS, LNCS, 526*. Springer, pp. 244–264. https://doi.org/10.1007/3-540-54415-1_49

Posse, E., & Dingel, J. (2016). An executable formal semantics for UML-RT. *Software and Systems Modeling, 15*(1), 179–217. https://doi.org/10.1007/s10270-014-0399-z

Purchase, H. C., Allder, J., & Carrington, D. A. (2000). User preference of graph layout aesthetics: A UML study. In: *Graph Drawing, 8th International Symposium, LNCS, 1984*. Springer, pp. 5–18. https://doi.org/10.1007/3-540-44541-2_2

Ratiu, D., Voelter, M., & Pavletic, D. (2018). Automated testing of DSL implementations - experiences from building mbeddr. *Software Quality Journal, 26*(4), 1483–1518. https://doi.org/10.1007/s11219-017-9390-6

Romero, A. G., Schneider, K., & Ferreira, M. G. V. (2014). Using the base semantics given by fUML for verification. In: MODELSWARD 2014. SciTePress, pp 5–16. https://doi.org/10.5220/0004662400050016

Seidewitz, E. (2003). What models mean. *IEEE Software, 20*(5), 26–32. https://doi.org/10.1109/MS.2003.1231147

Seidewitz, E. (2014). UML with meaning: executable modeling in foundational UML and the alf action language. In: SIGAda annual conference on High integrity language technology HILT. *ACM*, pp 61–68. https://doi.org/10.1145/2663171.2663187

Seidewitz, E., & Tatibouet, J. (2015). Tool paper: Combining Alf and UML in modeling tools - an example with Papyrus -. In: OCL@MoDELS, vol 1512. CEUR-WS.org, pp 105–119.

Selic, B. (2004). On the semantic foundations of standard UML 2.0. In: Formal Methods for the Design of Real-Time Systems, LNCS, vol 3185. Springer, pp 181–199, https://doi.org/10.1007/978-3-540-30080-9_6

Selic, B. (2012). The less well known UML - A short user guide. In: 12th Int. School on Formal Methods, Advanced Lectures, LNCS, vol 7320. Springer, pp 1–20. https://doi.org/10.1007/978-3-642-30982-3_1

Semeráth, O., Babikian, A. A., Chen, B., et al. (2021). Automated generation of consistent, diverse and structurally realistic graph models. *Software and Systems Modeling, 20*(5), 1713–1734. https://doi.org/10.1007/s10270-021-00884-z

Søndergaard, H., Korsholm, S. E., & Ravn, A. P. (2017). Conformance test development with the Java modeling language. *Concurrency and Computation: Practice and Experience, 29*(22). https://doi.org/10.1002/cpe.4071

Störrle, H. (2014). On the impact of layout quality to understanding UML diagrams: Size matters. In: MODELS, Lecture Notes in Computer Science, vol 8767. Springer, pp 518–534. https://doi.org/10.1007/978-3-319-11653-2_32

Wiesmayr, B., Zoitl, A., & Rabiser, R. (2021). Assessing the usefulness of a visual programming IDE for large-scale automation software. In: 2021 *ACM/IEEE 24th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. https://doi.org/10.1109/models50736.2021.00037

Zschaler, S., Bousse, E., Deantoni, J., et al. (2023). A generic framework for representing and analyzing model concurrency. *Software and Systems Modeling*. https://doi.org/10.1007/s10270-022-01073-2

Márton Elekes is a PhD student at the Budapest University of Technology and Economics. His research interests include software testing and graph databases. He is a contributor of the LDBC Social Network Benchmark. He participates in the Arrowhead Tools and EMBrACE EU projects.

Vince Molnár is an assistant professor at the Budapest University of Technology and Economics. His main research field is model-based development and formal methods, with the primary focus on concurrent, distributed and safety-critical systems. He is the leader of the development of the Gamma Statechart Composition Framework and also contributed to the development of the PetriDotNet and the Theta model checking frameworks. He participates in the EU projects EMBrACE and ADVANCE, and various industrial cooperations, including one with NASA-JPL. As a member of the SysMLv2 Submission Team, he works on the execution semantics of the new systems modeling language.

Zoltán Micskei received the M.S. and Ph.D. degree in software engineering from the Budapest University of Technology and Economics in 2005 and 2013. He is currently an associate professor at the same university, leading the Critical Systems Research Group. His research interests include software testing and model-based engineering with a focus on empirical studies. Dr. Micskei is a Senior Member of ACM. A publication he co-authored won the Most Influential Regular Paper award for the Journal on Software and Systems Modeling in 2021. He received the Kalmár Award from NJSZT in 2021.