Formalizing UML State Machines for Automated Verification – A Survey

ÉTIENNE ANDRÉ, Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
SHUANG LIU, College of Intelligence and Computing, Tianjin University, China
YANG LIU, Nanyang Technological University, Singapore
CHRISTINE CHOPPY, Université Sorbonne Paris Nord, LIPN, CNRS, UMR 7030, F-93430, France
JUN SUN, Singapore Management University, Singapore
JIN SONG DONG, National University of Singapore, Singapore

The Unified Modeling Language (UML) is a standard for modeling dynamic systems. UML behavioral state machines are used for modeling the dynamic behavior of object-oriented designs. The UML specification, maintained by the Object Management Group (OMG), is documented in natural language (in contrast to formal language). The inherent ambiguity of natural languages may introduce inconsistencies in the resulting state machine model. Formalizing UML state machine specification aims at solving the ambiguity problem and at providing a uniform view to software designers and developers. Such a formalization also aims at providing a foundation for automatic verification of UML state machine models, which can help to find software design vulnerabilities at an early stage and reduce the development cost. We provide here a comprehensive survey of existing work from 1997 to 2021 related to formalizing UML state machine semantics for the purpose of conducting model checking at the design stage.

CCS Concepts: • **Software and its engineering → Unified Modeling Language (UML)**; *Semantics*; **Visual languages;**

Additional Key Words and Phrases: UML, semantics, formal specification, formal verification

ACM Reference format:
Étienne André, Shuang Liu, Yang Liu, Christine Choppy, Jun Sun, and Jin Song Dong. 2023. Formalizing UML State Machines for Automated Verification – A Survey. *ACM Comput. Surv.* 55, 13s, Article 277 (July 2023), 47 pages.
https://doi.org/10.1145/3579821

This work is supported by project 9.10.11 "Software Verification from Design to Implementation" of French-Singaporean Programme Merlion.

Authors' addresses: É. André, Université de Lorraine, CNRS, Inria, LORIA, BP 239, 54506, Vandoeuvre-lès-Nancy, France; email: eandre93430@lipn13.fr; S. Liu, College of Intelligence and Computing, Tianjin University, Tianjin, China; liushuangcs@gmail.com; Y. Liu, Nanyang Technological University, 50 Nanyang Avenue, 639798, Singapore, Singapore; email: yangliu@ntu.edu.sg; C. Choppy, Université Sorbonne Paris Nord, LIPN, CNRS, UMR 7030, F-93430, Avenue Jean-Baptiste Clément, 93430, Villetaneuse, France; email: christine.choppy@lipn.univ-paris13.fr; J. Sun, Singapore Management University, 81 Victoria Street, 188065, Singapore, Singapore; email: junsun@smu.edu.sg; J. Song Dong, National University of Singapore, Computing Drive, Singapore, Singapore; email: dongjs@comp.nus.edu.sg.

0360-0300/2023/07-ART277 $15.00
https://doi.org/10.1145/3579821

1 INTRODUCTION

The **Unified Modeling Language (UML)** [170] is a standard for modeling dynamic systems. UML behavioral state machines, an object-oriented variation of Harel's statecharts [95], can be used to model the dynamic behaviors of a system. The UML is considered to "have become a *de facto* 'standard' for describing object-oriented design models, supported by a range of software tools and textbooks" [45]. The UML specification, published and managed by the **Object Management Group (OMG)**, is written in natural language. Although this description is named "formal specification" and contains numerous details, it can be at most referred to as a "semi-formal" specification (a term often met in the literature). This "formalization" in natural language introduces ambiguities and inconsistencies, which are tedious for manual detection or to be verified automatically due to the lack of formal semantics. In fact, the meaning of "formal" in that "formal specification" could even be understood as "formally adopted by the committee," and not as in "formal methods."

Defining a formal semantics for UML state machines has been capturing a large attention since the mid-1990s. The benefit of a formal UML state machine semantics is threefold. First, it allows more precise and efficient communication between engineers. Second, it yields more consistent and rigorous models. Last and most importantly, it enables automatic formal verification of UML state machine models through techniques like model checking [21], which enables the verification of properties in the early development stage of a system. This results in a possible reduction in the overall cost of the software development cycle.

Since the mid-1990s, a number of works appeared in the literature, which provide formalization for UML state machines usually for the purpose of performing model checking or more generally formal verification. Those approaches adopt different semantic models, support different subsets of UML state machine features, and only some of them are supported by an implementation.

In the past, three main surveys attempted to summarize the various methods for formalizing UML state machines, viz., [37, 62, 146]. The work in Reference [62] is the most complete and accurate, but it is now out-of-date, and hence does not cover recent approaches. Figure 2 shows that the formalization of UML state machines is still an active topic, and many recent works (typically all written after 2010) are not covered by any of these surveys.

Contribution. In this manuscript, we survey approaches aiming at formalizing UML behavioral state machines, with a specific focus on the automated verification support for UML state machines. Our survey provides comparisons of those approaches in two dimensions. The first dimension is the semantic model used by the approaches. The existing approaches in the literature can be divided into two main categories:

(1) approaches that translate UML state machines into an existing formal language and
(2) approaches that directly provide an *ad-hoc* operational semantics for UML state machines.

The approaches in the first category provide translation rules from UML state machines to some existing formal language, such as PROMELA (which is the input language of the Spin model checker [100]) or extensions of automata and Petri nets. The second category of approaches provides operational semantics for UML state machines, generally in terms of inference rules, which is a common format for formalizing structural operational semantics.

For the second dimension, we compare those approaches with respect to the covered features, such as entry or exit behaviors, run-to-completion step, deferred events, and tool support to automatically verify UML state machine models.

The contributions of the survey are as follows:

(1) Our first and main contribution is to provide an overview of the status of researches in the area of formalizing UML state machines.

(2) As a second contribution, we provide a study of tool supports for the formal verification of UML state machine models. We find out that, despite the interesting development of several formally grounded prototype tools, quite disappointingly, most of these tools are unavailable nowadays, seemingly lost over the years.
(3) Last, we draw general conclusions and identify future directions of research in this area.

The rest of this survey is organized as follows: Section 2 briefly recalls the syntax and informal semantics of UML state machines. Section 3 defines the categorization criteria of our survey. Sections 4 and 5 discuss the translation approaches and the formalization approaches, respectively; we also summarize these works and draw some higher-level conclusions. Section 6 surveys the existing tools supporting formal verification of UML state machines. Section 7 discusses the related surveys. Section 8 concludes the article and gives some perspectives.

2 UML STATE MACHINES

A brief history of UML versions. After a draft (named "1.0") was proposed in 1996, the first actual version of the UML specification (named "UML 1.1") was proposed by the OMG in December 1997. After a minor update named "UML 1.2" was released in July 1999, a more significant upgrade named UML 1.3 was released in February 2000. Then, UML 1.4 was released in September 2001, followed by UML 1.5 in March 2003.

In July 2005, a major change was brought to the UML with the release of UML 2.0: Several new diagrams were proposed, and several existing diagrams (including state machines) were modified. Several subsequent updates followed: UML 2.1.2 in October 2007, UML 2.2 in January 2009, UML 2.3 in May 2010, UML 2.4.1 in July 2011, and UML 2.5.1 in December 2017 [170].

In this section, we focus on the latest stable version, i.e., UML 2.5.1 [170]. However, the various works surveyed here can address various versions of UML, which we will discuss.

UML state machines in a nutshell. The paradigm of UML behavioral state machines [170] is that of a finite-state automaton: That is, each entity (or subentity) is in one state at any time and can move to another state through a well-defined conditional transition. We assume the reader's basic knowledge on state machines, and we only briefly recall here the syntax and informal semantics of state machines. The UML provides an extended range of constructs for state machines: simple/composite states, entry/exit/do behaviors, concurrency (regions in composite states, fork/join transitions), shared variables, shallow and deep history pseudostates, and so on. The use of composite states or submachine states allows to define state machines in a hierarchical manner. Communication is ensured via a (synchronous or asynchronous) broadcast mechanism and through variables.

In the following, we briefly recall these elements. We use as a running example the state machine diagram given in Figure 1.

2.1 States

The UML defines three main kinds of states: *simple* states, *composite* states (that can be *orthogonal*), and *submachine* states. A simple state (e.g., S1 or S21 in Figure 1) has no region, and hence contains no "internal state" such as a region or submachine; internal transitions are still allowed in a simple state [170, p. 320].

A *composite state* (e.g., S2 or S3 in Figure 1) is a state that contains at least one region and can be a *simple* composite state or an *orthogonal* state. A simple composite state (e.g., S3 in Figure 1) has exactly one region, which can contain other states, allowing to construct hierarchical state machines. An orthogonal state (S2 and S5 in Figure 1) has multiple regions (regions can contain other states), allowing to represent concurrency. Regions are separated using a dashed line.

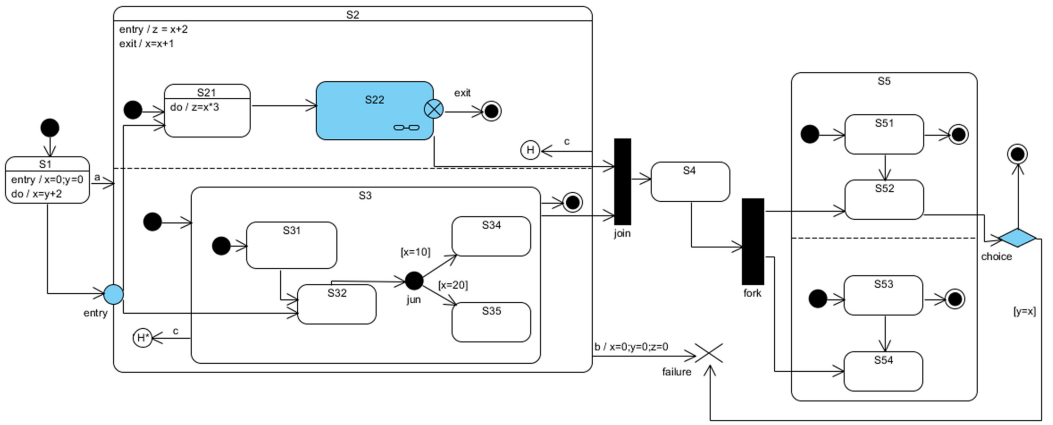

Fig. 1. An example of a UML state machine diagram.

A *submachine state* (e.g., S22 in Figure 1) refers to an entire state machine that can be nested within a state. It is said to be "semantically equivalent to a composite State" [170, p. 311]. (In Figure 1, the actual definition of S22 is not given for sake of conciseness.)

Final states. A final state (e.g., the two right-most states in S5 in Figure 1) is a special kind of state enclosed in a region to indicate when the region has completed. (The UML assumes that the state machine itself is a region, which explains that a final state can be defined at level zero, i.e., the final state right of S5 in Figure 1.)

2.2 Behaviors

Behaviors may be defined when entering states ("entry behavior"), when exiting states ("exit behavior"), while in states ("do behavior"), or when firing transitions. The entry behavior is executed when the state is entered: This can be the case of a default entry ("graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state" [170, p. 310]), an explicit entry, a shallow or deep history entry, or an entry point entry. Similarly, the exit behavior is executed when the state is exited. The exit behavior of a source state is not executed in the case of a local or internal transition. (We give in Section 2.6 more information on the actual sequence of entry and exit behaviors to be executed when firing a transition.) The do behavior is executed only after the execution of the entry behavior of the state and continues to be executing (in parallel with others behaviors, if any) until it completes or the state is exited. In Figure 1, the entry behavior of S2 is "`z = x+2`" while its exit behavior is "`x = x+1`."

2.3 Pseudostates

A main difference between a state and a pseudostate is that the active state configuration of a system only consists of *states*—but not of pseudostates. The UML defines different kinds of pseudostates.

Initial. Each composite state (simple or orthogonal) and each state machine may have an initial pseudostate (e.g., the left-most node in Figure 1).

History. Only composite states can have, at most, one history pseudostate in each region. The UML defines two kinds of history pseudostate: *shallow history* pseudostates and *deep history* pseudostates.

A *shallow history pseudostate* is a kind of variable that represents the most recent active state configuration of its containing state but not the substates of that substate, which means that the pseudostate saves only the latest visited state inside its containing composite state. Shallow history pseudostates are depicted using an "H." In Figure 1, there is one shallow history pseudostate in the upper region of S2.

A *deep history pseudostate* is a kind of variable that represents the most recent active state configuration of its owning state. That is, a deep history pseudostate saves the most recent active state configuration of all visited states inside the containing composite state (the state configuration is restored when a transition is terminating on the pseudostate). Deep history pseudostates are depicted using an "H*." In Figure 1, there is one deep history pseudostate in the lower region of S2. Let us illustrate the difference between shallow and history pseudostate on this example. Assume the lower region of S3 is currently in S32; assume this deep history pseudostate is visited (via trigger "c"). Then the new active state becomes (again) S32 (and entry behaviors are again executed). However, if this deep history pseudostate were replaced with a *shallow* history pseudostate, then the new active state would become S31, as only the first level of hierarchy (S3) would be memorized.

Fork and join. The *join* pseudostate "serves as a common target vertex for two or more transitions originating from vertices in different orthogonal regions" [170, p. 311]. Therefore, join pseudostates can be seen as a synchronization function; they cannot have a guard or a trigger (event). The outgoing transition from a join pseudostate is executed only after the execution of all incoming transitions.

The *fork* pseudostate serves "to split an incoming transition into two or more transitions terminating on vertices in orthogonal regions of a composite state" [170, p. 311]. Similarly to the join pseudostate, the transitions outgoing from a fork pseudostate cannot have a guard or a trigger.

For example, in Figure 1, the pseudostate source of the transition with S4 as a target is a join pseudostate, while the pseudostate target of the transition with S4 as a source is a fork pseudostate.

Choice and junction. The UML defines two kinds of pseudostates that allow to merge and/or choose between various flows, i.e., they can have multiple incoming and outgoing transitions. The transitions can be guarded by Boolean expressions, and the system chooses nondeterministically one of the outgoing transitions for which the guard evaluates to true. The main difference between choice and junction pseudostates is that the guards are evaluated dynamically (after the exit behaviors are performed and also after the effect of transition segments before reaching the choice) in choice pseudostates, whereas they are evaluated statically (before any compound transition containing this pseudostate is executed) in junction pseudostates. In Figure 1, the right-most diamond with one incoming arc (from S52) and two outgoing arcs is a choice pseudostate; and the node labeled with jun within S3 in Figure 1 is a junction.

Terminate. Entering a terminate pseudostate (e.g., the pseudostate labeled with "failure" in Figure 1) implies that the execution of the state machine is terminated immediately, without performing any behavior.

Entry and exit points. "An entry point pseudostate represents an entry point for a state machine or a composite state that provides encapsulation of the insides of the state or state machine" [170, p. 317]. The exit point is the dual concept of entry point. For example, the node in the lower left on the border of S2 is an entry point; if it is entered, the system enters the regions of S2 via S21 and S32, without passing through the (regular) initial pseudostates. Conversely, the right-most node of S22 is an exit point of this submachine state. Note that entry and exit points in composite states are very close semantically to forks and joins.

2.4 Transitions

The UML defines three kinds of transitions: external, local, and internal.

- External transitions are between two different vertices. This transition exits its source vertex, and the exit behavior of the incoming state is executed.
- Local transitions can only be defined within a composite state and are such that "the transition does not exit its containing state (and, hence, the exit behavior of the containing state will not be executed)" [170, p. 314].
- Internal transitions are special local self-transitions (with same source and target). "The state is never exited (and, thus, not re-entered), which means that no exit or entry behaviors are executed when this transition is executed" [170, p. 314].

For example, the transition to the history pseudostate of the upper region of `S2` and labelled with event `c` in Figure 1 is a local transition.

Transitions are executed as part of a more complex *compound* transition that takes a state machine execution from one stable state configuration to another. For example, in Figure 1, when in `S4`, a compound transition can be taken that contains the incoming and the two outgoing transitions of the fork pseudostate.

We also emphasize so-called "inter-level transitions," which are a kind of transition that crosses the border of some composite state (called "multi-level" in Reference [188]). For example, the transition from `S22` to the `join` pseudostate in Figure 1 is an inter-level transition.

Each kind of transition can have a guard (e.g., `y=x` on the right-most transition in Figure 1), a trigger (e.g., `a` on the left-most transition in Figure 1), which can be seen as an event, a behavior (e.g., `x=0;y=0;z=0` on the left transition leading to the terminate pseudostate), and can be a completion transition (a transition without event) or a transition with event. A completion transition (never labeled with an event) is a transition that is taken when a state finished its activity; in case of composite states, all regions must be in their final state.

Often, variables (integers, Booleans, etc.) can be used in state machines (notably, in behaviors), and then tested in guards and updated along transitions. This is also the case of the examples of the specification (e.g., Reference [170, Figures 14.24 and 14.25]). In Figure 1, `x`, `y`, and `z` are (integer-valued) variables.

2.5 Deferred Events

A deferrable trigger allows to postpone the handling of the "request" event occurrence to subsequent states.

2.6 Run-to-completion Paradigm

A central notion in UML state machines is the *run-to-completion step*, which we briefly recall. Events are processed one-by-one, and only when the state machine is in a stable configuration. That is, an event cannot be processed during the processing of another event (entry or exit behaviors, etc.).

More specifically, an event in the event pool is selected; then, among the list of enabled transitions, one or more transitions will be executed for firing. In an orthogonal composite state, different compound transitions with the same event can be executed (in an undefined order) during the same step. This set has to be maximal and conflict-free. A priority mechanism is defined in the UML specification to solve conflicts, i.e., to decide which transition will fire when such a choice is to be made: Without going into full details, this mechanism will generally give higher priority to transitions in substates. That is, "a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states" [170, p. 317]. Note that even

with the transition selection algorithm, non-determinism remains: It may happen that different (sets of) transition(s) could be selected; only one such set will be actually selected, and therefore, the execution is non-deterministic. When performing model checking, this means that *all* such sets of transitions must be explored to assess some formal property.

Once a set of transitions is chosen for a given event, the following actions are performed: (1) the active state exit behavior is executed, followed by the exit behaviors of the containing states, up to a "least common ancestor" (i.e., the "innermost" region that contains both the source and the target state concerned by the transition); (2) the transition behavior is executed; (3) the entry behaviors are executed, starting from the outermost state in the least common ancestor region that contains the target state, until the target state entry behavior itself.

This whole notion is rather delicate, and its proper handling will be one of the key features we will survey in this article.

3 METHODOLOGY AND CATEGORIZATION CRITERIA

3.1 Methodology

In this article, we survey works formalizing UML state machines for the purpose of automated verification. We conducted a systematic literature review (SLR, see, e.g., References [44, 116, 212]), explained in the following. We collected works using different methods: (*i*) going through the former surveys [37, 62, 146]; (*ii*) performing search engine requests, typically DBLP and Google Scholar; (*iii*) collecting citations to our own works (notably, Reference [144]); and (*iv*) collecting systematically relevant citations from and to all the aforementioned works, until reaching a fixpoint.

3.1.1 Criteria for Inclusion. The works we survey must address as main goal the formalization of UML behavioral state machines, with automated verification as the ultimate objective. We therefore exclude purely theoretical approaches, except when they can be (potentially) used for formal automated verification.

Time range. We focus on works published between December 1997 and the end of December 2021. The start date of our considered time range (December 1997) is the publication of the first official version (named "UML1.1") by the OMG. The end date (end of December 2021) corresponds roughly to the time of revising our survey; as the research on formalizing UML state machines is still ongoing, we have to set up an end date for the collection of the surveyed works, which can seem arbitrary.

3.1.2 Criteria for Exclusion. We exclude publications formalizing diagrams prior to the OMG formalization; this can be the case of "Harel's statecharts" (e.g., Reference [162]). In particular, any work prior to December 1997 is excluded from our survey.

We exclude publications strictly subsumed by others; this is generally the case of journal versions of conference papers (e.g., Reference [60] is subsumed by References [59], [190] are subsumed by Reference [191]).

We exclude publications in which the formalization of state machines is really too shallow; this can be the case of publications formalizing other UML diagrams (e.g., activity diagrams), with only a small focus on state machines.

We exclude publications related to formalisms close to (but different from) UML state machines. We exclude significant extensions of UML, such as krtUML (for which an executable semantics is discussed in Reference [64]), or SysML (partially formalized in, e.g., Reference [102]). The same applies to UML-RT. Similarly, the **Foundational UML (fUML)** [171], which aims at providing an executable semantics for a subset of the UML syntax, is excluded from our work. We, however, discuss fUML in Section 8.

Table 1. Candidate Articles Not Selected

Work	Authors	Reason for exclusion
[177]	Pnueli and Shalev (1991)	out of time range
[205]	Uselton and Smolka (1994)	out of time range
[98]	Harel and Naamad (1997)	out of time range
[96]	Harel and Gery (1997)	out of time range
[42]	Breu et al. (1997)	out of time range
[43]	Bruel and France (1998)	no technical details
[77]	Evans et al. (1998)	no technical formalization
[99]	Harel and Politi (1998)	not focusing on UML SMDs
[148]	Lüttgen et al. (1999)	not UML SMDs
[9–13]	Alur et al. (1999–2004)	not focusing on UML SMDs
[58]	Clarke and Heinle (2000)	not UML SMDs
[60]	Compton et al. (2000)	seems subsumed by Reference [59]
[53]	Börger and Schmid (2000)	not specifically focusing on SMDs
[19]	Aoki et al. (2001)	too shallow
[49]	Börger et al. (2001)	only local aspects of formalization
[190]	Shen et al. (2001)	subsumed by Reference [191]
[50]	Börger et al. (2001)	only local aspects of formalization
[28]	Behrmann et al. (2002)	not focusing on UML SMDs
[29]	Bernardi et al. (2002)	relies on Reference [158]
[65]	Damm et al. (2003)	complements [63]
[41]	Bozga et al. (2004)	no formal semantics
[168]	Ober et al. (2004)	subsumed by Reference [169]
[85]	Fox and Luangsodsai (2005)	no formal semantics
[64]	Damm et al. (2005)	focusing on krtUML
[86–88]	Furfaro et al. (2005–2007)	not UML SMDs
[128]	Kyas et al. (2005)	no details on SMDs
[83]	Fekih et al. (2006)	reverse translation from B
[204]	Truong and Souquières (2006)	no details on SMDs
[208]	von der Beeck (2006)	focusing on UML-RT
[75]	Dubrovin et al. (2008)	focusing on the symbolic step
[200]	Thierry-Mieg and Hillah (2008)	no details on SMDs
[156]	Mekki et al. (2009)	pattern-based subset of UML SMDs
[199]	ter Beek et al. (2011)	UMC rather than UML
[159]	Miao (2011)	no formal semantics
[218]	Zurowska and Dingel (2012)	focusing on UML-RT
[16]	Androutsopoulos et al. (2013)	not UML SMDs
[46]	Butler et al. (2013)	no details on SMDs
[102]	Jacobs and Simpson (2015)	focusing on SysML
[120]	Knapp et al. (2015)	mostly subsumed by Reference [118]
[14]	Amtoft et al. (2020)	not focusing on UML SMDs
[6]	Al-Fedaghi (2020)	no formal semantics
[92]	Haga et al. (2022)	out of time range
[180]	Rosenberger et al. (2022)	out of time range

We tabulate the non-selected works in Table 1, with a reason for exclusion. We discuss some of these excluded publications in the following:

In a series of works [9–11], Alur et al. discuss several issues related to *hierarchic reactive machines* and, notably, semantic issues. While some of these works were discussed in the former survey [37], we did not integrate this line of works in our survey, because their semantics differs too much from that of UML state machines. In Reference [9], it is explicitly mentioned that the "mode" (as a central component of their behavioral description) has several strong semantic

differences with statecharts and UML state machines: Such differences include, notably, the fact that (*i*) transitions can originate from and target entry/exit points only, (*ii*) a default exit always retains the history, and (*iii*) the priority between transitions differs. In that sense, the semantics of Reference [9] is closer to Room [189] than UML state machines. In Reference [13], hierarchical state machines are discussed; they differ too significantly from UML state machines to be integrated to our survey. In Reference [12], hierarchical reactive modules [9] are considered. Again, while sharing some similarities with UML state machines, the semantics of actual UML state machines is not discussed.

In Reference [28], a version of finite state machines called **hierarchical state/event machines (HSEMs)** is discussed; as explicitly mentioned in Reference [28], their semantics differ from UML state machines.

In Reference [19], a formalization of class and state machine diagrams is given via an axiom system. The syntactic features of UML state machines are very scarce (not even composite states nor final states), which rules out this work as a real formalization attempt. An implementation in the HOL theorem proving system is, however, proposed.

The case of the UMC framework [199] is somehow borderline: Ter Beek et al. propose a UML-like framework for systems modeled using a collection of UML-like state machines. Properties can be specified using the UML-oriented state-based and event-based logic UCTL. While UMC is claimed to (partially) match the UML informal semantics ("UMC makes certain assumptions that, while compatible with the UML standard, are not necessarily the only possible choice" [199]), UMC is not exactly UML either. The UMC documentation [153] does not really give a formal semantics to UML, but rather explains "how UMC can be used to generate system models according to the UML paradigm." Also, it is mentioned that it is expected that "users directly use UMC for writing their model specification." However, the UMC documentation [153] still handles some rather subtle aspects of the UML semantics, such as completion transitions and recursive or parallel operation calls. In the end, we do not integrate the semantic framework in our survey, but we do mention the KandISTI/UMC tool suite in Section 6.1.

Finally, note that we still used as much as possible the "excluded publications" in our search, i.e., to gather further references until reaching a fixpoint.

3.1.3 Summary. We ended up selecting 61 works (to be surveyed in the following and integrated to our comparison tables, Tables 2 to 5).

3.2 Categorization

Some of the surveyed works directly provide an operational semantics for UML state machines in the form of inference rule or **SOS (structured operational semantics)**; hence, dedicated verification tools can be developed based on these semantics. This is what we call the *direct approach*. Other works conduct an *indirect approach*: They translate UML state machines into an existing specification language, which is usually supported as an input language by model-checking tools.

Based on the above observations, we categorized the surveyed approaches in two dimensions. The first and main dimension is whether the approach is a direct or an indirect approach. Hence, we split the main body of our survey into two main sections:

(1) Section 4 surveys the indirect approaches, i.e., the translation-based approaches;
(2) Section 5 surveys the direct approaches, i.e., the approaches that define an operational semantics for UML state machines.

As a second dimension, we compare the surveyed works on the features supported, semantic models used, UML specification version, and so on.

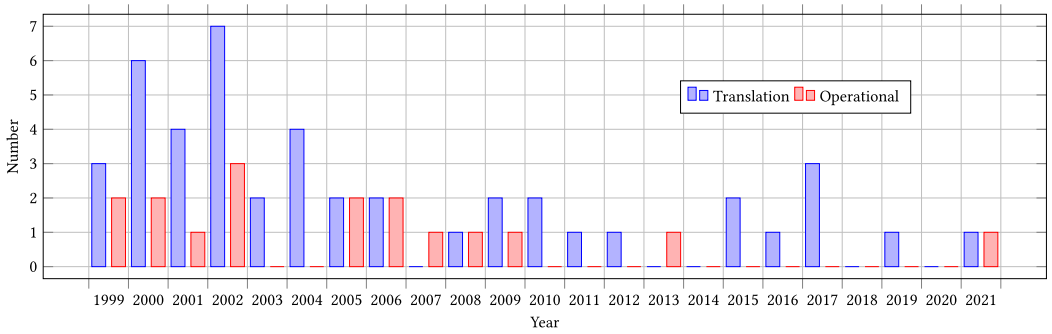

Fig. 2. Number of selected works along the years.

We focus in particular on approaches with tool support. These tools are mentioned throughout the survey and are then specifically gathered in Section 6. A main outcome of our survey concerning tools is that, quite unfortunately, many tools are nowadays unavailable.

Time range. We depict in Figure 2 the number of publications per year of the surveyed works by splitting them among the two identified categories. While many works are concentrated around the 1999–2004 period, the research in formalizing UML state machines never really decreased, with still a number of works in the 2010s, up to the early 2020s. And in fact, some very recent works even go beyond our time range (e.g., References [92, 180]), showing a still-active field.

4 THE TRANSLATION APPROACHES

A popular approach to formalize UML state machines is to provide a translation to some existing formal language (such as Abstract State Machines [54] or automata). These formal languages have their own operational semantics, and they are usually the input languages of model checkers (such as Spin [100], NuSMV [57], or Uppaal [134]). This kind of approach can be regarded as an indirect way of providing formalizations for UML state machines. The purpose of this kind of approach is to utilize existing verification techniques and tools.

We categorize these approaches based on the target formal languages they adopt, viz., abstract state machines (Section 4.1), graph transformation (Section 4.2), automata and extensions (Section 4.3), Petri nets and extensions (Section 4.4), the input language of SMV and NuSMV (Section 4.5), extensions of CSP (Section 4.6), PVS, KIV and (extensions of) B (Section 4.7), and using institutions (Section 4.8). We summarize the translation-based approaches in Section 4.9 and draw conclusions on these approaches. In the following subsections, we only briefly mention the syntactic aspects covered by each work; the full list of these aspects will be summarized in Table 2.

4.1 Translation into Abstract State Machines

Abstract State Machines (ASMs) [54] offer a very general notion of state in the form of structures of arbitrary data and operations, which can be tailored to any desired level of abstraction. State machines' configuration changes are represented by transition rules, which consist of conditions and update functions. However, the notion of multi-agent (distributed) ASMs can naturally reflect the interaction between objects [54].

References [27, 68, 195] provide theoretic and tool support for model checking abstract state machines.

4.1.1 Translation into ASMs. Börger et al. [48, 51, 52] are among the pioneers in formalizing UML state machines using ASMs. ASMs contain a collection of states and a collection of rules (conditional, update, Do-forall, etc.) that update those states.

The first work in this direction is Reference [48] (in fact, this work relies on a previous attempt to formalize not state machines but activity diagrams [47]). This approach covers many UML state machine features, including shallow history and deep history pseudostates, internal/external transitions, deferred events, completion events, and internal activities associated with states—which is important to note, as these features are discarded by most other approaches. However, pseudostates such as fork, join, junction, choice (the latter being perhaps less straightforward) are surprisingly not directly considered. The authors argue that these constructs can find their semantically equivalent constructs in their defined subset, where they use a transition from (respectively, to) the boundary of an orthogonal composite state to replace the join (respectively, fork) pseudostates.

In 2003, another work [52] extends Reference [48] to support transitions from and to orthogonal composite states[1] in the context of event deferral and run-to-completion step.

In 2004, Reference [51] provides some further discussions about the ambiguities in the official semantics of UML state machines and their solutions.

The works by Börger et al. [48, 51, 52] cover a large set of features and the formalization is easy to follow due to the abstract feature of ASM notations. However, to the best of our knowledge, no automatic translating tool was developed based on these works.

Another approach that also translates UML state machines into ASMs was proposed by Compton et al. in a technical report [59] (a shorter version was proposed in Reference [60]). To be precise, this work translates UML state machines into *extended* ASMs, in which ASMs are extended to represent inter-level transitions with multiple transitions that do not cross any boundary of states. This extension makes it easier to deal with interruptions; it also makes the formalization procedure more structured and layered (since inter-level transitions break the hierarchical structure of UML state machine, and such a decomposition of inter-level transitions into multiple transitions preserve such a hierarchical structure). Agents are used to process executions of UML state machines. An activity agent is used to model the execution of an activity associated with a node. The execution of agents is divided into different modes, which indicate what kind of rules (operations) the current agent should take. Entry and exit behaviors are considered, as well as orthogonal states, guards, and completion events. History pseudostates, deferred events,[2] forks, joins, choices, junctions are not considered. Finally, the authors use SMV as a backend—which will be discussed in Section 4.5.

Finally, Jürjens also provides in Reference [113] a semantics in the form of abstract state machines, extending the semantics given in Reference [48]. However, in contrast to Reference [48], the focus of Reference [113] is not on supporting various features of UML state machine. Instead, it rather focuses on the communications aspects between state machines. The work explicitly models the message (with parameters) passing between state machines as well as the event queue.[3] Composite states, guards, events, and (of course) message passing between various state machines is supported. However, several common syntactic elements are discarded, e.g., history pseudostates, transitions from or to composite states other than completion transitions, deferred events, fork/joins, choice/junctions, and so on. Note that consistency between various diagrams (including activity diagrams and class diagrams) is considered by the same author in Reference [112].

[1]The orthogonal composite state acts as the main source/target state of the transition, i.e., the source/target of the transition can be a substate of the orthogonal composite state at any depth.

[2]In fact, the paper is a little ambiguous; on the one hand, it is explicitly stated that "deferred events are not considered" [59, p. 22]; on the other hand, it seems that the ASM formalization does consider (at least in some definitions) deferred events.

[3]Note that the UML specification [170] mentions an event *pool*.

4.1.2 Translation into Object Mapping Automata. Jin et al. [108] provide an approach that syntactically defines UML statecharts as attributed graphs, which are described using the **Graph Type Definition Language (GTDL)**. This work extends their previous work [107]. They further provide some constraints in the form of predicates to specify the well-formedness rules of statecharts, which is considered as the static semantics of a UML statechart. The semantic domain is defined as an **Object Mapping Automaton (OMA)** [103], which is a variant of ASMs. Given the abstract syntax (of the attributed graph) of a well-formed statechart, they first "compile" it into OMA algebraic structures, which specifies "advanced static semantics" of a UML statechart. Based on OMA algebraic structures, two rules (viz., the initialization rule and the run-to-completion rule) are defined to describe the dynamic behavior of a UML statechart. The syntax and semantics provided by this approach are more intuitive and easy to follow, benefiting from the highly compatibility of the abstract syntax of attributed graph with UML statecharts.

This work is quite complete in terms of syntax, as its supports completion events, history pseudostates (deep and shallow, including default entry), deferred events, entry/exit/do behaviors, internal and compound transitions, as well as inter-level transitions.

Discussion. Approaches translating UML state machines into ASMs tend to support more advanced features such as orthogonal composite states, completion/defer events, fork/join/history/choice pseudostates, and inter-level transitions. In Table 2, the global score (last column), which is a way to measure the number of syntactic features considered, is among the highest of the translation approaches; in particular, Reference [108] has the highest score of all approaches surveyed (together with Reference [201]; see Section 4.7). A reason may be that ASMs are more flexible in terms of syntax format as well as update rules and are more suitable to express the non-structured feature of UML state machines.

4.2 Translation via Graph Transformation

In Reference [125], Kuske proposes a formalization of UML state machines using graph transformation. The subset of syntactic elements is rather small, while not trivial either. No automation of the translation is made, and no example of the verification of some property (even manually) is presented. This approach is then extended with other UML diagrams in Reference [126].

In Reference [123], Kong et al. propose another approach to formalize UML state machines using graph transformation. The supported syntax is much wider than in Reference [125]; but, oddly enough, some features such as (entry/exit) behaviors do not seem to be supported. It is also unclear whether the run-to-completion step is properly encoded. This still makes this approach one of the most complete translation-based approaches.

4.3 Translation into Automata

4.3.1 Translation into Extended Hierarchical Automata. In Reference [162], Mikk et al. propose **extended hierarchical automata (EHAs)** to encode the semantics of Harel's statecharts. EHAs are basically extensions of finite-state automata with hierarchical capabilities very similar to the notion of hierarchy in state machines. An operational semantics is given to EHAs, and a translation from statecharts to EHAs is then described.

Latella et al. are among the first researchers who contributed to the formal verification of UML state machines. Reference [136] utilizes EHAs defined in Reference [162] as an intermediate representation of UML state machines; then they define the formal semantics of EHAs using Kripke structures.

Gnesi et al. [89] propose a translation approach based on the formalization of UML state machines in their early work [136]. The translation is from a hierarchical automaton into a **labeled transition system (LTS)**. The LTS is then further translated into the FC2 format, which is the

standard input format of JACK [39], an environment based on process algebras, automata, and temporal logic formalisms. The model checking is done w.r.t. a correctness property expressed in the action-based temporal logics ACTL.

4.3.2 Translation into PROMELA. PROMELA is the input language of the Spin model checker [100].

Based on the formalization work using EHAs [136], Latella et al. proceed one step further in Reference [135] by providing an automated translation from UML state machines to a PROMELA model. The translation takes a hierarchical automaton as input and generates a PROMELA model as output. A dedicated PROMELA process (called *STEP*) is defined to encode the run-to-completion step in UML state machines, which includes the following sequence: (1) dispatching events from the environment; (2) identifying candidate transitions to fire; (3) solving conflicts and selecting firable transitions; (4) performing the actual execution of the selected transitions (including identifying the next configuration after execution of the current transition and possible side effects, which are events generated during the execution of actions associated with the transition). The run-to-completion step is, as indicated by the name itself, non-interruptable (but can be stopped[4]). This non-interruptable nature is guaranteed by the PROMELA "**atomic**" command. This mechanism guarantees in particular that "the only values available for verification are those which variables evaluate to *at the end* of each cycle" [135].

The translation process is structured, since it is based on the predefined formal semantics of EHAs [136]. The authors also provide a proof for the translation to guarantee the correctness of the procedure w.r.t. the semantics they defined for UML state machines.

Schäfer et al. [185] provide a method to model checking UML state machines as well as collaborations with the other UML diagrams. They translate, on the one hand, UML state machines into a PROMELA model and, on the other hand, collaborations into sets of Büchi automata; then invoke the Spin model checker to verify the model against the automata. Each state in the state machine is mapped to an individual PROMELA process. Two additional PROMELA processes are generated to handle event dispatching and transitions. The event queue is modeled as buffered channels, and communication among processes is modeled via unbuffered channels, i.e., they are synchronized. This approach further considers the consistencies between UML diagrams, i.e., collaboration diagram and state machine diagram. The possible communications among objects shown in a collaboration diagram should be consistent with the dynamic behavior represented in the state machine diagram. By translating collaboration diagrams into sets of Büchi automata, which is the form of property to be checked against the model, this approach checks the consistencies between the two diagrams. The approach is implemented in the tool HUGO.

Jussila et al. provide in Reference [111] another approach to translate UML state machines into PROMELA models. This approach considers multiple objects interacting with each other. The translation is based on a formally defined semantics of UML state machines. It supports initial and choice pseudostates as well as deferred and completion events. It further provides an action language, a subset of the Jumbala [73] action language, that is used to specify guard constraints and the effects of transitions of a UML state machine. The authors implemented a tool called PROCO that takes a UML model in the form of XMI files and outputs a PROMELA model. Another translation for non-hierarchical state machines is also presented.

Carlsson and Johansson [55] have designed a prototype tool to link Spin with RSARTE, a modeling tool for UML diagrams. Their work focuses on all kinds of RT-UML diagrams, i.e., UML diagrams related with real-time features. As part of UML, state machines are also translated into

[4]The difference between interrupt and stop relies in the fact that interrupt means a temporary stop that needs to be resumed afterwards, whereas stop means a permanent stop without resuming.

PROMELA in their approach. Since their work is not aiming at specifically model checking UML state machines, it does not provide detailed discussions about each feature of UML state machines but discusses the communications between different objects.

4.3.3 Translation into Timed Automata. Timed automata are an extension of finite-state automata with *clocks*, i.e., real-valued variables that can be compared to integer constants along transitions ("guards") or in discrete states ("invariants") [8]. They represent a powerful formalism to reason about systems featuring both concurrency and timing aspects. Timed automata are supported by several model checkers, of which the most famous one is certainly Uppaal [134].

One of the earliest works using (an extension of) timed automata as the target formalism for formalizing UML state machines is performed by David et al. in Reference [66]. A large subset of UML state machines is translated into *hierarchical* timed automata, an ad hoc extension of timed automata. These hierarchical timed automata are subsequently translated into "flat" timed automata. Verification is done with Uppaal.

Knapp et al. present in Reference [117] another approach to translate timed UML state machines into timed automata. Event queue and UML state machine are separately modeled by timed automata and the communication is modeled with a channel. This approach is implemented in hugo/RT, which translates UML state machines into the Uppaal model checker, which can then verify whether scenarios specified by UML collaborations with time constraints are consistent with the corresponding set of timed UML state machines. Note that the syntactic features displayed in Table 2 are those described in Reference [117]. However, the tool hugo/RT was significantly enhanced since then, and all language constructs for UML state machines are now supported, with the exception of submachines, connection point references, and entry/exit points (see Section 6).

Ober et al. present in Reference [168] an approach to translate a timed extension of a subset of UML state machines into communicating extended timed automata. Not a lot of syntactic elements are considered, but two key concepts (the run-to-completion step and proper priority handling) are encoded. Some constructs seem to come from Reference [162], but are not further detailed. The IF toolset [41] (whose formal language is based on communicating extended timed automata) is used as an underlying verification engine; properties can be specified using observer automata. Then, Ober et al. define in Reference [169] (seemingly extending Reference [168]) a UML profile called "OMEGA UML," extending UML state machines using real-time extensions. The syntax is translated into the input language of the IF toolset [41]. Again, properties can be specified using a lightweight extension of the UML syntax. While this paper [169] does not focus in detail on UML state machines (arguing they have been formalized in former works), again, the run-to-completion step is properly encoded. Verification and simulation can be performed using the IF toolset.

Finally, Mekki et al. propose in Reference [156] an approach to translate UML statechart "patterns" into a network of timed automata. The verification then reduces to reachability checking using a model checker for timed automata. This approach is of particular interest, as it avoids the designer to understand all subtle aspects of UML state machines and allows them to compose predefined patterns instead. These patterns involve temporal and timed requirements (expressing concepts such as minimum and maximum delays, latency, simultaneity or sequence, in the line of other works relating patterns and timed automata [15, 71]); concurrency (in the line of orthogonal composite states in UML state machines) is also considered. We do not integrate this work in our subsequent summaries (Table 2) though, as it does not take as input regular UML state machines, but only a set defined by an ad hoc "patterns" grammar.

Discussion. From Table 2, it is clear that the automata-based translation approaches do not support much of the syntax: No approach translating UML state machines into EHAs or PROMELA supports even half of the features. However, a surprise is that References [66, 117] are among the

best of the approaches, with 9 (respectively, 10)/17 features supported; this may come as a surprise, as their works mainly focus on timed properties. The run-to-completion step is properly encoded in Reference [117], which is not often the case in translation approaches.

4.4 Translation into Petri Nets

Petri nets [175] are bipartite graphs (with places and transitions and tokens evolving within places) and could be seen as an extension of automata; still, we consider them in a separate subsection, in part due to the large literature translating UML state machines into (variants of) Petri nets.

Various extensions of Petri nets were defined. In particular, **colored Petri nets (CPNs)** [106] are a special case of Petri nets in which the tokens are extended with attributes (types). This results in a clearer and more compact representation. Several approaches in the literature, notably, in the 2010s, translate UML state machines into (possibly colored) Petri nets. We review them in the following:

Pettit IV and Gomaa present in Reference [176] an approach that uses CPNs to model and validate the behavioral characteristics of concurrent object architectures modeled using UML. UML collaboration diagrams are considered in particular. The authors discuss how to map active/negative objects as well as message communications into CPNs. Synchronous as well as asynchronous communications are discussed in message communications. Though not specifically dealing with UML state machines, this work provides a (very first) general idea of transforming UML diagrams to Petri nets. Verification is carried out using Design/CPN,[5] which is an ancestor of CPNtools [211].

Baresi and Pezzè propose in Reference [25] another approach to formalize UML with high-level Petri nets, i.e., Petri nets whose places can be refined to represent composite places. Class diagrams, state diagrams, and interaction diagrams are considered. Customization rules are provided for each diagram. However, the authors do not provide details about those customization rules; instead, they illustrate the steps with the "hurried philosopher problem." The analysis and validation are also discussed, especially how to represent properties (such as absence of deadlocks or fairness) in UML, as well as how to translate them into Petri nets models. Instead of providing an automatic tool, the paper discusses how model checking can be conducted on various properties by querying the existing CASE tools (taking high-level Petri nets as input).

In Reference [181], Saldhana et al. propose an approach to formalize UML state machine diagrams using extensions of Petri nets. First, state machine diagrams are translated into flat state machines and then into **OPNs (object Petri nets)**; then, UML collaboration diagrams are used to derive a colored Petri net. The translation process is not very formal, but several examples illustrate it. No experiments are performed, but the model of a Spacecraft Control System is thoroughly discussed. This work is then extended to simulation in References [101, 140].

In Reference [158], Merseguer et al. translate state machines into generalized stochastic Petri nets (see, e.g., Reference [152]). Unfortunately, only "flat" state machines are considered, and therefore any hierarchical construct is disregarded, as well as pseudostates (except initial states). No automated translation is proposed. Then, in Reference [29], Bernardi et al. translate state machines (and sequence diagrams) to generalized stochastic Petri nets. The overall goal is to ensure consistency between the sequence diagrams and the statecharts. The translation of state machines itself relies on the translation proposed in Reference [158].

In References [202, 203], Trowitzsch and Zimmermann translate a subset of *timed* UML state machines into stochastic Petri nets. In Reference [202], the authors use stochastic Petri nets, which contain exponential transitions, making it more suitable to model time events. The

[5] https://homepages.inf.ed.ac.uk/wadler/realworld/designcpn.html.

approach covers a quite interesting subset of the UML state machine syntax, including time events.

In Reference [56], Choppy et al. propose an approach that formalizes UML state machines by translating them to colored Petri nets. They provide a detailed pseudo algorithm for the formalization procedure. They map simple states of UML state machine into Petri nets places and composite states of UML state machine into composite Petri nets places. Transitions in UML state machines are mapped to arcs in Petri nets, and corresponding triggering events are properly labeled. An extra place called *Events* is modeled with an "event place" in Petri nets, in which each event type is translated into a different color. Entry and exit actions of UML state machines are modeled with an arc in Petri nets that is labeled with the proper event type and ends in the event place. Though the mapping from UML state machines to high-level Petri nets is clearly expressed compared to Reference [25], a very limited subset of UML state machine features is supported: Only the very basic features such as simple state, composite state, transitions, triggers, and entry/exit actions are discussed. How to deal with more complex concurrent composite states and, notably, non-trivial forks or joins, is not discussed. A translation prototype[6] has been implemented, and verification is carried out using CPN-AMI [93].

André et al. propose in Reference [18] an approach different from the work by Reference [56] and support a larger subset of UML state machine features, including state hierarchy (i.e., composite states), internal/external transitions, entry/exit/do activities, history pseudostates, and so on. However, a limitation of that approach is that concurrency is left out: Hence, fork and join pseudostates, as well as communication between state machines via triggers is not considered. Verification is carried out using CPNtools [211].

Reference [17] extends Reference [18] by reintroducing the concurrency; hence, Reference [17] supports the syntactic elements considered in Reference [18], with the addition of fork and join pseudostates, orthogonal regions, as well as concurrent inter-level transitions. The run-to-completion semantics is also properly handled.[7]

In Reference [154], Meghzili et al. propose another translation from UML state machine diagrams to colored Petri nets. The subset of the syntax is rather limited (mostly orthogonal states, entry and exit behaviors). However, a major feature is that the model translation is formalized in Isabelle/HOL, which makes it a *verified* translation. The verified transformation is then performed with Scala. The authors extend their work in Reference [155], and a translation from BPMN (Business process) to CPNs is proposed to show the genericity of the approach.

In a parallel direction, Kumar et al. propose in Reference [124] safety analyses in terms of quantitative probabilistic hazard assessment; the authors convert (a quite restricted subset of) UML state machine diagrams into Petri Nets. Probabilistic model checking is then used; a reactor core isolation cooling system is used as a case study.

In Reference [147], Lyazidi and Mouline use Petri nets to model the behavior of UML state machines. Model checking is performed against safety properties as well as deadlock-freeness and liveness. A quite restricted subset of the UML syntax is considered, mostly simple states and fork/join pseudostates. The transformation is automated, and the resulting model is given in the TiNA syntax [31]; it remains unclear whether timing aspects are actually considered (for which the use of TiNA would make sense).

Discussion. Petri nets are used both in the academics and the industry (in particular, for modeling workflows). But they may be more difficult to understand for non-experts than UML. With

[6]That we were not able to find online.

[7]Note that some of the authors of this survey are involved in this series of works.

automatic translators from UML state machines to Petri nets, engineers can benefit from the rigorous verification power of existing Petri nets verification tools.

However, approaches translating UML state machines to Petri nets usually cover a small subset of UML state machine features: With the exception of Reference [17], no approach supports more than half of the UML syntax (see Table 2); in addition, and again with the exception of Reference [17], no Petri nets translation approach supports the notion of run-to-completion step. Even the support of the run-to-completion step in Reference [17] was not easy to manage and required a separate encoding.

This little syntactic support of approaches based on translations to Petri nets for formalizing UML state machines can be seen as a paradox for two main reasons. First, Petri nets are a graphical formalism that is not very distant from the graphical representation of UML state machines. Second, the UML formal specification mentions Petri nets as an analogy to both UML activity diagrams [170, p. 285] and state machine diagrams [170, p. 313].

Explanations for this little support may be that Petri nets (and especially their extensions) are maybe less popular than, e.g., Spin. In addition, the lack of very well established model-checking tools (in contrast to tools such as Uppaal or Spin) can be another explanation (CPNtools, although well-known for CPNs, has a far from intuitive user interface). Finally, Petri nets are a typically concurrent formalism, whereas UML state machines require a more global understanding: A local UML transition cannot be fired if some other transition in another region shall fire first (e.g., because it has a higher priority). This situation may be encoded into Petri nets, but usually using a rather cumbersome manner.

4.5 Translation into the Input Language of SMV and NuSMV

A first translation into SMV is given by Kwon [127]: A formal semantics for UML statecharts is defined in the form of rule-rewriting systems, and a translation approach is provided from the formalized semantics to the SMV model checker. No detailed implementation is discussed. This work, in fact, fits both into the translation approach and the operational semantics approach.

In Section 4.1, we reviewed the work by Compton et al. in Reference [59] that translates UML state machines into ASMs. In the same work, the authors actually then use SMV as the back-end model checker to automatically verify UML state machines. The work first translates UML state machines into ASMs, which has been discussed before in Section 4.1. Then, the SMV model checker is invoked to verify the SMV specification of a UML state machine.

Lam and Padget propose in Reference [129] a symbolic encoding of UML statecharts and invoke NuSMV [57] to perform the model checking. Their work adopts a three-step procedure and uses ϕ-calculus as an intermediate format for the translation. They have implemented the translator from UML statecharts to ϕ-calculus and claim that the implementation of a translator from ϕ-calculus to the input language of NuSMV (named SC2PiCal) was ongoing—although we did not find any later updates on this.

Beato et al. also provide in Reference [26] a translation from UML diagrams to the input language of the SMV model checker. Instead of focusing on just UML state machines, this work focuses on the collaborations of different UML diagrams such as class diagrams, state machine diagrams, and activity diagrams. Noticing that high-level model designers may be unfamiliar with property languages used by model checkers (such as LTL and CTL), the authors also provide some aid in the form of a versatile assistant to guide users with their property writing. This paper does not describe the detailed translation rules, but illustrates their translation procedure with an ATM machine example. Hence, only the features appearing in that example are shown as "√" in Table 2.

In Reference [74], Dubrovin and Junttila first provide a compact symbolic encoding for UML state machines. Its symbolic nature makes it suitable for an application to symbolic model

checking, such as **BDD-based (binary decision diagrams)** or SAT-based bounded model checking. The approach discusses event dispatching mechanisms, multi-object (asynchronous) communication, as well as choice pseudostates—which are often left out in other approaches. However, some commonly considered constructs, such as history pseudostates, are not included in their formalization. They perform a translation from the defined semantics to the input language of the NuSMV [57] symbolic model checker. The detailed translation steps are not discussed in the paper, but an implementation SMUML [2] has been provided, and some experiment results are reported in Reference [74]. The symbolic step encoding itself is discussed in Reference [75].

4.6 Translation into Process Algebras

Ng and Butler [166, 167] propose to translate UML state machines into CSP and utilize the FDR model checker to proceed with the model-checking procedure. The priority mechanism is not encoded. Thanks to the capabilities of FDR, they are able to perform not only model checking but also trace refinement. An extension of Reference [167] is then considered in Reference [213].

Zhang and Liu [217] provide an approach that translates UML state machines into CSP♯, an extension of the CSP language, which serves as the input modeling language of PAT [198]. Many aspects are considered; however, it is unclear whether the run-to-completion step is correctly encoded: The "atomic" construct used in the translation could ensure its correctness, but it does not seem to be used to encode a complete transition. An implementation of the translator was done, and experimental results of the verification of UML state machines using PAT are presented.

Hansen et al. [94] translate a subset of **executable UML (xUML)** [157] into the process algebraic specification language mCRL2 [3, 91]. Interestingly, they compare several definitions of the run-to-completion step and show that this has an impact on the verification of properties. In addition, the class diagram is translated together with the state machines. Symbolic verification is performed using LTSmin [114]. The xUML state machines subset considered in Reference [94] is close to that of UML 2.2. The supported syntax includes concurrency and hierarchy and event dispatching; history and final pseudostates are explicitly not supported, and fork/join do not seem to be either.

Then, Djaaboub et al. translate in Reference [70] UML state machines into flat state machines, and then ultimately into **LOTOS (Language Of Temporal Ordering Specification)** [38]. A graph grammar is proposed for the translation, and their approach uses the meta-modeling tool AToM. Few features only are considered, including composite states and entry and exit behaviors.

Finally, Jacobs and Simpson translate in Reference [102] a part of the activity diagram and state machine diagram syntax of the **Systems Modeling Language (SysML)**. SysML is close to UML (in fact, the syntax of SysML state machines can be seen as a subset of UML behavioral state machines), and this work is therefore worth mentioning. The UML state machine syntax supported by Reference [102] is very limited, but its main interest is that the translation is performed in conjunction with activity diagrams. The event queue seems well encoded. As it is not exactly addressing UML state machines, we do not add this work to Table 2.

Discussion. According to Table 2, approaches that translate UML state machines to CSP do not seem more inclusive than other approaches in terms of supported syntax, with the exception of Reference [217], which is the third-most-inclusive approach in the surveyed translation approaches. However, it is unclear whether the run-to-completion step is properly encoded. The proper encoding of the run-to-completion step requires some global view on the system, whereas CSP reasons in terms of parallel processes, which may be inherently incompatible with (or at least be quite inappropriate to model) the atomic, sequential nature of the run-to-completion step. Although the proper encoding of the run-to-completion step using CSP is certainly possible, we

believe that (similarly to Petri nets) it must be cumbersome, which explains its little support by the CSP-based translation approaches.

4.7 Translation to PVS, KIV, B, and Z

PVS. Traoré [201] and Aredo [20] proposed the same year in two independent works (though relatively close in spirit) to translate UML state machines into **PVS (Prototype Verification System)** [172]. Many non-trivial aspects of the syntax of UML are considered in these two works, such as shallow history pseudostates, submachines states, and even time (in Reference [201]).

KIV. In Reference [23], Balser et al. use interactive verification to formally prove properties of UML state machines. They use KIV [24]. The property language is a variant of **ITL (interval temporal logic)** [165]. The set of considered features is quite large, with composite states, internal transitions, inter-level transitions, forks and joins, variables, and a proper encoding of the run-to-completion step. A quite complex example (Reference [23, Figure 1]) is given. However, quite surprisingly, history pseudostates (both deep and shallow) are left out, and so are choice pseudostates, call and deferred events, as well as internal transitions.

B and extensions. In Reference [137], Ledang and Souquières translate UML state machine diagrams into B specifications [4]. The set of syntactic features is quite restricted, and no automation seems to be provided. A simple lift control system is used for exemplification.

In Reference [131], a subset of the UML-RSDS specifications [132] is translated into B. The article mostly considers class diagrams, and state machines are only mentioned without being formally considered in the translation. The focus is not mainly on state machines, and therefore this work is not integrated in Table 2. Note that UML-RSDS is now called AgileUML [133].[8]

Snook et al. [194] provide an approach to translate UML models (including class diagrams and state machine diagrams) into UML-B diagrams [192, 193], which incorporate Event-B method into UML diagrams. The idea of integrating UML with Event-B was introduced in Reference [192] and improved by introducing an action modeling language μ-B. A translator U2B was developed and later made an Eclipse plugin [193], which is also part of the Rodin platform [5]. Unfortunately, the translation in Reference [194] is not detailed enough and therefore not integrated into Table 2.

Refinement of systems through a combination of UML-B state machines and Event-B is also discussed in Reference [46].

Z. In Reference [115], Kim and Carrington propose a model transformation-based approach to transform UML state machine diagrams into Z specifications, but also a transformation from the latter formalism into the former. The authors formalize a UML metamodel using Object-Z and then propose the opposite formalization. Only basic syntactic aspects are considered in the translation.

In Reference [216], Zhan and Miao propose a formalization for UML state machines using Z (notably, extending Reference [160]). The model can then be translated to **"FREE" (flattened regular expressions)** models. The run-to-completion step is carefully encoded, but many syntactic aspects of the UML are missing. While the ultimate goal is testing, no automated translation seems to be available. Testing UML state machines is further considered in Reference [215].

In Reference [163], El Miloudi and Ettouhami propose an approach to translate UML state machine diagrams (in addition to class and sequence diagrams) into the Z notation [196]. An originality is multi-view modeling, with a consistency check with class and sequence diagrams. Few syntactic elements are considered in Reference [163], but signal events are considered. Overall,

[8] https://projects.eclipse.org/projects/modeling.agileuml.

the translation mechanism is described in a rather shallow manner, and no automatic translation engine seems to be available.

Discussion. According to Table 2, the two works translating state machines to PVS [20, 201] are among the most complete when it comes to translating UML state machines into a target formalism (and Reference [201] is actually the most complete work surveyed in the translation-based approaches, together with Reference [108]). They consider syntactic aspects that few other works support (submachine states, junctions, history pseudostates, run-to-completion step, and even time). This is surprising, as the UML version considered in their work is outdated (1.3) but also because these two works are among the two earliest.

4.8 Formalization Using Institutions

In Reference [120], Knapp et al. formalize "simple" (non-hierarchical) UML state machines using institutions. In Reference [118], Knapp and Mossakowski do not only extend the former work to hierarchy, but also add sequence diagrams and composite structure diagrams. The set of syntactic elements is rather limited, and no automated translation seems to be provided; however, a simple check using the **Distributed Ontology, Model and Specification Language (DOL)** is performed.

In Reference [179], UML state machines are first embedded into a logical framework (called "$\mathcal{M}_{\mathcal{D}}^{\downarrow}$"), which is then mapped to **CASL (Common Algebraic Specification Language)** [164]. As in Reference [120], only "simple" (flat) state machines are considered; the set of covered syntactic elements is not precisely given, but seems rather limited. Notably, communication between state machines (and the associated event pool) is not considered. However, an implementation of the translation is available, and the verification of a safety property on a simple counterexample is performed using the automated theorem prover SPASS [210].

4.9 Summary

4.9.1 Summary of Features. We summarize the surveyed translation-based approaches in Table 2, ordered by target formalism. We give from left to right the number of citations (see below), the UML version, the target formalism, and then we record the features considered by each work, viz., the orthogonal and submachine states, the fork/join, junction, choice, shallow history, deep history pseudostates, the entry/exit points, the entry/exit behaviors, the internal, inter-level and completion transitions, the run-to-completion step, the use of variables, the proper handling of deferred events, the handling of time, and the ability to have communicating state machines, i.e., several state machines communicating with each other, e.g., using synchronization. For space consideration, we remove the features that are commonly supported by all approaches, typically simple states, external transitions, initial pseudostate, and final states.

Evaluating the syntax support. The symbol "√" denotes the fact that a syntactic feature is supported, "×" means the feature is not supported, "○" means the featured is discussed in the paper but is not thoroughly solved. For example, for "conflict/priority," some works considered conflict among enabled transitions but did not discuss conflict due to deferred events. In this case, we regard the features to be partially supported. Some information could not be retrieved from some papers, in which case, we mark it by "?." At the bottom of the table, we count the approaches handling each feature (we count 1 when "√," 0 when "×," 0.5 when "○," and 0.25 when "?"); we hence derive the percentage of works handling that feature. The right-most column ("sum") of Table 2 also counts for each approach the number of supported features by counting the supported features with the aforementioned conventions (1 when "√," 0 when "×," 0.5 when "○," and 0.25 when "?"). This is certainly not an absolute way to compare approaches with each other (see threats

to validity below), but rather a way to quickly identify more complete approaches as opposed to approaches supporting very few syntactic features.

We can conclude from Table 2 that, in the translation-based approaches, the least-supported features are entry/exit points, submachine states, time, and deferred events. Internal transitions are discarded from most works formalizing UML, which may come as a surprise, as they do not seem to pose particular theoretical problems. In contrast, the most supported features are orthogonal states, entry/exit behaviors, and completion and inter-level transitions. But, quite disappointingly, with the exception of orthogonal states (68%) and entry/exit behaviors (52%), none of the UML state machines elements are supported by more than half of the surveyed works.

Evaluating the popularity. Finally, we also tabulate the number of citations of each work (given by Google Scholar, as of early April 2022) in Table 2. While this measure is not perfect, and while earlier works have obviously more been cited since their publication than more recent works, it gives an (approximate) measure of the popularity of each work. Only four of the surveyed works reached over 250 citations (with a maximum of 444): References [117, 135, 136, 185], all prior to 2002, and from only two groups of authors: Latella, Majzik, and Massink, on the one hand [135, 136], and Knapp and Merz, on the other hand [117, 185]. They also consider rather restricted subsets of the UML syntax. An explanation for this popularity may, however, come from the target language: Three of them target PROMELA, the input language of Spin, while the last one [117] is associated to a tool (HUGO/RT), the development of which is still active nowadays (see Table 7).

Threats to validity. We briefly discuss possible threats to validity regarding Table 2. A first potential issue is that knowing exactly the set of features supported by each work is not always easy. Most works did not provide an exact summary, leading us to have a detailed look at sometimes subtle semantic definitions, or even at the examples more or less formally detailed in each paper, to gather the set of supported syntactic features. This also explains a number of "○" and "?" cells in Table 2.

A second potential issue is the "sum" score: While we believe it is useful to quickly identify rather complete approaches, it should not be taken too literally; for example, from Table 2, Reference [66] has sum 10 while Reference [117] has sum 9, but we do not necessarily mean that Reference [66] is a "better" approach than Reference [117]. It solely means that Reference [66] supports one more feature: Both works [66, 117] are certainly more "complementary" to each other, rather than one strictly better than the other. In addition, note that we counted exactly "1" for each feature to compute the sum; this is debatable, as some features may be considered as more important than others. (And the relative importance of the syntactic features of UML state machines certainly goes beyond the scope of this survey.) Finally, counting 0.5 for "○" and 0.25 for "?" can again be seen as a somehow arbitrary choice. Note, however, that a different method (e.g., 0.25 for "○" and 0 for "?") would not change significantly the order of the respective sums.

4.9.2 Tool Support. We review the tool support offered by the translation-based approaches in Table 3. We give, from left to right, the UML version, the target formalism, the model-checking tool (if any), the kind of verification offered, and the tool responsible to perform the translation (if any). From Table 3, we see that, while many approaches offer a tool support, some do not—which can be seen as a somehow debatable choice, considering the highly applied motivation of this field of research. In addition, a few approaches aim at translating UML into some existing model checkers but do not provide any automated software to do so. This is, again, a very debatable choice.

We will review all tools from a user point of view in Section 6.

4.9.3 Discussion. Clearly, a first disappointing conclusion is that most works consider a quite restricted subset of the syntax of state machines. No work considers more than 11/19 features,

Table 2. UML State Machine Features Supported by Each Translation Approach

Approach	nb cit.	UML v	Target	States				Pseudostates				Entry/exit	Transitions			RTC	Variables	Deferred	Time	Multiple	Sum
				ortho	subm	fk/jn	junct.	choice	sH	dH	en/ex	behv	intern	interl	compl			events		charts	
Börger *et al.* (2000) [48]	146	1.3	ASM	○	○	○	○	×	√	√	×	√	√	○	√	√	×	√	×	×	9.5
Compton *et al.* (2000) [59]	55	1.3	ASM	√	×	×	×	×	×	×	×	√	×	√	√	×	×	?	×	×	4.25
Jürjens (2002) [113]	56	1.4	ASM	√	×	×	×	×	×	×	×	√	√	×	○	×	×	×	×	√	4.5
Börger *et al.* (2003) [52]	16	1.3	ASM	√	×	×	×	×	√	√	×	√	?	?	√	?	?	√	×	×	6.0
Jin *et al.* (2004) [108]	34	1.5	OMA	√	×	√	√	×	√	√	×	√	√	√	√	√	×	√	×	×	11.0
Kuske (2001) [125]	130	1.3	graph transformation	√	×	?	×	×	×	×	×	(√)	(√)	×	×	√	×	?	×	×	4.5
Kong *et al.* (2009) [123]	51	2.0	graph transformation	√	×	√	√	√	√	√	×	×	×	√	√	?	√	×	×	√	10.25
Latella *et al.* (1999) [136]	317	<1.3	EHA	√	×	√	×	×	×	×	×	×	×	√	×	×	×	×	×	×	3.0
Gnesi *et al.* (1999) [89]	112	<1.3	EHA/LTS	√	×	√	×	×	×	×	×	×	×	√	×	×	×	×	×	×	3.0
Latella *et al.* (1999) [135]	344	1.1	EHA/PROMELA	√	×	√	×	×	×	×	×	×	×	√	×	×	×	×	×	×	3.0
Schäfer *et al.* (2001) [185]	313	1.4	PROMELA	√	×	√	×	√	×	×	×	√	×	√	√	√	×	×	×	×	7.0
Jussila *et al.* (2006) [111]	89	1.4	PROMELA	√	×	×	×	√	×	×	×	×	×	×	√	×	√	√	×	√	6.0
Carlsson *et al.* (2009) [55]	7	2.0 (?)	PROMELA	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	√	√	2.0
Knapp *et al.* (2002) [117]	261	1.4	TA	√	×	√	√	×	×	×	×	√	×	×	√	√	√	×	√	√	9.0
David *et al.* (2002) [66]	163	1.4 (?)	TA	√	√	√	√	×	√	×	×	×	×	√	√	×	√	×	√	√	10.0
Ober *et al.* (2006) [169]	138	2.0	communicating ext. TA	?	×	×	×	√	×	×	×	?	×	×	×	√	√	×	√	?	4.75
Pettit IV *et al.* (2000) [176]	63	≤1.3	CPN	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	√	1.0
Baresi *et al.* (2001) [25]	78	≤1.3	HLPN	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	0.0
Saldhana *et al.* (2001) [181]	84	1.2?	objet Petri nets	√	×	√	×	×	×	×	×	√	√	×	×	×	×	√	×	√	6.0
Merseguer *et al.* (2002) [158]	116	1.4	GSPNs	×	×	×	×	×	×	×	×	√	√	×	√	?	×	√	×	×	4.25
Trowitzsch *et al.* (2005) [202]	22	2.0	SPN	○	×	√	√	√	×	×	×	√	×	×	×	×	√	×	√	×	6.5
Choppy *et al.* (2011) [56]	59	2.2 (?)	HCPN	○	×	×	√	√	×	×	×	√	×	×	×	×	×	×	×	√	4.5
André *et al.* (2012) [18]	23	2.2	CPN	×	×	×	×	×	√	○	×	√	√	√	√	×	√	×	×	×	6.5
André *et al.* (2016) [17]	12	2.5 beta 1	CPN	√	○	√	×	×	√	○	×	√	√	√	√	√	√	×	×	×	10.0
Kumar *et al.* (2017) [124]	15	≤ 2.5	Petri nets	√	×	√	×	×	×	×	×	?	×	×	×	×	×	×	×	×	2.25
Meghzili *et al.* (2017) [154]	16	≤ 2.5	CPN	√	×	×	×	×	×	×	×	√	×	×	×	×	?	×	×	×	2.25
Lyazidi *et al.* (2019) [147]	3	≤ 2.5.1	Petri nets	×	×	√	√	√	×	×	×	×	×	×	×	×	×	×	?	×	3.25
Kwon (2000) [127]	81	1.3	SMV	√	×	×	×	×	√	×	×	×	√	×	×	×	×	×	×	×	3.0
Lam *et al.* (2004) [129]	42	≤1.5	NuSMV	×	×	×	×	×	×	×	×	×	?	?	×	×	×	×	×	√	1.0
Beato *et al.* (2005) [26]	68	<1.3	SMV	√	×	×	×	√	×	×	×	√	×	√	×	×	√	×	×	?	5.25
Dubrovin *et al.* (2008) [74]	64	≤2.1.1	NuSMV	√	×	×	×	√	×	×	×	×	×	√	√	√	√	√	×	√	8.0
Ng *et al.* (2002) [166]	37	1.3	CSP	×	×	×	×	√	×	×	×	×	×	×	×	×	√	×	×	√	3.0
Ng *et al.* (2003) [167]	82	1.4	CSP	√	×	×	×	√	×	×	×	√	×	○	√	×	√	×	×	√	6.5
Zhang and Liu (2010) [217]	63	2.2	CSP♯	√	√	√	×	√	√	×	√	√	×	√	√	?	√	×	×	×	10.25
Hansen *et al.* (2010) [94]	59	2.2	mCRL2	√	×	×	×	×	×	×	×	×	×	×	√	√	×	×	×	×	3.0
Djaaboub *et al.* (2015) [70]	2	2.0	LOTOS	√	×	×	×	×	×	×	×	√	×	√	×	×	×	×	×	×	3.0
Aredo (2000) [20]	24	1.3	PVS	√	√	√	√	√	√	√	×	√	×	×	√	×	√	×	×	×	10.0
Traoré (2000) [201]	47	1.3	PVS	√	×	√	√	√	√	√	×	×	×	√	√	√	√	×	√	×	11.0
Kim *et al.* (2002) [115]	24	1.3	Z	√	×	×	×	×	√	√	×	√	√	×	×	×	×	√	×	×	6.0
Ledang *et al.* (2002) [137]	88	1.2?	B	×	×	×	×	×	×	×	×	√	×	√	×	?	×	√	×	√	4.25
Balser *et al.* (2004) [23]	68	1.5	KIV	√	×	√	√	×	×	×	×	√	×	√	√	√	√	×	×	×	8.0
Zhan *et al.* (2004) [216]	14	2.0	Z	√	?	×	×	×	×	×	×	×	√	√	√	√	√	×	×	×	6.25
El Miloudi *et al.* (2015) [163]	6	2.4.1	Z	×	×	?	×	×	×	×	×	?	×	×	×	×	×	×	×	×	0.5
Knapp *et al.* (2017) [118]	8	2.5	interactions	×	?	×	√	×	×	×	×	×	?	×	√	?	√	×	×	×	3.75
Rosenberger *et al.* (2021) [179]	1	2.5.1	interactions	×	×	×	×	×	×	×	×	×	×	×	×	?	×	×	×	×	0.25
Features supported	-	-	-	30.75	4.5	18.0	11.5	14.0	12.0	8.0	1.0	23.75	10.25	20.0	21.5	13.25	18.5	9.5	6.25	14.5	17
%	-	-	-	68 %	10 %	40 %	25 %	31 %	26 %	17 %	2 %	52 %	22 %	44 %	47 %	29 %	41 %	21 %	13 %	32 %	100 %

which gives 58%, hence, not much more than one half. Conversely, with the exception of orthogonal composite states (68%) and entry/exit behaviors (52%), no feature is supported by more than 50% of the approaches. The strong support of a particular feature does not seem to be specifically correlated with neither the fact that it is older (introduced in earlier versions of the UML), nor the fact that it can be substituted easily by some other features.

The least-supported feature is the entry/exit points (2%, a single approach), which is highly surprising, as these entry/exit points do not bring any difficulty to translate to other languages in a relatively easy fashion; in addition, they can turn really useful from a modeling perspective. More worrying is the fact that the run-to-completion step (a central notion in UML) is not correctly supported by most approaches (only 29%).

Another surprising point is that more recent approaches are not necessarily the best; in fact, the most complete approaches (i.e., References [108, 201]) are among the oldest.

Finally, two recent works [154, 155] are rather shallow in terms of the syntax considered but are a verified translation, which brings some additional guarantees into the confidence one can have in the result.

4.9.4 Common Advantages and Drawbacks of Translation-based Approaches. The translation approaches aim at utilizing the automatic verification ability of different model checkers. However, translation-based approaches may suffer from the following defects:

(1) Due to the semantic gaps, it may be hard to translate some syntactic features of UML state machines, introducing sometimes additional but undesired behaviors. For example,

Table 3. Translation Approach: Tool Support

Approach	UML v	Target	Verification tool	Verification	Translation tool
Börger *et al.* (2000) [48]	1.3	ASM	×	×	×
Compton *et al.* (2000) [59]	1.3	ASM	SMV	model checking	veriUML
Jürjens (2002) [113]	1.4	ASM	×	×	×
Börger *et al.* (2003) [52]	1.3	ASM	×	×	×
Jin *et al.* (2004) [108]	1.5	OMA	×	×	×
Kuske (2001) [125]	1.3	graph transformation	×	×	×
Kong *et al.* (2009) [123]	2.0	graph transformation	×	×	×
Latella *et al.* (1999) [136]	<1.3	EHA	×	×	×
Gnesi *et al.* (1999) [89]	<1.3	EHA/LTS	JACK	model checking with ACTL	×
Latella *et al.* (1999) [135]	1.1	EHA/PROMELA	Spin	model checking with ACTL	√
Schäfer *et al.* (2001) [185]	1.4	PROMELA	Spin	model checking	HUGO
Jussila *et al.* (2006) [111]	1.4	PROMELA	Spin	model checking	PROCO
Carlsson *et al.* (2009) [55]	2.0 (?)	PROMELA	Spin	model checking	RSARTE
Knapp *et al.* (2002) [117]	1.4	TA	UPPAAL	subset of TCTL	HUGO/RT
David *et al.* (2002) [66]	1.4 (?)	TA	UPPAAL	subset of TCTL	?
Ober *et al.* (2006) [169]	2.0	communicating ext. TA	IF	simulation and verification	√
Pettit IV *et al.* (2000) [176]	≤1.3	CPN	DesignCPN	deadlock / statistical analysis	×
Baresi *et al.* (2001) [25]	≤1.3	HLPN	×	×	×
Saldhana *et al.* (2001) [181]	1.2?	objet Petri nets	×	×	×
Merseguer *et al.* (2002) [158]	1.4	GSPNs	×	×	×
Trowitzsch *et al.* (2005) [202]	2.0	SPN	×	×	×
Choppy *et al.* (2011) [56]	2.2 (?)	HCPN	CPN-AMI	LTL/CTL	√
André *et al.* (2012) [18]	2.2	CPN	CPNtools	model checking	×
André *et al.* (2016) [17]	2.5 beta 1	CPN	CPNtools	model checking	prototype
Kumar *et al.* (2017) [124]	≤ 2.5	Petri nets	×	probabilistic model checking	×
Meghzili *et al.* (2017) [154]	≤ 2.5	CPN	none (CPNtools envisioned)	model checking	Scala
Lyazidi *et al.* (2019) [147]	≤ 2.5.1	Petri nets	TiNA	model checking	prototype
Kwon (2000) [127]	1.3	SMV	SMV	model checking	?
Lam *et al.* (2004) [129]	≤1.5	NuSMV	NuSMV	model checking	SC2PiCal (?)
Beato *et al.* (2005) [26]	<1.3	SMV	SMV	model checking	TABU
Dubrovin *et al.* (2008) [74]	≤2.1.1	NuSMV	NuSMV	model checking	√
Ng *et al.* (2002) [166]	1.3	CSP	FDR	model checking / refinement	√
Ng *et al.* (2003) [167]	1.4	CSP	FDR	model checking / refinement	√
Zhang and Liu (2010) [217]	2.2	CSP♯	PAT	model checking	?
Hansen *et al.* (2010) [94]	2.2	mCRL2	LTSmin	model checking	×
Djaaboub *et al.* (2015) [70]	2.0	LOTOS	×	×	graph grammar
Aredo (2000) [20]	1.3	PVS	PVS	model checking / theorem proving	×
Traoré (2000) [201]	1.3	PVS	PVS	model checking / theorem proving	PrUDE
Kim *et al.* (2002) [115]	1.3	Z	×	×	×
Ledang *et al.* (2002) [137]	1.2?	B	B	theorem proving	×
Balser *et al.* (2004) [23]	1.5	KIV	KIV	interactive verification	×
Zhan *et al.* (2004) [216]	2.0	Z	×	×	×
El Miloudi *et al.* (2015) [163]	2.4.1	Z	Z	consistency checks	×
Knapp *et al.* (2017) [118]	2.5	interactions	DOL	consistency checks	×
Rosenberger *et al.* (2021) [179]	2.5.1	interactions	CASL / HETS / SPASS	×	√

in Reference [217], extra events have to be added to each process to model exit behaviors of orthogonal composite states.

(2) For the verification, translation approaches heavily depend on the tool support of the target formal languages. Furthermore, the additional behaviors introduced during the translation may significantly slow down the verification. These additional behaviors may come in the form of additional steps required by the translation; for example, in automata-based formalisms, some (internal) automata transitions may be added, which do not modify the overall UML state machine execution, but still add some extra transitions to be explored by the model checker; this may blow up in case of different automata in parallel. (Similar behaviors can happen in other formalisms such as Petri nets, process algebras, etc.) In addition, optimizations and reduction techniques (such as partial order reduction) may not apply to preserve the semantics of the original model.

(3) Last, when a counterexample is found by the verification tool, it is hard to map it to the original state machine execution, especially when state space reduction techniques are used. One of the exceptions is References [59, 190], where a counterexample can be exhibited in a visual fashion.

Note that a direct implementation based on an operational semantics may solve some of the aforementioned issues.

5 APPROACHES PROVIDING OPERATIONAL SEMANTICS FOR UML STATE MACHINES

Different from the translation-based approaches, the second kind of approach directly provides an operational semantics to UML state machines, usually by defining inference rules. Various verification techniques can then be conducted based on the operational semantics. The benefits of this kind of approach are:

(1) they do not rely on the target formal languages, thus no redundancies are introduced; and
(2) the semantic steps defined in the operational semantics directly coincide with the UML state machine semantic step, i.e., the run-to-completion step.

Semantic approaches are harder to classify than translation-based approaches. Therefore, we follow a mostly chronological approach based on the UML version: We first describe UML 1.x approaches (Section 5.1), then the UML 2.0 approaches by Schönborn et al. and extensions (Section 5.2), then one approach for UML 2.1 (Section 5.3), and finally some approaches for UML 2.4 (Section 5.4). Finally, we summarize the approaches in Section 5.5.

5.1 Operational Semantics for UML 1.x

Approaches using EHAs. Hierarchical Automata require a strict hierarchical structure. The existence of inter-level transitions and local transitions breaks the hierarchical structure. **Extended Hierarchical Automata (EHAs)** extend Hierarchical Automata to deal with inter-level transitions i.e., an inter-level transition that crosses multiple states will be assigned to the outermost Sequence Automaton. Although approaches using EHAs as an intermediate representation do not "directly" provide the operational semantics, EHAs still resemble UML state machines in the hierarchical structure, and EHAs are equipped with an operational semantics. For this reason, we consider this kind of approach as directly providing operational semantics.

Latella et al. [136] are among the pioneers who began to focus on formalizing UML statecharts (instead of other variants of statecharts) semantics. They use a slightly modified variant of EHAs as an intermediate model and map the UML-statecharts into an EHA. The hierarchical structure of UML statecharts and EHAs makes the translation structured and intuitive. Then they define the operational semantics for EHAs using Kripke structures.

A following work by Gnesi et al. [90] extends Reference [136] to include multicharts, i.e., multiple UML state machines communicating asynchronously, using a non-deterministic choice of event dispatch between the various components. The work also discusses how to incorporate the semantics into the model-checking tool JACK [39, 89]. This approach covers a quite restricted subset of UML state machine structures: No pseudostates (except the initial pseudostate) are considered, neither entry/exit/do behaviors nor deferred events are considered, and the triggering events are restricted to signal and call events without parameters. Even though the ultimate goal is to use JACK for the automated verification, the authors do not discuss the implementation of the translation, and it is not clear whether it has been done. In fact, the authors themselves explicitly focus on the design issues rather than on the implementation ones.

Dong et al. [72] further extend EHAs to support more features such as entry/exit behaviors or parameters in behaviors and provide a formal semantics for a subset of UML statecharts based on this EHA model. The authors discuss the findings on the cost of solving conflicts introduced by concurrent composite states and the importance of modeling with multiple objects instead of modeling them with concurrent regions within one UML state machine. They consider the

non-determinism caused by multiple concurrent state machines, which was not captured by Reference [136]. This work is extended in Reference [209]. Note that slicing has also been discussed in References [85, 159] for statecharts and in References [14, 16] for **extended finite state machines (EFSMs)**.

Other approaches for UML 1.x. Von der Beek [207] also formalizes a partial set of UML statecharts, partially based on the work proposed in Reference [136]. But it supports some more features compared to Reference [136], such as history mechanisms, entry and exit behaviors. The syntax used in this work is called a UML-statechart term, which is inductively defined on three kinds of terms, viz., basic term, or-term, and and-term. All of them contain basic information about a state such as a unique ID, entry and exit behavior, and sub-terms (for or-term and and-term) that contain the hierarchical information of a UML state machine. UML-statechart terms basically represent static information about UML statechart vertices. Inter-level transitions are captured by explicitly specifying source restrictions and target determinators in an or-term; this notation follows the idea of Reference [136].

In Reference [127], Kwon proposes another approach using Kripke structures and aims at model checking UML statecharts. Similarly to Reference [207], Kwon uses terms as the syntax domain of UML statecharts, which represent state hierarchy in the form of subterms as a field in a term. But Reference [127] uses conditional rewriting rules to represent the transition relation in a UML statechart (while Reference [207] explicitly defined five structural operational semantics rules). Then the semantics of UML statecharts is defined as a Kripke Structure. Reference [127] also provides a translation from the defined Kripke structure to the input language of the SMV model checker, which we have discussed in Section 4.5.

Eshuis and Wieringa [76] provide an operational semantics for UML statecharts using LTS. This work focuses more on the communication and timing aspect of UML statecharts. It also considers object construction and destruction, which is not always considered by the other approaches. Model checking is performed using the ATCTL logics and the Kronos model checker [214], and experiments are tabulated in Reference [105].

Lilius and Paltor [141, 142] provide an abstract syntax and semantics for a subset of UML state machines. They use terms as syntax model and consider most features of UML state machines. Although it does not define a clear semantic model, their work formalizes the run-to-completion step semantics into an algorithm. The algorithm is given at an abstract level, and many concepts such as history pseudostates and completion events are described in a rather informal manner. But the procedure of the run-to-completion step is properly described. Some features such as join, fork, junction, choice vertices are unspecified and, instead, it is claimed that these pseudostates can be replaced with extra transitions.

Reggio et al. [178] provide UML state machines with a formal semantics given as an LTS. This work considers an early version (1.3) of UML specifications and discusses some inconsistencies and ambiguities in the specification. The work does not provide a clear syntax model, and UML state machines are not represented formally. But it does discuss in detail the event dispatching, as well as the way the events are inserted into the queue: Notably, the authors assume that "it is better to have a mechanism ensuring that when two events are received in some order they will be dispatched in the same order," and therefore the event queue is modeled as a multiset of events—and neither as a queue nor as a set.

Damm et al. [63] provide a formal semantics for a kernel set of UML to model real-time applications, including static and dynamic aspects of the UML models. The formalization contains two steps. First, **real-time UML (rtUML)** is represented in terms of the introduced concept of **"kernel subset of real-time UML" (krtUML)**. Second, krtUML is equipped with a formal semantics.

This approach provides a self-defined action language, which supports object creation/destruction, assignment and operation calls. The semantics is given in terms of symbolic transition systems, a concept originally introduced in Reference [151] as "synchronous transition systems," and that can be seen as an extension of transition systems with first-order logic predicates. UML state machines are just a component of krtUML, and therefore state machines are not the core of the formalization proposed in Reference [63]. Still, this work provides a good reference for communications between different objects, such as event dispatching and handling. Additional details are tabulated in Reference [65].

5.2 Operational Semantics for UML 2.0

An almost complete operational semantics for UML 2.0 Schönborn [186] provides in his *Diplomarbeit* (Master thesis) a very comprehensive analysis about UML 2.0 behavioral state machines, including discussions about detailed semantics of each feature and an exhibition of numerous ambiguities in the UML specification. This approach covers almost all features of UML 2.0 state machines, except for choice, termination pseudostates, and completion events. In addition, junction pseudostates are considered as syntactic sugar and are said to be easily represented by separate transitions. More precisely, Schönborn argues in Reference [186] that junction pseudostates "are used as a shorthand notation for collections of transitions" and that "submachine states (and therefore also entry and exit pseudostates) can be compiled away" (the UML specification explicitly mentions that a submachine state is "semantically equivalent to a composite State" [170, p. 311]). Even better, not only most of the syntax is considered in this work, but the author even discards some restrictions from the UML specification, claiming that his semantics is still valid in the absence of these restrictions; put differently, the syntax considered in Reference [186] can be seen as *larger* than the official specification.

In a first part, a formal syntax for UML state machines is defined (made of a 10-tuple to encode states, regions, substates, behaviors, transitions, etc.).

In a second part, the formal semantics is introduced. Arguing that "flattening" the hierarchical structure may lead to an unnecessary state explosion, the semantics is presented in a hierarchical manner whenever possible. Many auxiliary functions are defined to capture the execution of a run-to-completion step, such as collecting all actions generated during transition execution and putting them in the event pool. Priorities are handled in a particularly precise manner.

This work not only can be considered as a very detailed discussion about the semantics of UML 2.0, but it also contributes to the analysis of *ambiguities* in the UML 2.0 specification (see Reference [82], discussed below).

In Reference [80], a formalization very similar to Reference [186] is given. The relationship between both works is unclear: One author is coauthor of both works, and Reference [186] is a Diplomarbeit, while Reference [80] is a technical report; none of these two works (both dating from 2005) cite the other one. In Reference [80], again, most syntactic aspects are formalized: final pseudostates, composite states, deferred events, conflicts, priorities, both shallow and deep history pseudostates, entry/exit behaviors, completion, internal transitions, join/fork, and run-to-completion step. As in Reference [186], several ambiguities in the UML specification are exhibited. The selection mechanism of events is not considered in the paper; neither is the execution of actions. No tool implementation is mentioned.

Extensions and other works In a separate paper [82], Schönborn and additional co-authors discuss 29 new "unclarities" in the UML 2.0 state machine specification. They can consist of ambiguities, inconsistencies, or unnecessarily strong restrictions; these unclarities are clearly linked to Schönborn's Diplomarbeit [186], where he already had spotted such issues, and discussed unnecessarily strong restrictions (actually lifted in his formalization). The work in Reference [82] is clearly not

a formalization of UML state machines but can help the community to better formalize the specification. (Also note that a 30th ambiguity is pointed out by Reference [81].)

Fecher and Schönborn use in Reference [81] "core state machines" as the semantic domain for UML state machines. A core state machine is a 7-tuple including a set of states (including region and parent relations), a set of do actions, a set of deferred events, a set of transitions, an initial state, a set of variables, and an initial variable assignment. History is explicitly described by a mapping from a region to its direct substate. The work first formalizes both syntax and semantics of the core state machine. This article considers more UML state machine features. Although this approach is one of the most complete approaches in terms of syntax considered, it suffers from some limitations. The run-to-completion step of a UML state machine is not properly defined. The transformation steps from a UML state machine to a core state machine are provided, but the steps are not formally defined; instead, only natural language descriptions with example illustrations are given. Moreover, the translation is complex, since a lot of auxiliary vertexes need to be added, such as enter/exit vertices. This is actually obvious in the article's own figures, which are barely readable (see, e.g., Reference [81, p. 258]). These limitations may make it difficult for automatic tool development—which indeed does not seem to have been done.

Two other approaches were considered by this group of authors. First, in Reference [79], Fecher et al. are specifically interested in the compositional aspect: They define a compositional operational semantics for "flat" UML state machines (with only simple transitions guarded by expressions on variables).

Second, in Reference [78], Fecher et al. define a semantics to model persistent nondeterminism, which can model faulty systems. The paper is specifically interested in refinement and uses so-called "μ-automata."

Finally, Lano and Clark propose in Reference [130] an axiomatic semantics for a subset of the syntax of UML *protocol* state machines, based on Reference [82]. Although we are here interested in *behavioral* state machines, the authors give enough hints so its formalization can be directly applied to behavioral state machines—basically without entry/exit/do behaviors. The supported syntax is reasonably large, including composite states, deferred events, and history states; however, junction, choice, internal transitions, and so on., are left out. They use the B formalism [4] as a backend to perform formal verification.

5.3 Operational Semantics for UML 2.1

In Reference [187], Seifert proposes a formal semantics for UML state machines, following as much as possible the (informal) semantics and inspired by existing works [23, 135, 141]. The full translation is tabulated in Reference [188]. While this work ultimately aims at generating test cases (which goes beyond the scope of this survey), the formalization of the UML state machine semantics is sufficiently interesting to be included. The set of syntactic elements covered by the formalization is explicitly stated and includes, notably, composite states and a careful handling of the run-to-completion step; however, more basic features (entry and exit behaviors, forks, joins, ...) are left out. A subset of the Java programming language is used to express guard and action ("behaviors") expressions. The actual test generation is made using the Teager tool suite [182, 183].

5.4 Operational Semantics for UML 2.4

Liu et al. [144] provide a formal operational semantics for UML state machines using LTS.[9] The approach covers all the features of UML state machines except for time events. The approach also considers asynchronous/synchronous communications between objects.

[9]Note that the authors of this survey were involved in Reference [144].

Table 4. UML State Machine Features Supported by Each Semantic Approach

Approach	nb cit.	UML v	States		Pseudostates					Entry/exit		Transitions			RTC	Variables	Deferred events	Time	Multiple charts	Sum
			ortho	subm	fk/jn	junct.	choice	sH	dH	en/ex	behv	intern	interl	compl						
Lilius *et al.* (1999) [142]	116	1.3	√	×	√	√	×	√	√	×	√	√	×	√	√	○	√	√	×	11.5
Reggio *et al.* (2000) [178]	108	1.3	×	×	×	√	×	×	×	×	×	×	√	×	×	×	×	×	×	2.0
Eshuis *et al.* (2000) [76]	81	1.3	√	×	×	×	×	×	×	×	√	×	×	√	√	×	×	√	×	5.0
Dong *et al.* (2001) [72]	52	1.1	√	×	×	×	×	×	×	×	√	×	√	√	√	√	×	×	√	7.0
von der Beeck (2002) [207]	136	1.4	√	×	×	×	×	√	√	×	√	×	√	×	×	×	×	×	×	5.0
Gnesi *et al.* (2002) [90]	59	1.3	√	√	√	×	×	×	×	×	×	×	√	×	×	×	×	×	√	5.0
Damm *et al.* (2002) [63]	89	1.4	×	×	×	×	×	×	×	×	×	×	×	√	√	√	×	√	√	5.0
Schönborn (2005) [186]	9	2.0	√	(√)	√	(√)	×	√	√	(√)	√	√	√	×	√	×	√	×	×	12.0
Fecher *et al.* (2005) [80]	15	2.0	√	(√)	√	(√)	×	√	√	(√)	√	√	√	√	√	×	√	×	×	13.0
Fecher *et al.* (2006) [79]	7	2.0	×	×	×	×	×	×	×	×	×	×	×	×	×	√	×	×	√	2.0
Fecher *et al.* (2006) [81]	38	2.0	√	√	√	√	√	√	√	√	×	√	√	√	○	√	√	√	×	14.5
Lano *et al.* (2007) [130]	21	2.0	√	×	√	×	×	√	√	×	×	×	√	√	?	√	√	?	×	8.5
Seifert (2008) [188]	5	2.1.1	√	?	×	×	×	×	×	×	×	√	√	√	√	√	×	×	√	7.25
Fecher *et al.* (2009) [78]	7	2.0	×	×	×	×	√	×	×	×	×	×	×	×	×	√	×	×	√	3.0
Liu *et al.* (2013) [144]	54	2.4.1	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√	×	√	16.0
Besnard *et al.* (2021) [36]	0	2.5.1	×	×	×	×	√	×	×	×	√	×	×	×	√	√	√	×	×	5.0
Features supported	-	-	11.0	5.25	7.0	6.0	4.0	7.0	7.0	4.0	8.0	6.0	10.0	9.0	9.75	9.5	7.0	4.25	7.0	17
%	-	-	68 %	32 %	43 %	37 %	25 %	43 %	43 %	25 %	50 %	37 %	62 %	56 %	60 %	59 %	43 %	26 %	43 %	100 %

More recently, in Reference [36], Besnard et al. propose an approach to model not only systems, but also properties, in a unified UML framework. Their approach uses a "Semantic Transition Relation" interface, presented with a formal syntax strongly inspired by the Lean theorem prover [67]. The presentation is not detailed enough to get a full idea of what syntactic elements are considered; but the approach is particularly interesting due to its unified view. In addition, not only verification, but also online monitoring, can be performed. These algorithms are implemented in the EMI framework [32–35].

5.5 Summary

5.5.1 Summary of Features. We summarize the surveyed semantic-based approaches in Table 4. We use the same conventions as in Table 2. In addition, "(√)" denotes an indirect handling, e.g., submachine states are not supported in Reference [186], but the author reminds that they can be encoded using composite states. Also, see the threats to validity discussed in Section 4.9.1 concerning our method to compute the "sum" value for each work.

First, as in Table 2, it is clear from Table 4 that most approaches do not take into consideration many syntactic elements of UML state machines. Similarly, few such elements are supported by many approaches—with the exception of orthogonal states and inter-level transitions. In fact, only five approaches support more than half of the syntactic elements [80, 81, 142, 144, 186], with two of them [81, 144] supporting a very large majority of elements.

Overall, the semantic-based approaches support in average more elements than the translation-based approaches. This can be seen as a paradox: Indeed, semantic-based approaches are often theoretical, and it can be natural to not consider all elements if the missing elements can be encoded themselves into supported elements (this could be the case for, e.g., submachine states, junctions, and even history pseudostates). In contrast, translation-based approaches missing these elements have no other choice but translating them into supported elements, which makes the approach incomplete.

5.5.2 Tool Support. We review the tool support offered by the translation-based approaches in Table 5. The mostly theoretical nature of the semantic-based approaches is confirmed by Table 5, as most approaches provide strictly no tool support, neither a parser from an existing formalism for UML state machines, nor a model checker supporting the chosen semantics. Sadly, out of the two most complete approaches (References [81, 144]), only the latter provides a tool support. As

Table 5. Semantic Approach: Tool Support

Approach	UML v	Verification tool	Verification	Translation tool
Lilius *et al.* (1999) [142]	1.3	vUML	model checking	?
Reggio *et al.* (2000) [178]	1.3	×	×	×
Eshuis *et al.* (2000) [76]	1.3	×	×	×
Dong *et al.* (2001) [72]	1.1	×	×	×
von der Beeck (2002) [207]	1.4	×	LTL model checking	×
Gnesi *et al.* (2002) [90]	1.3	JACK	ACTL model checking	×
Damm *et al.* (2002) [63]	1.4	×	×	×
Schönborn (2005) [186]	2.0	×	×	×
Fecher *et al.* (2005) [80]	2.0	×	×	×
Fecher *et al.* (2006) [79]	2.0	×	×	×
Fecher *et al.* (2006) [81]	2.0	×	×	×
Lano *et al.* (2007) [130]	2.0	B	consistency	×
Seifert (2008) [188]	2.1.1	Teager	test generation	√
Fecher *et al.* (2009) [78]	2.0	×	×	×
Liu *et al.* (2013) [144]	2.4.1	USMMC	model checking	√
Besnard *et al.* (2021) [36]	2.5.1	EMI-UML	MC and monitoring	√

mentioned earlier, we suspect that the quite complex formalization of Reference [81] made it rather delicate to allow for a practical implementation.

6 TOOL SUPPORT

In this section, we discuss tool support for verifying UML state machines. There are both commercial and academic tool supports for UML modeling. Commercial tools include Eclipse Papyrus,[10] IAR Visual State,[11] IBM Rhapsody,[12] Microsoft Visual Modeler, or Yakindu.[13]

Also, some open-source tools take as input UML state machines, notably, PlantUML[14] and VUML[15]; however, these two tools are purely syntactic, and no analysis (simulation, verification) is possible. In addition, we noticed a number of shortcomings in PlantUML, notably, with orthogonal composite states (impossibility to draw "cross-border" transitions originating from or targeting a state belonging to a region of an orthogonal composite state; or define an entry point on an orthogonal composite state); in other words, even such a purely syntactic tool does not support the full UML specification. In addition, Umple [139] allows some automated code generation from UML state machines to a number of languages (including Java).

To the best of our knowledge, current non-academic commercial tools only support the design/graphical editing of UML models or perform some limited "verification," but without any publicly available academic foundation—hence, we discard them in our survey.

Some academic prototype tools were developed based on either the translation or the semantic approaches and aim at automatically verifying UML state machines. We survey these tools in Section 6.1 and draw comparisons and conclusions in Section 6.2.

6.1 Surveying Tools for Verifying UML State Machines

vUML. vUML [143] aims at automatically verifying UML model behaviors specified by UML statechart diagrams. This tool utilizes Spin as a backend to perform model checking and creates a UML sequence diagram according to the counterexample provided by Spin. The formal semantics is defined in Reference [142]. The authors also conduct a case study with the production cell example in Reference [141].

[10] https://www.eclipse.org/papyrus/.
[11] https://www.iar.com/products/iar-visual-state/.
[12] https://www.ibm.com/fr-fr/products/uml-tools.
[13] https://www.itemis.com/en/yakindu/state-machine/.
[14] https://plantuml.com/.
[15] https://sourceforge.net/projects/vuml/.

vUML aims at checking collaborations of UML models instead of a single UML state machine. vUML provides an event generator to emulate external events without parameters and removes external events carrying parameters to avoid state space explosion.

vUML can check the following properties: deadlock, livelock, reaching an invalid state, violating a constraint on an object, sending an event to a terminated object, overrunning the input queue of an object, and overrunning the deferred event queue.

To verify LTL formulas with UML, the user needs to understand the PROMELA model to come up with a proper LTL formula, which we consider to be a potential drawback.

A nice advantage of vUML is that, when the property is violated, it converts the counterexample output by Spin into a UML sequence diagram, hence offering a visual trace to the designer.

vUML does not seem to be either available online nor maintained anymore.[16]

JACK. Gnesi et al. [89] provide an algorithm to support direct model checking UML statecharts based on the formal semantics they have defined in Reference [136]. The implementation is based on the tool set **JACK ("Just Another Concurrency Kit")** [40], which is an environment based on process algebras, automata, and a temporal logic formalism. Different components of the JACK tool set communicate with the FC2 format. There is a model-checking tool in the JACK tool set named AMC that supports ACTL model checking. The system should be translated into the FC2 format first to utilize the AMC component. The users also need to specify their own ACTL property according to the model. This requires users to have a knowledge of model checking, the underlying model as well as temporal logic formulas.

HUGO. Knapp et al. [185] developed a tool called HUGOthat translates UML state machines into PROMELA, which is then verified using the Spin model checker. HUGO requires the presence of Java and (of course) Spin to be executed. HUGO used to be available online (see Table 7) in the form of a binary; no source code is available.

HUGO/RT. HUGO/RT is a UML model translator for model checking and code generation. The current HUGO/RT is a rewrite of both HUGO [185] and HUGO/RT [117]. A UML model containing active classes with state machines, collaborations, interactions, and OCL constraints can be translated into Java, C++, Arduino, PROMELA (Spin), and timed automata (UPPAAL), by first representing UML state machines into an intermediate common language called Smile. Several analyses based on partial evaluation are used on the Smile-level to produce performant and readable code. A similar approach is used for an intermediate representation of UML interactions in the language Ida, which can be translated to PROMELA and timed automata.

As of today (2022), all language constructs for state machines are supported by HUGO/RT, with the exception of submachines, connection point references, and entry/exit points.

HUGO/RT requires a Java environment to work, but is then multi-platform for the same reason. HUGO/RT is available online for download (see Table 7) in the form of a binary; the license is unclear, but the tool's Web page invites interested users to write to an email address to obtain the source code. Several applications were made, notably, to model checking (possibly timed) interactions [122, 197] and to coloring test cases for software product lines [121]. The development is still active as of 2022.

ASM-based Verification Tool. Shen et al. [191] introduce a tool based on an ASM model checker (which is itself based on the SMV model checker). The semantics they adopt is defined in Reference

[16]The page cited in Reference [143] (http://www.abo.fi/~iporres/vUML/vUML.html) does not seem to be available anymore. Also note that vUML should not be confused with VUML (standing for Visual UML), an open source project available at https://sourceforge.net/projects/vuml/.

[59]. This tool set supports both static and dynamic checks of UML diagrams. For static aspects, syntax as well as well-formedness rules given by OCL can be checked. Static views in UML, such as object diagram and class diagram, can also be transformed into ASM and checked. For the dynamic aspects, UML state machine diagrams are transformed into ASM models, and an ASM model checker is invoked to do the model checking. This tool takes UML diagrams specified in the XMI format as input and outputs a counterexample in the form given by the SMV model checker (since the dynamic checking component of this toolset is based on the SMV model checker). The counterexample trace can be fed to their analysis tool, which will analyze the error trace and produce some UML diagrams such as sequence diagrams or a collaboration diagram to the users. Details about the tool are described in Reference [59], and details about the transformation procedures are discussed in Reference [48].

TABU. Beato et al. [26] introduce a tool called **TABU ("Tool for the Active Behavior of UML")**. TABU takes UML diagrams (activity and state diagrams) in the form of XMI as input, automatically translates them into an `.smv` representation (the input format of SMV) and calls the Cadence SMV model checker to verify the UML model. In addition, TABU also provides an assistant for writing LTL/CTL properties to verify against the model. This feature makes the underlying model and the translation procedure transparent to the users and solves the problem faced by vUML [142] to some extent.

The translation covers most UML 2.0 features (though not described in detail in their paper) except for synchronization states, events with parameters, and dynamic creation and destruction of objects. It also provides guides in writing properties. A limitation is that the counterexample is given in the input format of SMV, which is not intuitive for model designers to map to their models.

PROCO. PROCO translates a UML state machine (version 1.4, described in XMI format supported by the CORA tool [7]) into PROMELA, which is discussed in Reference [111]. No details are discussed in that paper, but the paper reports the bugs found by the tool, which shows its practical effectiveness.

UML-B State Machine Animation Tool. UML-B state machine Animation [184] is able to translate a UML-B diagram into an Event-B representation and utilizes ProB [1]—a plug-in of the Rodin platform [5]—to perform the simulation and model-checking tasks. The tool is still available: It can be installed from the Rodin platform and is still maintained (latest version is 3.7 as of June 2022).

USMMC. Liu et al. implemented into the USMMC tool [145] the operational semantics defined in Reference [144].[17] The tool supports most features of UML state machines and is capable of model checking various properties, such as safety, deadlock-freeness, and LTL. Although the verification of more elaborated properties (written in LTL) requires the mastering of temporal logics, model-checking safety properties as well as deadlock-freeness and liveness can be used without any additional knowledge. USMMC is a standalone tool using the PAT model-checking library [198].

EMI-UML. In Reference [36], the tool EMI-UML (EMI stands for "embedded model interpreter") [32–34] is used to perform monitoring and formal verification of UML state machines. The tool is equipped with a well-founded theory, described in several publications. LTL model checking and deadlock-freeness checking are both supported. The tool is primarily available for Linux, with runtime execution of UML models available for Windows and MacOS, too, but not formal verification.

[17]Note that some of the authors of this survey have been involved in the development of USMMC.

Table 6. Summary of the Tools: Features

Tool	Reference	Model checker	GUI	Manual	Reach	DLF	LTL	CTL	Counterexample
vUML	[143]	Spin	× (?)	× (?)	√	√	√ (?)	× (?)	√
JACK	[89]	AMC	× (?)	× (?)	√	× (?)	× (?)	√	× (?)
HUGO	[185]	Spin	×	× (?)	√	√	× (?)	× (?)	?
HUGO/RT	[117]	Spin/UPPAAL	×	√	√	√	√	√	√
ASM-based	[191]	SMV	× (?)	× (?)	√	× (?)	× (?)	× (?)	√
TABU	[26]	SMV	×	× (?)	√	× (?)	√	√	○
PROCO	[111]	Spin	× (?)	× (?)	√	√	× (?)	× (?)	× (?)
UML-B	[184]	ProB	√	√	√	× (?)	× (?)	× (?)	× (?)
USMMC	[145]	(standalone)	√	× (?)	√	√	√	×	○
AnimUML	[110]	OBP2	√	×	√	√	√	×	√
EMI-UML	[36]	OBP2	√	√	√	√	√	×	√

The underlying model checker is OBP2, which can be interfaced with either EMI-UML or AnimUML (see below) to perform formal verification on the UML model executed by EMI-UML or AnimUML. The authors defined some generic interfaces between OBP2 and EMI execution engines such that they can use other model checkers (tested with a FPGA-based model-checker called Dolmen [84]) or other controllers. EMI-UML has not been officially released, neither put in open-source yet. The tool is still in active development as of 2022.

AnimUML. Recently, a tool called AnimUML was proposed by Jouault et al. in References [109, 110] and uses the same model checker (OBP2) as EMI-UML. One of its specificities is to allow for *partial* UML models and to maintain a list of "semantic relaxation points," as the UML specification is sometimes ambiguous. The set of syntactic features is rather small, but additional diagrams such as sequence diagrams can also be considered. AnimUML also has the possibility to represent counterexamples as UML sequence diagrams.

KandISTI/UMC toolset. As mentioned in Section 3.1.2, the KandISTI/UMC toolset relies on the UMC formalism, together with properties specified using UCTL [199]. While UMC is not, strictly speaking, UML, both formalisms share many similarities, and we believe that KandISTI/UMC could be used to verify UML models, possibly after some manual adaptation or automated translation. The tool is available for Linux, MacOS, and Windows, as well as using a browser equipped with HTML and JavaScript. Since it does not, strictly speaking, take UML state machines as input, we do not review the KandISTI/UMC toolset in Table 6.

6.2 Summary

Table 6 summarizes some information of the surveyed tools with—from left to right—the tool name and its main reference, the underlying model checking engine (if any), whether the tool features a graphical user interface, whether a decent user manual is publicly available, and the types of properties available: reachability or safety, deadlock-freeness, LTL model checking, CTL model checking, and whether a counterexample can be returned (○ denotes a counterexample in the translated formalism, while √ denotes a counterexample back to the original UML model). Most of these information were obtained from the associated tool papers and/or documentation, since most tools are not available anymore (see Table 7). This also explains the large number of "?" in Table 6.

We give in Table 7 the most recent known URL of each tool, followed from left to right by the actual availability (for download, on the Web), the supported operating systems (√∗ denotes a support of Linux or Mac OS, using the mono utility, which is an open source implementation of Microsoft's .NET Framework, or for Windows using Cygwin), the earliest and latest modification, and the license (when known).

Table 7. Summary of the Tools: Availability and Platforms

Tool	URL	Available	Linux	Mac OS	Windows	Started	Latest	License
vUML	http://www.abo.fi/~iporres/vUML	×	?	?	?	1999	1999?	Open source
JACK	×	×	?	?	?	1994	1999?	Not available
HUGO	http://www.pst.ifi.lmu.de/projekte/hugo/usage.html	×	√	× (?)	× (?)	?	2002	Unclear
HUGO/RT	https://www.uni-augsburg.de/en/fakultaet/fai/informatik/prof/swtsse/hugo-rt/	√	√	√	√*	2004	2022	Free
ASM-based	×	×	?	?	?	2001	2002?	Not available
TABU	×	×	?	?	?	2004	2005?	Not available
PROCO	http://www.tcs.hut.fi/SMUML/	×	?	?	?	2006?	2006?	Not available
UML-B	https://www.uml-b.org/	√	√	√	√	2010?	2022	Not available
USMMC	http://www.comp.nus.edu.sg/~lius87/UMLSM.html	×	√*	√*	√	2013	2013	Free (proprietary)
AnimUML	https://github.com/fjouault/AnimUML	√	√	√	√	2020	2022	Open source
EMI-UML	http://www.obpcdl.org/bare-metal-uml/	√	√	(√)	(√)	2017	2022	Free

We draw some conclusions on the surveyed tools in the following:

Delegation to external model checkers. We notice from Table 6 that all the available tools, with the notable exception of USMMC and EMI-UML, just provide a front-end supporting translation from UML state machines to languages of existing model checkers. Such a translation may introduce extra cost for the verification procedure.

Soundness. For the tools using a translation to an external language or engine, the informal translation procedure does not, in general, guarantee the soundness of the obtained model. In addition, almost none of the model-checkers are "certified" in the sense of a fully proved translation and verification. One may argue that this might be less critical for UML diagrams than for, e.g., code to be embedded into a safety-critical system; still, this is unfortunate that almost no attempt was performed to provide users with a certified model checker addressing the verification of UML state machines. The only exception is in Reference [154]; but this is not, strictly speaking, a certified model-checking engine, but rather a verified translation (using Isabelle/HOL).

Counterexample. In most tools, it is hard to map the found vulnerabilities back to the original UML model. That is, only three tools (vUML, "ASM-based," and HUGO/RT) map the possible counterexample back to the original UML model. Other tools either show no counterexample or exhibit it but fail in mapping it to the original model (e.g., USMMC).

Long-term availability. An obvious conclusion from Table 7 is that, with four notable exceptions (HUGO/RT, UML-B, AnimUML, and EMI-UML), all tools seem to be unavailable publicly nowadays. None of the advertised URLs in the papers work nowadays. Research using search engines all failed, and no code or binary seems to be archived on publicly available code repositories (such as GitHub), nor on long-term archiving venues (such as Zenodo). This is a major issue of the research related to (formal) verification of UML state machines: Most prototypes have been designed in the framework of an academic research, in the context of one (or several) particular paper(s), and the authors did not perform the necessary steps to make their tool publicly available, nor ensure the long-term conservation of it. This trend goes by far beyond the context of UML verification, and, unfortunately, concerns most areas of academic research concerned with tool development. The fact that most of the surveyed prototypes are relatively old (with three exceptions, the development of all tools surveyed in Tables 6 and 7 started before 2010) does not improve the situation, as, at that time, researchers may have been less concerned with long-term availability and experiment reproducibility than nowadays. Note that even one of the most recent tools (USMMC, developed in 2013) seems to have been lost as well.[18]

[18]The story of this loss is, unfortunately, classical: The developer of USMMC wrote the tool code during her Ph.D. thesis. After graduation, her university deleted all her data and, while she did perform some backup on external drives, these were eventually lost over the years.

7 RELATED SURVEYS

We survey here the previous attempts to summarize the formalization of UML state machines. In the past, to the best of our knowledge, there were three main surveys on the subject [37, 62, 146].

Bhaduri et al. The first survey dates from 2004. Bhaduri et al. [37] summarize approaches that translated variants of statecharts (including STATEMATE statecharts and UML statecharts) into the input language of the SMV and Spin model checkers. The survey only covers a subset of approaches, with detailed descriptions and discussions about each individual work. The paper only lists a subset of works translating statecharts into SMV or Spin. It does not provide any comparison among those works or conclusive comments on those works. Finally, Reference [37] focuses on many variants of Harel's statechart [95, 97, 98], such as RSML or UML.

Crane and Dingel. Crane and Dingel [62] provide in 2005 a categorization and comparison of 26 different approaches of formalizing UML state machine semantics (including denotational and operational semantics). They categorize those approaches based on the underlying formalism used and conduct comparisons on other dimensions such as UML state machine features coverage or tool support. That paper provides a high-level comparison and discussion on different aspects of the existing 26 approaches. Although the amount of work discussed is not large, it covers different kinds of approaches and provides a good way to categorize those approaches. The categorization of the works is wide, but the coverage within each categorization is too narrow, as several translation-based approaches are not included.

Lund et al. In 2010, Lund et al. [146] survey existing works on formalizing UML sequence diagram-like and state machine diagram-like semantics. Their survey does not focus on a thorough coverage of all the existing approaches; instead, it selectively discusses some representative approaches that fall in one of their categorization criteria. The focus is on both sequence diagrams and state machine diagrams. Although the survey provides new comparison criteria such as the supported properties or refinement support, it covers a limited number of approaches only, especially for the UML state machine part.

Other surveys. In Reference [206], Von der Beeck compares variants of the "statecharts" (prior to their formalization by the OMG), notably, in terms of concurrency model (true concurrency or interleaving), timeout on trigger transitions, and so on.

In Reference [138], Lee and Yannakakis discuss *testing* finite state machines.

In Reference [22], Balsamo and Simeoni survey the transformation of UML models into performance models. This survey came at the early stage of UML formalization and does not focus only on state machine diagrams, but also considers class, use case, activity, sequence, collaboration, deployment, and component diagrams. It remains, therefore, relatively shallow on that particular aspect.

In Reference [149], two different interpretations of the statecharts step semantics are considered (notably, w.r.t. simulation, ready trace preorder, failure preorder, and trace preorder).

In Reference [30], dependability modeling with UML is surveyed. That survey does not focus specifically on UML state machines.

In Reference [119], a survey is made on 57 approaches related to the consistency of multi-view models in UML/OCL. All 14 different UML diagram types are considered (not only state machines); but few approaches cover many diagram types; in fact, even the most comprehensive surveyed approach covers (partially) only five UML diagram types.

Other surveys were proposed for other UML diagrams, e.g., sequence diagrams in Reference [161].

8 CONCLUSION

In this manuscript, we provide a detailed survey of approaches aiming at giving a formal semantics to UML behavioral state machines, thus enabling their automated verification. We categorize the approaches into two major groups, viz., the translation approaches and those directly providing operational semantics. In each group, we also provide comparisons of the surveyed approaches on dimensions such as UML version, coverage of syntactic features, and tool support. We also try to provide a focus on tool support and implementation.

8.1 Main Conclusions

Completeness. In translation-based approaches, numerous works support various syntactic features, but all fail in supporting a complete or near-to-complete subset of the UML syntax. Table 2, which summarizes these approaches, shows that no work supports more than 65% of the existing syntactic features, and most works support less than 50%, sometimes far less. An additional difficulty comes from the fact that, while a certain level of formalization would be expected from works aiming at *formalizing* UML, several works describe their formalization in a rather "informal" (textual) manner; this results in numerous "?" or ∘ in Table 2. However, all syntactic features are supported by at least one translation approach.

Approaches providing UML state machines with a dedicated operational semantics perform better: While many approaches support only a restricted subset of the UML syntax, two (viz., References [81, 144]) support almost completely the UML syntax, as shown in Table 4.

These comments on the completeness also raise questions on the usefulness of the UML syntax: Since most works (especially translation-based) only support a restricted subset of the UML syntax, are the remaining syntactic features useful? In other words, are the infrequent syntactic constructs not considered in most papers because they are deemed of low usefulness or because they lead to complex translations? The answer probably goes beyond this survey, and it would be worth investigating the actual use of the UML in practice to survey which syntactic features are most commonly used.

Tools long-term availability. A frustrating outcome from Table 7 is that most tools developed over the years for the formal verification of UML state machines are now lost (i.e., they seem completely unavailable online). While this phenomenon certainly goes beyond this particular research on formalization of UML, it is particularly obvious, as all the tools we surveyed—with only four exceptions—are lost. This should be in itself a motivation for researchers to handle with much more care the prototypes they develop to ensure they remain permanently available for several reasons: (1) public availability; (2) surveying purpose; (3) avoiding to re-invent previously coded translations; and, (4) experiment reproducibility and comparisons. We believe long-term archiving venues (such as Zenodo, providing a **digital object identifier (DOI)** to software and data) should be frequently used. Also, the Software Heritage initiative [69] can be used to that purpose.[19]

8.2 Perspectives

Consistency. An outcome of our survey is that, overall, few works considered large subsets of the UML state machine syntax in their formalization. Similarly, no single syntactic element surveyed in Tables 2 and 4 was addressed by all works (note that our tables exclude "trivial" elements, such as simple states or initial states). Therefore, a first perspective includes studying the consistency of

[19] In addition to manual archiving requests, the Software Heritage initiative automatically browses, archives, and replicates several existing repositories, such as most of the software code stored publicly in large repositories such as BitBucket, GitHub, GitLab, or HAL.

these works over the formalized elements of the UML syntax. Whether a common understanding of these elements throughout the literature has been met is not clear, and this understanding could be put in perspective with fUML and SysML (see below).

A full formalization. Clearly, the perfect formalization of UML state machine diagrams remains to be done, as none of the existing approaches is entirely satisfactory. One may wonder why it has not been done before.

On the one hand, academic papers may not always aim at completeness, but rather at formalizing some yet-uncovered syntactic feature or showing that (some of) the UML syntax can be encoded using a formalism that was not used for that purpose yet. Difficulties can also come from the target formalism, e.g., it can be difficult to encode the complex hierarchy of entry and exit behaviors using a formalism such as Petri nets.

On the other hand, one may wonder whether the entire syntax of UML is useful for practical purposes (see discussion above). In addition, the OMG semi-formal semantics is intrinsically not (entirely) formal, and obstacles to the formalization can also come from some perhaps unnecessarily complex syntactic elements.

A minimal syntax. Whether the syntax of UML state machines as described in the specification can be reduced to a minimal subset remains blurry. In other words, can we (really) encode perfectly some constructs into others?

On the one hand, it is, for example, claimed that a submachine state is "semantically equivalent to a composite State" [170, p. 311]. Or that "an entry point is equivalent to a junction pseudostate (fork in cases where the composite state is orthogonal)" [170, p. 311]. We may wonder whether this is perfectly accurate. For example, if a model contains recursive references (e.g., a state machine that would contain a submachine state linking to itself), then the model becomes ill-formed (infinitely nested); this cannot happen by using only regular composite states, and therefore the two constructs are not, strictly speaking, equivalent.

On the other hand, it may be relatively straightforward to show that junctions could be encoded using a number of independent transitions.

Adoption of the UML in the industry. While many academic works targeted the formalization of UML diagrams (as it is shown by this article, dozens of academic works targeted the formalization of the sole UML state machines), the use of UML diagrams in an industrial context is debatable.

First, it is noted in Reference [45] that "there are few studies of adoption [of the UML] and use in the field." Among the 49 papers surveyed in Reference [45], only 2 aim at surveying the *adoption* of the UML.

Then, in Reference [173], Marian Petre reports on 50 interviews with software-engineering professionals over two years; among these 50 interviews, 35 did not use UML at all, and only 3 out of 50 used it for automated code generation—which can be seen as one of the goals (though not the only one) of using UML diagrams. In addition, among the 11 out of 50 professionals using a selective subset of the UML, only 3 out of 11 used state machine diagrams. Petre therefore concludes that, while UML might still be a "*de facto* standard," it is "by no means universally adopted." In addition, according to the author of Reference [174], most reactions from software professionals to this study were that it came with absolutely no surprise ("the response from software professionals was largely 'No shit"' [174]).

These works show that, despite an extensive academic literature on formalizing (or using) UML diagrams, its adoption in the industry can be largely improved. A main challenge is not only to understand the gap between the works done by the academic (and, notably, the formalization works), on the one hand, and the actual need in the software industry, on the other hand.

Other UML diagrams. Verification of UML state machines was in most (but not) all cases performed in isolation, while a verification taking other UML diagrams into consideration would be highly welcome. UML state machines can be considered jointly with sequence diagrams or class diagrams or even activity diagrams. A future global approach supporting various types of diagrams simultaneously would therefore be most interesting.

Extensions. Few works studied quantitative extensions of state machines with time and probabilities. Time is briefly mentioned in the official (semi-formal) semantics [170], while probabilities are not. But either timing or probabilistic aspects are considered by a minor number of works (notably, References [55, 66, 117, 124, 201, 202]). A probabilistic extension of UML statecharts was also proposed in Reference [104]. This seems an interesting direction of research to us, as these quantitative extensions can be of great use for practical systems. Also, real-time extensions of UML were proposed, which includes krtUML [64], but also the MARTE profile [150]. An orthogonal question is also whether a real-time semantics is necessary for the UML or if a discrete semantics can be sufficient.

Another interesting perspective is the relationship with the **Foundational UML (fUML)** [171], which aims at providing an executable semantics for a subset of the UML syntax. Notably, is the semantics considered in the works surveyed in this manuscript compatible with that of fUML? Is the subset of syntactic features of UML state machines that do not belong to fUML useful for practical needs? This would deserve a survey in itself. A similar perspective can be made on SysML.

Tool support. As surveyed in Table 6, the tool support for formal UML verification remains unsatisfactory. Beyond the fact that most tools are now unavailable (see discussion above), none of the tools is entirely satisfactory: The subset of the considered syntax is usually small, counterexamples are rarely mapped back to the original models, and most tools are prototypes with no or basic GUI, and few OS support (not even mentioning usage through a Web access). This is probably more an engineering issue rather than a research issue, considering the fact that a satisfactory operational semantics was defined in at least two works (References [81, 144]).

It remains unfortunate that no team dedicated enough efforts into providing the community with a decent verification engine for UML state machines; nevertheless, three tools still under development as of 2022 (HUGO/RT, AnimUML, and EMI-UML) do propose a satisfactory support for UML state machines verification.

Finally, a further perspective would be to prove the correctness of the verification, perhaps using proof certificates (e.g., à la Reference [61]).

ACKNOWLEDGMENTS

We warmly thank the reviewers for their numerous useful suggestions. We thank the developers of AnimUML, EMI-UML, and HUGO/RT for their useful precisions concerning their tools. We would also like to thank Mohamed Mahdi Benmoussa for his help with Figure 1.

REFERENCES

[1] ProB. [n. d.]. The ProB Animator and Model Checker. Retrieved from https://prob.hhu.de/.

[2] SMUML. [n. d.]. Symbolic Methods for UML Behavioural Diagrams (SMUML). Retrieved from http://www.tcs.hut.fi/Research/Logic/SMUML.shtml.

[3] 2012. mCRL2, a Specification Language and Toolset. Retrieved from https://www.mcrl2.org/web/user_manual/.

[4] Jean-Raymond Abrial. 1996. *The B-book - Assigning Programs to Meanings.* Cambridge University Press. DOI: https://doi.org/10.1017/CBO9780511624162

[5] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. 2010. Rodin: An open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.* 12, 6 (2010), 447–466. DOI: https://doi.org/10.1007/s10009-010-0145-y

[6] Sabah Al-Fedaghi. 2020. Modeling the semantics of states and state machines. *J. Comput. Sci.* 16, 7 (2020). DOI: https://doi.org/10.3844/jcssp.2020.891.905

[7] Marcus Alanen and Ivan Porres. 2004. Coral: A metamodel kernel for transformation engines. Technical report. In *MDA*, David H. Akehurst (Ed.). Kent University. Retrieved from https://kar.kent.ac.uk/14116/.

[8] Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoret. Comput. Sci.* 126, 2 (Apr. 1994), 183–235. DOI: https://doi.org/10.1016/0304-3975(94)90010-8

[9] Rajeev Alur and Radu Grosu. 2004. Modular refinement of hierarchic reactive machines. *ACM Trans. Program. Lang. Syst.* 26, 2 (2004), 339–369. DOI: https://doi.org/10.1145/973097.973101

[10] Rajeev Alur, Radu Grosu, and Michael McDougall. 2000. Efficient reachability analysis of hierarchical reactive machines. In *CAV (LNCS)*, E. Allen Emerson and A. Prasad Sistla (Eds.), Vol. 1855. Springer, 280–295. DOI: https://doi.org/10.1007/10722167_23

[11] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. 1999. Communicating hierarchical state machines. In *ICALP (LNCS)*, Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen (Eds.), Vol. 1644. Springer, 169–178. DOI: https://doi.org/10.1007/3-540-48523-6_14

[12] Rajeev Alur, Michael McDougall, and Zijiang Yang. 2002. Exploiting behavioral hierarchy for efficient model checking. In *CAV (LNCS)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.), Vol. 2404. Springer, 338–342. DOI: https://doi.org/10.1007/3-540-45657-0_25

[13] Rajeev Alur and Mihalis Yannakakis. 2001. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 273–303. DOI: https://doi.org/10.1145/503502.503503

[14] Torben Amtoft, Kelly Androutsopoulos, and David Clark. 2020. Correctly slicing extended finite state machines. In *From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement (LNCS)*, Alessandra Di Pierro, Pasquale Malacaria, and Rajagopal Nagarajan (Eds.), Vol. 12065. Springer, 149–197. DOI: https://doi.org/10.1007/978-3-030-41103-9_6

[15] Étienne André. 2013. Observer patterns for real-time systems. In *ICECCS*, Yang Liu and Andrew Martin (Eds.). IEEE Computer Society, 125–134. DOI: https://doi.org/10.1109/ICECCS.2013.26

[16] Kelly Androutsopoulos, David Clark, Mark Harman, Robert M. Hierons, Zheng Li, and Laurence Tratt. 2013. Amorphous slicing of extended finite state machines. *Trans. Softw. Eng.* 39, 7 (2013), 892–909. DOI: https://doi.org/10.1109/TSE.2012.72

[17] Étienne André, Mohamed Mahdi Benmoussa, and Christine Choppy. 2016. Formalising concurrent UML state machines using coloured Petri nets. *Form. Asp. Comput.* 28, 5 (Sept. 2016), 805–845. DOI: https://doi.org/10.1007/s00165-016-0388-9

[18] Étienne André, Christine Choppy, and Kais Klai. 2012. Formalizing non-concurrent UML state machines using colored Petri nets. *ACM SIGSOFT Softw. Eng. Notes* 37, 4 (2012), 1–8. DOI: https://doi.org/10.1145/2237796.2237819

[19] Toshiaki Aoki, Takaaki Tateishi, and Takuya Katayama. 2001. An axiomatic formalization of UML models. In *pUML (LNI)*, Andy Evans, Robert B. France, Ana M. D. Moreira, and Bernhard Rumpe (Eds.), Vol. P-7. GI, 13–28. Retrieved from https://dl.gi.de/20.500.12116/30860.

[20] Demissie B. Aredo. 2000. Semantics of UML statecharts in PVS. In *NWPT*.

[21] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

[22] Simonetta Balsamo and Marta Simeoni. 2001. On transforming UML models into performance models. In *Workshop on Transformations in the UML*.

[23] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. 2004. Interactive verification of UML state machines. In *ICFEM (LNCS)*, Jim Davies, Wolfram Schulte, and Michael Barnett (Eds.), Vol. 3308. Springer, 434–448. DOI: https://doi.org/10.1007/978-3-540-30482-1_36

[24] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. 1998. KIV 3.0 for provably correct systems. In *FM-Trends (LNCS)*, Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann (Eds.), Vol. 1641. Springer, 330–337. DOI: https://doi.org/10.1007/3-540-48257-1_23

[25] Luciano Baresi and Mauro Pezzè. 2001. On formalizing UML with high-level Petri nets. In *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets (LNCS)*, Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg (Eds.), Vol. 2001. Springer, 276–304. DOI: https://doi.org/10.1007/3-540-45397-0_9

[26] Maria Encarnación Beato, Manuel Barrio-Solórzano, Carlos Enrique Cuesta Quintero, and Pablo de la Fuente. 2005. UML automatic verification tool with formal methods. *Electron. Notes Theoret. Comput. Sci.* 127, 4 (2005), 3–16. DOI: https://doi.org/10.1016/j.entcs.2004.10.024

[27] Jörg Beckers, Daniel Klünder, Stefan Kowalewski, and Bastian Schlich. 2008. Direct support for model checking abstract state machines by utilizing simulation. In *ABZ (LNCS)*, Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca (Eds.), Vol. 5238. Springer, 112–124. DOI: https://doi.org/10.1007/978-3-540-87603-8_10

[28] Gerd Behrmann, Kim Guldstrand Larsen, Henrik Reif Andersen, Henrik Hulgaard, and Jørn Lind-Nielsen. 2002. Verification of hierarchical state/event systems using reusability and compositionality. *Form. Meth. Syst. Des.* 21, 2 (2002), 225–244. DOI: https://doi.org/10.1023/A:1016095519611

[29] Simona Bernardi, Susanna Donatelli, and José Merseguer. 2002. From UML sequence diagrams and statecharts to analysable Petri net models. In *WOSP@ISSTA*. ACM, 35–45. DOI: https://doi.org/10.1145/584369.584376

[30] Simona Bernardi, José Merseguer, and Dorina C. Petriu. 2012. Dependability modeling and analysis of software systems specified with UML. *Comput. Surv.* 45, 1 (2012), 2:1–2:48. DOI: https://doi.org/10.1145/2379776.2379778

[31] Bernard Berthomieu and François Vernadat. 2006. Time Petri nets analysis with TINA. In *QEST*. IEEE Computer Society, 123–124. DOI: https://doi.org/10.1109/QEST.2006.56

[32] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov. 2017. Towards one model interpreter for both design and deployment. In *EXE (CEUR Workshop Proceedings)*, Loli Burgueño, Jonathan Corley, Nelly Bencomo, Peter J. Clarke, Philippe Collet, Michalis Famelis, Sudipto Ghosh, Martin Gogolla, Joel Greenyer, Esther Guerra, Sahar Kokaly, Alfonso Pierantonio, Julia Rubin, and Davide Di Ruscio (Eds.), Vol. 2019. CEUR-WS.org, 102–108. Retrieved from http://ceur-ws.org/Vol-2019/exe_4.pdf.

[33] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. 2018. Embedded UML model execution to bridge the gap between design and runtime. In *STAF (LNCS)*, Manuel Mazzara, Iulian Ober, and Gwen Salaün (Eds.), Vol. 11176. Springer, 519–528. DOI: https://doi.org/10.1007/978-3-030-04771-9_38

[34] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. 2018. Unified LTL verification and embedded execution of UML models. In *MODELS*, Andrzej Wasowski, Richard F. Paige, and Øystein Haugen (Eds.). ACM, 112–122. DOI: https://doi.org/10.1145/3239372.3239395

[35] Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. 2019. Verifying and monitoring UML models with observer automata: A transformation-free approach. In *MODELS*, Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño (Eds.). IEEE, 161–171. DOI: https://doi.org/10.1109/MODELS.2019.000-5

[36] Valentin Besnard, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, and Philippe Dhaussy. 2021. Unified verification and monitoring of executable UML specifications. *Softw. Syst. Model.* 20, 6 (2021), 1825–1855. DOI: https://doi.org/10.1007/s10270-021-00923-9

[37] Purandar Bhaduri and S. Ramesh. 2004. *Model Checking of Statechart Models: Survey and Research Directions*. Technical Report cs.SE/0407038. arXiv. Retrieved from https://arxiv.org/abs/cs/0407038.

[38] Tommaso Bolognesi and Ed Brinksma. 1987. Introduction to the ISO specification language LOTOS. *Comput. Netw.* 14 (1987), 25–59. DOI: https://doi.org/10.1016/0169-7552(87)90085-7

[39] Amar Bouali, Stefania Gnesi, and Salvatore Larosa. 1994. *The Integration Project for the JACK Environment*. Technical Report CS-R9443. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[40] Amar Bouali, Stefania Gnesi, and Salvatore Larosa. 1994. JACK: Just another concurrency kit. The integration project. *Bull. EATCS* 54 (1994), 207–223.

[41] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. 2004. The IF toolset. In *SFM-RT (LNCS)*, Marco Bernardo and Flavio Corradini (Eds.), Vol. 3185. Springer, 237–267. DOI: https://doi.org/10.1007/978-3-540-30080-9_8

[42] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. 1997. Towards a formalization of the unified modeling language. In *ECOOP (LNCS)*, Mehmet Aksit and Satoshi Matsuoka (Eds.), Vol. 1241. Springer, 344–366. DOI: https://doi.org/10.1007/BFb0053386

[43] Jean-Michel Bruel and Robert B. France. 1998. Transforming UML models to formal specifications. In *1998 Workshop on Formalizing UML. Why? How?*

[44] David Budgen and Pearl Brereton. 2006. Performing systematic literature reviews in software engineering. In *ICSE*, Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.). ACM, 1051–1052. DOI: https://doi.org/10.1145/1134285.1134500

[45] David Budgen, Andy J. Burn, O. Pearl Brereton, Barbara A. Kitchenham, and Rialette Pretorius. 2011. Empirical evidence about the UML: A systematic literature review. *Softw. - Pract. Exper.* 41, 4 (2011), 363–392. DOI: https://doi.org/10.1002/spe.1009

[46] Michael J. Butler, John Colley, Andrew Edmunds, Colin F. Snook, Neil Evans, Neil Grant, and Helen Marshall. 2013. Modelling and refinement in CODA. In *Refine@IFM (EPTCS)*, John Derrick, Eerke A. Boiten, and Steve Reeves (Eds.), Vol. 115. 36–51. DOI: https://doi.org/10.4204/EPTCS.115.3

[47] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. 2000. An ASM semantics for UML activity diagrams. In *AMAST (LNCS)*, Teodor Rus (Ed.), Vol. 1816. Springer, 293–308. DOI: https://doi.org/10.1007/3-540-45499-3_22

[48] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. 2000. Modeling the dynamics of UML state machines. In *ASM (LNCS)*, Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele (Eds.), Vol. 1912. Springer, 223–241. DOI: https://doi.org/10.1007/3-540-44518-8_13

[49] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. 2001. Solving conflicts in UML state machines concurrent states. In *CIUML*.

[50] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. 2002. A precise semantics of UML state machines: Making semantic variation points and ambiguities explicit. In *SFEDL*.
[51] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. 2004. On formalizing UML state machines using ASM. *Inf. Soft. Technol.* 46, 5 (2004), 287–292. DOI: https://doi.org/10.1016/j.infsof.2003.09.009
[52] Egon Börger, Elvinia Riccobene, and Alessandra Cavarra. 2003. Modeling the meaning of transitions from and to concurrent states in UML state machines. In *SAC*, Gary B. Lamont, Hisham Haddad, George A. Papadopoulos, and Brajendra Panda (Eds.). ACM, 1086–1091. DOI: https://doi.org/10.1145/952532.952745
[53] Egon Börger and Joachim Schmid. 2000. Composition and submachine concepts for sequential ASMs. In *CSL (LNCS)*, Peter Clote and Helmut Schwichtenberg (Eds.), Vol. 1862. Springer, 41–60. DOI: https://doi.org/10.1007/3-540-44622-2_3
[54] Egon Börger and Robert F. Stärk. 2003. *Abstract State Machines. A Method for High-level System Design and Analysis*. Springer. Retrieved from http://www.springer.com/computer/swe/book/978-3-540-00702-9.
[55] Mats Carlsson and Lars Johansson. 2009. *Formal Verification of UML-RT Capsules using Model Checking*. Master's Thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.
[56] Christine Choppy, Kais Klai, and Hacene Zidani. 2011. Formal verification of UML state diagrams: A Petri net based approach. *ACM SIGSOFT Softw. Eng. Notes* 36, 1 (Jan. 2011), 1–8. DOI: https://doi.org/10.1145/1921532.1921561
[57] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. NuSMV 2: An OpenSource tool for symbolic model checking. In *CAV (LNCS)*, Ed Brinksma and Kim Guldstrand Larsen (Eds.), Vol. 2404. Springer, 359–364. DOI: https://doi.org/10.1007/3-540-45657-0_29
[58] Edmund M. Clarke and Wolfgang Heinle. 2000. *Modular Translation of Statecharts to SMV*. Technical Report CMU-CS-00-XXX. Carnegie Mellon University.
[59] Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. 2000. *An Automatic Verification Tool for UML*. Technical Report CSE-TR-423-00. University of Michigan.
[60] Kevin J. Compton, James Huggins, and Wuwei Shen. 2000. A semantic model for the state machine in the unified modeling language. In *Proceedings of the UML 2000 Workshop: Dynamic Behaviour in UML Models: Semantic Questions*.
[61] Sylvain Conchon, Alain Mebsout, and Fatiha Zaïdi. 2015. Certificates for parameterized model checking. In *FM (LNCS)*, Nikolaj Bjørner and Frank S. de Boer (Eds.), Vol. 9109. Springer, 126–142. DOI: https://doi.org/10.1007/978-3-319-19249-9_9
[62] Michelle L. Crane and Jürgen Dingel. 2005. *On the Semantics of UML State Machines: Categorization and Comparison*. Technical Report 2005-501. School of Computing, Queen's University, Kingston, Ontario, Canada. Retrieved from https://research.cs.queensu.ca/TechReports/Reports/2005-501.pdf.
[63] Werner Damm, Bernhard Josko, Amir Pnueli, and Anjelika Votintseva. 2002. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *FMCO (LNCS)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 2852. Springer, 71–98. DOI: https://doi.org/10.1007/978-3-540-39656-7_3
[64] Werner Damm, Bernhard Josko, Amir Pnueli, and Anjelika Votintseva. 2005. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Sci. Comput. Program.* 55, 1-3 (2005), 81–115. DOI: https://doi.org/10.1016/j.scico.2004.05.012
[65] Werner Damm, Bernhard Josko, Anjelika Votintseva, and Amir Pnueli. 2003. *A Formal Semantics for a UML Kernel Language*. Technical Report IST-2001-33522. OMEGA.
[66] Alexandre David, M. Oliver Möller, and Wang Yi. 2002. Formal verification of UML statecharts with real-time extensions. In *FASE (LNCS)*, Ralf-Detlef Kutsche and Herbert Weber (Eds.), Vol. 2306. Springer, 218–232. DOI: https://doi.org/10.1007/3-540-45923-5_15
[67] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 theorem prover and programming language. In *CADE (LNCS)*, André Platzer and Geoff Sutcliffe (Eds.), Vol. 12699. Springer, 625–635. DOI: https://doi.org/10.1007/978-3-030-79876-5_37
[68] Giuseppe Del Castillo and Kirsten Winter. 2000. Model checking support for the ASM high-level language. In *TACAS (LNCS)*, Susanne Graf and Michael I. Schwartzbach (Eds.), Vol. 1785. Springer, 331–346. DOI: https://doi.org/10.1007/3-540-46419-0_23
[69] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software heritage: Why and how to preserve software source code. In *iPRES*, Shoichiro Hara, Shigeo Sugimoto, and Makoto Goto (Eds.). Retrieved from https://hdl.handle.net/11353/10.931064.
[70] Salim Djaaboub, Elhillali Kerkouche, and Allaoua Chaoui. 2015. From UML statecharts to LOTOS expressions using graph transformation. In *ICIST (CCIS)*, Giedre Dregvaite and Robertas Damasevicius (Eds.), Vol. 538. Springer, 548–559. DOI: https://doi.org/10.1007/978-3-319-24770-0_47

[71] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. 2008. Timed automata patterns. *Trans. Softw. Eng.* 34, 6 (2008), 844–859. DOI: https://doi.org/10.1109/TSE.2008.52

[72] Wei Dong, Ji Wang, Xuan Qi, and Zhichang Qi. 2001. Model checking UML statecharts. In *APSEC*. IEEE Computer Society, 363–370. DOI: https://doi.org/10.1109/APSEC.2001.991503

[73] Jori Dubrovin. 2006. *Jumbala — An Action Language for UML State Machines*. Technical Report HUT-TCS-A101. Helsinki University of Technology, Laboratory for Theoretical Computer Science, Finland.

[74] Jori Dubrovin and Tommi A. Junttila. 2008. Symbolic model checking of hierarchical UML state machines. In *ACSD*, Jonathan Billington, Zhenhua Duan, and Maciej Koutny (Eds.). IEEE, 108–117. DOI: https://doi.org/10.1109/ACSD.2008.4574602

[75] Jori Dubrovin, Tommi A. Junttila, and Keijo Heljanko. 2008. Symbolic step encodings for object based communicating state machines. In *FMOODS (LNCS)*, Gilles Barthe and Frank S. de Boer (Eds.), Vol. 5051. Springer, 96–112. DOI: https://doi.org/10.1007/978-3-540-68863-1_7

[76] Rik Eshuis and Roel J. Wieringa. 2000. Requirements level semantics for UML statecharts. In *FMOODS (IFIP Conference Proceedings)*, Scott F. Smith and Carolyn L. Talcott (Eds.), Vol. 177. Kluwer, 121–140. DOI: https://doi.org/10.1007/978-0-387-35520-7_6

[77] Andy Evans, Jean-Michel Bruel, Robert B. France, Kevin Lano, and Bernhard Rumpe. 1998. Making UML precise. In *1998 Workshop on Formalizing UML. Why? How?*

[78] Harald Fecher, Michael Huth, Heiko Schmidt, and Jens Schönborn. 2009. Refinement sensitive formal semantics of state machines with persistent choice. *Electron. Notes Theoret. Comput. Sci.* 250, 1 (2009), 71–86. DOI: https://doi.org/10.1016/j.entcs.2009.08.006

[79] Harald Fecher, Marcel Kyas, Willem Paul de Roever, and Frank S. de Boer. 2006. Compositional operational semantics of a UML-kernel-model language. *Electron. Notes Theoret. Comput. Sci.* 156, 1 (2006), 79–96. DOI: https://doi.org/10.1016/j.entcs.2005.08.008

[80] Harald Fecher, Marcel Kyas, and Jens Schönborn. 2005. *Semantic Issues in UML 2.0 State Machines*. Technical Report 0507. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universtität Kiel. Retrieved from https://macau.uni-kiel.de/receive/macau_mods_00001899. Bericht des Instituts für Informatik.

[81] Harald Fecher and Jens Schönborn. 2006. UML 2.0 state machines: Complete formal semantics via core state machine. In *FMICS/PDMC (LNCS)*, Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol (Eds.), Vol. 4346. Springer, 244–260. DOI: https://doi.org/10.1007/978-3-540-70952-7_16

[82] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem Paul de Roever. 2005. 29 new unclarities in the semantics of UML 2.0 state machines. In *ICFEM (LNCS)*, Kung-Kiu Lau and Richard Banach (Eds.), Vol. 3785. Springer, 52–65. DOI: https://doi.org/10.1007/11576280_5

[83] Houda Fekih, Leila Jemni Ben Ayed, and Stephan Merz. 2006. Transformation of B specifications into UML class diagrams and state machines. In *SAC*, Hisham Haddad (Ed.). ACM, 1840–1844. DOI: https://doi.org/10.1145/1141277.1141709

[84] Émilien Fournier, Ciprian Teodorov, and Loïc Lagadec. 2022. Dolmen: FPGA swarm for safety and liveness verification. In *DATE*. 1425–1430. DOI: https://doi.org/10.23919/DATE54114.2022.9774528

[85] Chris Fox and Arthorn Luangsodsai. 2005. And-or dependence graphs for slicing statecharts. In *Beyond Program Slicing (Dagstuhl Seminar Proceedings)*, David W. Binkley, Mark Harman, and Jens Krinke (Eds.), Vol. 05451. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. Retrieved from http://drops.dagstuhl.de/opus/volltexte/2006/493.

[86] Angelo Furfaro and Libero Nigro. 2005. Model checking hierarchical communicating real-time state machines. In *ETFA*. IEEE. DOI: https://doi.org/10.1109/ETFA.2005.1612546

[87] Angelo Furfaro and Libero Nigro. 2007. Timed verification of hierarchical communicating real-time state machines. *Comput. Stand. Interf.* 29, 6 (2007), 635–646. DOI: https://doi.org/10.1016/j.csi.2007.04.003

[88] Angelo Furfaro, Libero Nigro, and Francesco Pupo. 2006. Modular design of real-time systems using hierarchical communicating real-time state machines. *Real-time Syst.* 32, 1-2 (2006), 105–123. DOI: https://doi.org/10.1007/s11241-006-5318-0

[89] Stefania Gnesi, Diego Latella, and Mieke Massink. 1999. Model checking UML statechart diagrams using JACK. In *HASE*. IEEE Computer Society, 46–55. DOI: https://doi.org/10.1109/HASE.1999.809474

[90] Stefania Gnesi, Diego Latella, and Mieke Massink. 2002. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *J. Logic Algeb. Program.* 51, 1 (2002), 43–75. DOI: https://doi.org/10.1016/S1567-8326(01)00012-1

[91] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. 2006. The formal specification language mCRL2. In *Methods for Modelling Software Systems (MMOSS'06) (Dagstuhl Seminar Proceedings)*, Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel J. Wieringa (Eds.), Vol. 06351. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. Retrieved from http://drops.dagstuhl.de/opus/volltexte/2007/862.

[92] Steve Haga, Wei-Ming Ma, and William S. Chao. 2022. *Formalizing UML 2.0 State Machines Using a Structure-behavior Coalescence Method*. Technical Report. ResearchGate. Retrieved from https://www.researchgate.net/profile/William-S-Chao/publication/358199729_Formalizing_UML_20_State_Machines_Using_a_Structure-Behavior_Coalescence_Method/links/61f4d9f0aad5781d41b868bb/Formalizing-UML-20-State-Machines-Using-a-Structure-Behavior-Coalescence-Method.pdf.

[93] Alexandre Hamez, Lom Hillah, Fabrice Kordon, Alban Linard, Emmanuel Paviot-Adet, Xavier Renault, and Yann Thierry-Mieg. 2006. New features in CPN-AMI 3: Focusing on the analysis of complex distributed systems. In *ACSD*. IEEE Computer Society, 273–275. DOI: https://doi.org/10.1109/ACSD.2006.15

[94] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, Mohammad Reza Mousavi, and Jaco van de Pol. 2010. Towards model checking executable UML specifications in mCRL2. *Innov. Syst. Softw. Eng.* 6, 1-2 (2010), 83–90. DOI: https://doi.org/10.1007/s11334-009-0116-1

[95] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274. DOI: https://doi.org/10.1016/0167-6423(87)90035-9

[96] David Harel and Eran Gery. 1997. Executable object modeling with statecharts. *IEEE Comput.* 30, 7 (1997), 31–42. DOI: https://doi.org/10.1109/2.596624

[97] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark B. Trakhtenbrot. 1990. STATEMATE: A working environment for the development of complex reactive systems. *Trans. Softw. Eng.* 16, 4 (1990), 403–414. DOI: https://doi.org/10.1109/32.54292

[98] David Harel and Amnon Naamad. 1996. The STATEMATE semantics of statecharts. *Trans. Softw. Eng. Methodol.* 5, 4 (1996), 293–333. DOI: https://doi.org/10.1145/235321.235322

[99] David Harel and Michal Politi. 1998. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, Inc.

[100] Gerard J. Holzmann. 2004. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley.

[101] Zhaoxia Hu and Sol M. Shatz. 2004. Mapping UML diagrams to a Petri net notation for system simulation. In *SEKE*, Frank Maurer and Günther Ruhe (Eds.). 213–219.

[102] Jaco Jacobs and Andrew C. Simpson. 2014. A formal model of SysML blocks using CSP for assured systems engineering. In *FTSCS (CCIS)*, Cyrille Artho and Peter Csaba Ölveczky (Eds.), Vol. 476. Springer, 127–141. DOI: https://doi.org/10.1007/978-3-319-17581-2_9

[103] Jörn W. Janneck and Philipp W. Kutter. 1998. *Mapping Automata: Simple Abstract State Machines*. Technical Report 49. Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zürich (ETH). DOI: https://doi.org/10.3929/ethz-a-004289129

[104] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. 2002. A probabilistic extension of UML statecharts. In *FTRTFT (LNCS)*, Werner Damm and Ernst-Rüdiger Olderog (Eds.), Vol. 2469. Springer, 355–374. DOI: https://doi.org/10.1007/3-540-45739-9_21

[105] David N. Jansen and Roel Wieringa. 2002. Extending CTL with actions and real time. *J. Logic Computat.* 12, 4 (2002), 607–621. DOI: https://doi.org/10.1093/logcom/12.4.607

[106] Kurt Jensen and Lars Michael Kristensen. 2009. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer. DOI: https://doi.org/10.1007/b95112

[107] Yan Jin, Robert Esser, and Jörn W. Janneck. 2002. Describing the syntax and semantics of UML statecharts in a heterogeneous modelling environment. In *Diagrams (LNCS)*, Mary Hegarty, Bernd Meyer, and N. Hari Narayanan (Eds.), Vol. 2317. Springer, 320–334. DOI: https://doi.org/10.1007/3-540-46037-3_30

[108] Yan Jin, Robert Esser, and Jörn W. Janneck. 2004. A method for describing the syntax and semantics of UML statecharts. *Softw. Syst. Model.* 3, 2 (2004), 150–163. DOI: https://doi.org/10.1007/s10270-003-0046-6

[109] Frédéric Jouault, Valentin Besnard, Théo Le Calvar, Ciprian Teodorov, Matthias Brun, and Jérôme Delatour. 2020. Designing, animating, and verifying partial UML Models. In *MoDELS*, Eugene Syriani, Houari A. Sahraoui, Juan de Lara, and Silvia Abrahão (Eds.). ACM, 211–217. DOI: https://doi.org/10.1145/3365438.3410967

[110] Frédéric Jouault, Valentin Sebille, Valentin Besnard, Théo Le Calvar, Ciprian Teodorov, Matthias Brun, and Jérôme Delatour. 2021. AnimUML as a UML modeling and verification teaching tool. In *MODELS Companion*. IEEE, 615–619. DOI: https://doi.org/10.1109/MODELS-C53483.2021.00094

[111] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, and Ivan Porres. 2006. Model checking dynamic and hierarchical UML state machines. In *MoDeV^{2}a*, Benoît Baudry, David Hearnden, Nicolas Rapin, and Jörn Guy Süß (Eds.). 94–110.

[112] Jan Jürjens. 2002. Formal semantics for interacting UML subsystems. In *FMOODS (IFIP Conference Proceedings)*, Bart Jacobs and Arend Rensink (Eds.), Vol. 209. Kluwer, 29–43.

[113] Jan Jürjens. 2002. A UML statecharts semantics with message-passing. In *SAC*, Gary B. Lamont, Hisham Haddad, George A. Papadopoulos, and Brajendra Panda (Eds.). ACM, 1009–1013. DOI: https://doi.org/10.1145/508791.508987

[114] Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. 2015. LTSmin: High-performance language-independent model checking. In *TACAS (LNCS)*, Christel Baier and Cesare Tinelli (Eds.), Vol. 9035. Springer, 692–707. DOI: https://doi.org/10.1007/978-3-662-46681-0_61

[115] Soon-Kyeong Kim and David A. Carrington. 2002. A formal metamodeling approach to a transformation between the UML state machine and object-Z. In *ICFEM (LNCS)*, Chris George and Huaikou Miao (Eds.), Vol. 2495. Springer, 548–560. DOI: https://doi.org/10.1007/3-540-36103-0_55

[116] Barbara A. Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen G. Linkman. 2009. Systematic literature reviews in software engineering – A systematic literature review. *Inf. Softw. Technol.* 51, 1 (2009), 7–15. DOI: https://doi.org/10.1016/j.infsof.2008.09.009

[117] Alexander Knapp, Stephan Merz, and Christopher Rauh. 2002. Model checking timed UML state machines and collaborations. In *FTRTFT (LNCS)*, Werner Damm and Ernst-Rüdiger Olderog (Eds.), Vol. 2469. Springer, 395–416. DOI: https://doi.org/10.1007/3-540-45739-9_23

[118] Alexander Knapp and Till Mossakowski. 2017. UML interactions meet state machines—An institutional approach. In *CALCO (LIPIcs)*, Filippo Bonchi and Barbara König (Eds.), Vol. 72. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:15. DOI: https://doi.org/10.4230/LIPIcs.CALCO.2017.15

[119] Alexander Knapp and Till Mossakowski. 2018. Multi-view consistency in UML: A survey. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig (LNCS)*, Reiko Heckel and Gabriele Taentzer (Eds.), Vol. 10800. Springer, 37–60. DOI: https://doi.org/10.1007/978-3-319-75396-6_3

[120] Alexander Knapp, Till Mossakowski, Markus Roggenbach, and Martin Glauer. 2015. An institution for simple UML state machines. In *FASE (LNCS)*, Alexander Egyed and Ina Schaefer (Eds.), Vol. 9033. Springer, 3–18. DOI: https://doi.org/10.1007/978-3-662-46675-9_1

[121] Alexander Knapp, Markus Roggenbach, and Bernd-Holger Schlingloff. 2014. On the use of test cases in model-based software product line development. In *SPLC*, Stefania Gnesi, Alessandro Fantechi, Patrick Heymans, Julia Rubin, Krzysztof Czarnecki, and Deepak Dhungana (Eds.). ACM, 247–251. DOI: https://doi.org/10.1145/2648511.2648539

[122] Alexander Knapp and Jochen Wuttke. 2006. In *MoDELS Workshops (LNCS)*, Thomas Kühne (Ed.), Vol. 4364. Springer, 42–51. DOI: https://doi.org/10.1007/978-3-540-69489-2_6

[123] Jun Kong, Kang Zhang, Jing Dong, and Dianxiang Xu. 2009. Specifying behavioral semantics of UML diagrams through graph transformations. *J. Syst. Softw.* 82, 2 (2009), 292–306. DOI: https://doi.org/10.1016/j.jss.2008.06.030

[124] Vinay Kumar, Lalit Kumar Singh, and Anil Kumar Tripathi. 2017. Transformation of deterministic models into state space models for safety analysis of safety critical systems: A case study of NPP. *Annals Nucl. Energ.* 105 (July 2017), 133–143. DOI: https://doi.org/10.1016/j.anucene.2017.02.026

[125] Sabine Kuske. 2001. A formal semantics of UML state machines based on structured graph transformation. In *UML (LNCS)*, Martin Gogolla and Cris Kobryn (Eds.), Vol. 2185. Springer, 241–256. DOI: https://doi.org/10.1007/3-540-45441-1_19

[126] Sabine Kuske, Martin Gogolla, Ralf Kollmann, and Hans-Jörg Kreowski. 2002. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *iFM (LNCS)*, Michael J. Butler, Luigia Petre, and Kaisa Sere (Eds.), Vol. 2335. Springer, 11–28. DOI: https://doi.org/10.1007/3-540-47884-1_2

[127] Gihwon Kwon. 2000. Rewrite rules and operational semantics for model checking UML statecharts. In *UML (LNCS)*, Andy Evans, Stuart Kent, and Bran Selic (Eds.), Vol. 1939. Springer, 528–540. DOI: https://doi.org/10.1007/3-540-40011-7_39

[128] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. 2005. Formalizing UML models and OCL constraints in PVS. *Electron. Notes Theoret. Comput. Sci.* 115 (2005), 39–47. DOI: https://doi.org/10.1016/j.entcs.2004.09.027

[129] Vitus S. W. Lam and Julian A. Padget. 2004. Symbolic model checking of UML statechart diagrams with an integrated approach. In *ECBS*. IEEE Computer Society, 337–347. DOI: https://doi.org/10.1109/ECBS.2004.1316717

[130] Kevin Lano and David Clark. 2007. Direct semantics of extended state machines. *J. Obj. Technol.* 6, 9 (2007), 35–51. DOI: https://doi.org/10.5381/jot.2007.6.9.a2

[131] Kevin Lano, David Clark, and Kelly Androutsopoulos. 2004. UML to B: Formal verification of object-oriented models. In *iFM (LNCS)*, Eerke A. Boiten, John Derrick, and Graeme Smith (Eds.), Vol. 2999. Springer, 187–206. DOI: https://doi.org/10.1007/978-3-540-24756-2_11

[132] Kevin Lano, José Luiz Fiadeiro, and Luis Filipe Andrade. 2002. *Software Design Using Java 2*. Palgrave Macmillan.

[133] Kevin Lano, Kunxiang Jin, and Shefali Tyagi. 2021. Model-based testing and monitoring using AgileUML. In *ANT/EDI40 (Procedia Computer Science)*, Elhadi M. Shakshuki and Ansar-Ul-Haque Yasar (Eds.), Vol. 184. Elsevier, 773–778. DOI: https://doi.org/10.1016/j.procs.2021.04.012

[134] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *Int. J. Softw. Tools Technol. Transf.* 1, 1-2 (1997), 134–152.

[135] Diego Latella, István Majzik, and Mieke Massink. 1999. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Form. Asp. Comput.* 11, 6 (1999), 637–664. DOI: https://doi.org/10.1007/s001659970003

[136] Diego Latella, István Majzik, and Mieke Massink. 1999. Towards a formal operational semantics of UML statechart diagrams. In *FMOODS (IFIP Conference Proceedings)*, Paolo Ciancarini, Alessandro Fantechi, and Roberto Gorrieri (Eds.), Vol. 139. Kluwer.

[137] Hung Ledang and Jeanine Souquières. 2002. Contributions for modelling UML state-charts in B. In *iFM (LNCS)*, Michael J. Butler, Luigia Petre, and Kaisa Sere (Eds.), Vol. 2335. Springer, 109–127. DOI: https://doi.org/10.1007/3-540-47884-1_7

[138] David Lee and MMihalis Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123. DOI: https://doi.org/10.1109/5.533956

[139] Timothy C. Lethbridge, Andrew Forward, Omar Badreddin, Dusan Brestovansky, Miguel A. Garzón, Hamoud Aljamaan, Sultan Eid, Ahmed Husseini Orabi, Mahmoud Husseini Orabi, Vahdat Abdelzad, Opeyemi Adesina, Aliaa Alghamdi, Abdulaziz Algablan, and Amid Zakariapour. 2021. Umple: Model-driven development for open source and education. *Sci. Comput. Program.* 208 (2021), 102665. DOI: https://doi.org/10.1016/j.scico.2021.102665

[140] Jiexin Lian, Zhaoxia Hu, and Sol M. Shatz. 2008. Simulation-based analysis of UML statechart diagrams: Methods and case studies. *Softw. Qual. J.* 16, 1 (2008), 45–78. DOI: https://doi.org/10.1007/s11219-007-9020-9

[141] Johan Lilius and Iván Porres Paltor. 1999. Formalising UML state machines for model checking. In *UML (LNCS)*, Robert B. France and Bernhard Rumpe (Eds.), Vol. 1723. Springer, 430–445. DOI: https://doi.org/10.1007/3-540-46852-8_31

[142] Johan Lilius and Iván Porres Paltor. 1999. *The Semantics of UML State Machines.* Technical Report 273. Turku Centre for Computer Science.

[143] Johan Lilius and Iván Porres Paltor. 1999. vUML: A tool for verifying UML models. In *ASE*. IEEE Computer Society, 255–258. DOI: https://doi.org/10.1109/ASE.1999.802301

[144] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa, and Jin Song Dong. 2013. A formal semantics for complete UML state machines with communications. In *iFM (LNCS)*, Luigia Petre and Einar Broch Johnsen (Eds.), Vol. 7940. Springer, 331–346. DOI: https://doi.org/10.1007/978-3-642-38613-8_23

[145] Shuang Liu, Yang Liu, Jun Sun, Manchun Zheng, Bimlesh Wadhwa, and Jin Song Dong. 2013. USMMC: A self-contained model checker for UML state machines. In *ESEC/FSE*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 623–626. DOI: https://doi.org/10.1145/2491411.2494595

[146] Mass Soldal Lund, Atle Refsdal, and Ketil Stølen. 2010. Semantics of UML models for dynamic behavior - A survey of different approaches. In *International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems (LNCS)*, Holger Giese, Gabor Karsai, Edward Lee, Bernhard Rumpe, and Bernhard Schätz (Eds.), Vol. 6100. Springer, 77–103. DOI: https://doi.org/10.1007/978-3-642-16277-0_4

[147] Achraf Lyazidi and Salma Mouline. 2019. Formal verification of UML state machine diagrams using Petri nets. In *NETYS (LNCS)*, Mohamed Faouzi Atig and Alexander A. Schwarzmann (Eds.), Vol. 11704. Springer, 67–74. DOI: https://doi.org/10.1007/978-3-030-31277-0_5

[148] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. 1999. Statecharts via process algebra. In *CONCUR (LNCS)*, Jos C. M. Baeten and Sjouke Mauw (Eds.), Vol. 1664. Springer, 399–414. DOI: https://doi.org/10.1007/3-540-48320-9_28

[149] Andrea Maggiolo-Schettini, Adriano Peron, and Simone Tini. 2003. A comparison of statecharts step semantics. *Theoret. Comput. Sci.* 290, 1 (2003), 465–498. DOI: https://doi.org/10.1016/S0304-3975(01)00381-4

[150] Frédéric Mallet and Charles André. 2009. On the semantics of UML/MARTE clock constraints. In *ISORC*. IEEE Computer Society, 305–312. DOI: https://doi.org/10.1109/ISORC.2009.27

[151] Zohar Manna and Amir Pnueli. 1992. *The Temporal Logic of Reactive and Concurrent Systems – Specification.* Springer. DOI: https://doi.org/10.1007/978-1-4612-0931-7

[152] Marco Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. 1998. Modelling with generalized stochastic Petri nets. *SIGMETRICS Perform. Eval. Rev.* 26, 2 (1998), 2. DOI: https://doi.org/10.1145/288197.581193

[153] Fraco Mazzanti. 2009. *Designing UML Models with UMC*. Technical Report 2009-TR-043. Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo," ISTI-CNR. Retrieved from https://openportal.isti.cnr.it/data/2009/161078/2009_161078.pdf.

[154] Said Meghzili, Allaoua Chaoui, Martin Strecker, and Elhillali Kerkouche. 2017. On the verification of UML state machine diagrams to colored Petri nets transformation using Isabelle/HOL. In *IRI*, Chengcui Zhang, Balaji Palanisamy, Latifur Khan, and Sahra Sedigh Sarvestani (Eds.). IEEE Computer Society, 419–426. DOI: https://doi.org/10.1109/IRI.2017.63

[155] Said Meghzili, Allaoua Chaoui, Martin Strecker, and Elhillali Kerkouche. 2019. Verification of model transformations using Isabelle/HOL and Scala. *Inf. Syst. Front.* 21, 1 (2019), 45–65. DOI: https://doi.org/10.1007/s10796-018-9860-9

[156] Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. 2009. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS*. British Computer Society, 112–124.

[157] Stephen J. Mellor and Marc J. Balcer. 2002. *Executable UML - A Foundation for Model-Driven Architecture*. Addison-Wesley. Retrieved from http://www.informit.com/store/executable-uml-a-foundation-for-model-driven-architecture-9780201748048.

[158] José Merseguer, Javier Campos, Simona Bernardi, and Susanna Donatelli. 2002. A compositional semantics for UML state machines aimed at performance evaluation. In *DES*. IEEE Computer Society, 295–302. DOI: https://doi.org/10.1109/WODES.2002.1167702

[159] ChunYu Miao. 2011. Dynamic slicing research of UML statechart specifications. *J. Comput.* 6, 4 (2011), 792–798. DOI: https://doi.org/10.4304/jcp.6.4.792-798

[160] Huaikou Miao, Ling Liu, and Li Li. 2002. Formalizing UML models with object-Z. In *ICFEM (LNCS)*, Chris George and Huaikou Miao (Eds.), Vol. 2495. Springer, 523–534. DOI: https://doi.org/10.1007/3-540-36103-0_53

[161] Zoltán Micskei and Hélène Waeselynck. 2011. The many meanings of UML 2 sequence diagrams: A survey. *Softw. Syst. Model.* 10, 4 (2011), 489–514. DOI: https://doi.org/10.1007/s10270-010-0157-9

[162] Erich Mikk, Yassine Lakhnech, and Michael Siegel. 1997. Hierarchical automata as model for statecharts. In *ASIAN (LNCS)*, R. K. Shyamasundar and Kazunori Ueda (Eds.), Vol. 1345. Springer, 181–196. DOI: https://doi.org/10.1007/3-540-63875-X_52

[163] Khadija El Miloudi and Aziz Ettouhami. 2015. A multi-view approach for formalizing UML state machine diagrams using Z notation. *WSEAS Trans. Comput.* 14 (2015), 72–78.

[164] Peter D. Mosses. 2004. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*. LNCS, Vol. 2960. Springer. DOI: https://doi.org/10.1007/b96103

[165] Ben C. Moszkowski. 1985. A temporal logic for multilevel reasoning about hardware. *Computer* 18, 2 (1985), 10–19. DOI: https://doi.org/10.1109/MC.1985.1662795

[166] Muan Yong Ng and Michael J. Butler. 2002. Tool support for visualizing CSP in UML. In *ICFEM (LNCS)*, Chris George and Huaikou Miao (Eds.), Vol. 2495. Springer, 287–298. DOI: https://doi.org/10.1007/3-540-36103-0_31

[167] Muan Yong Ng and Michael J. Butler. 2003. Towards formalizing UML state diagrams in CSP. In *SEFM*. IEEE Computer Society, 138. DOI: https://doi.org/10.1109/SEFM.2003.1236215

[168] Iulian Ober, Susanne Graf, and Ileana Ober. 2004. Validation of UML models via a mapping to communicating extended timed automata. In *SPIN (LNCS)*, Susanne Graf and Laurent Mounier (Eds.), Vol. 2989. Springer, 127–145. DOI: https://doi.org/10.1007/978-3-540-24732-6_9

[169] Iulian Ober, Susanne Graf, and Ileana Ober. 2006. Validating timed UML models by simulation and verification. *Int. J. Softw. Tools Technol. Transf.* 8, 2 (2006), 128–145. DOI: https://doi.org/10.1007/s10009-005-0205-x

[170] Object Management Group. 2017. *Unified Modeling Language Superstructure, Version 2.5*. Retrieved from https://www.omg.org/spec/UML/2.5.1/PDF.

[171] Object Management Group. 2021. *Semantics of a Foundational Subset for Executable UML Models*. Retrieved from https://www.omg.org/spec/FUML/1.5/PDF.

[172] Sam Owre, John M. Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *CADE-11 (LNCS)*, Deepak Kapur (Ed.), Vol. 607. Springer, 748–752. DOI: https://doi.org/10.1007/3-540-55602-8_217

[173] Marian Petre. 2013. UML in practice. In *ICSE*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 722–731. DOI: https://doi.org/10.1109/ICSE.2013.6606618

[174] Marian Petre. 2014. "No shit" or "Oh, shit!": Responses to observations on the use of UML in professional practice. *Softw. Syst. Model.* 13, 4 (2014), 1225–1235. DOI: https://doi.org/10.1007/s10270-014-0430-4

[175] Carl Adam Petri. 1962. *Kommunikation mit Automaten*. Ph.D. Dissertation. Darmstadt University of Technology, Germany.

[176] Robert G. Pettit IV and Hassan Gomaa. 2000. Validation of dynamic behavior in UML using colored Petri nets. In *Proceedings of the UML 2000 Workshop: Dynamic Behaviour in UML Models: Semantic Questions*. 295–302.

[177] Amir Pnueli and M. Shalev. 1991. What is in a step: On the semantics of statecharts. In *TACS (LNCS)*, Takayasu Ito and Albert R. Meyer (Eds.), Vol. 526. Springer, 244–264. DOI: https://doi.org/10.1007/3-540-54415-1_49

[178] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hußmann. 2000. Analysing UML active classes and associated state machines - A lightweight formal approach. In *FASE (LNCS)*, T. S. E. Maibaum (Ed.), Vol. 1783. Springer, 127–146. DOI: https://doi.org/10.1007/3-540-46428-X_10

[179] Tobias Rosenberger, Saddek Bensalem, Alexander Knapp, and Markus Roggenbach. 2021. Institution-based encoding and verification of simple UML state machines in CASL/SPASS. In *WADT (LNCS)*, Markus Roggenbach (Ed.), Vol. 12669. Springer, 120–141. DOI: https://doi.org/10.1007/978-3-030-73785-6_7

[180] Tobias Rosenberger, Alexander Knapp, and Markus Roggenbach. 2022. An institutional approach to communicating UML state machines. In *FASE (LNCS)*, Einar Broch Johnsen and Manuel Wimmer (Eds.), Vol. 13241. Springer, 205–224. DOI: https://doi.org/10.1007/978-3-030-99429-7_12

[181] John Anil Saldhana, Sol M. Shatz, and Zhaoxia Hu. 2001. Formalization of object behavior and interactions from UML models. *Int. J. Softw. Eng. Knowl. Eng.* 11, 6 (2001), 643–673. DOI: https://doi.org/10.1142/S021819400100075X

[182] Thomas Santen and Dirk Seifert. 2006. *Executing UML State Machines*. Technical Report. Technische Universität Berlin, Softwaretechnik, Berlin, Germany.

[183] Thomas Santen and Dirk Seifert. 2006. TEAGER - Test automation for UML state machines. In *Software Engineering*, Bettina Biel, Matthias Book, and Volker Gruhn (Eds.). GI, 73–84.

[184] Vitaly Savicks, Colin F. Snook, and Michael J. Butler. 2010. *Animation of UML-B State-machines*. Technical Report 268261. University of Southampton, UK. Retrieved from https://eprints.soton.ac.uk/268261/1/TBFMsmAnim.pdf.

[185] Timm Schäfer, Alexander Knapp, and Stephan Merz. 2001. Model checking UML state machines and collaborations. *Electron. Notes Theoret. Comput. Sci.* 55, 3 (2001), 357–369. DOI: https://doi.org/10.1016/S1571-0661(04)00262-2

[186] Jens Schönborn. 2005. *Formal Semantics of UML 2.0 Behavioral State Machines*. Master's Thesis. Institute of Computer Science and Applied Mathematics, Technical Faculty, Christian-Albrechts-University of Kiel, Germany. Diplomarbeit.

[187] Dirk Seifert. 2008. Conformance testing based on UML state machines. In *ICFEM (LNCS)*, Shaoying Liu, T. S. E. Maibaum, and Keijiro Araki (Eds.), Vol. 5256. Springer, 45–65. DOI: https://doi.org/10.1007/978-3-540-88194-0_6

[188] Dirk Seifert. 2008. *An Executable Formal Semantics for a UML State Machine Kernel Considering Complex Structured Data*. Research Report inria-00274391. HAL. Retrieved from https://hal.inria.fr/inria-00274391.

[189] Bran Selic, Garth Gullekson, and Paul T. Ward. 1994. *Real-time Object-oriented Modeling*. Wiley.

[190] Wuwei Shen, Kevin J. Compton, and James Huggins. 2001. A UML validation toolset based on abstract state machines. In *ASE*. IEEE Computer Society, 315–318. DOI: https://doi.org/10.1109/ASE.2001.989819

[191] Wuwei Shen, Kevin J. Compton, and James Huggins. 2002. A toolset for supporting UML static and dynamic model checking. In *COMPSAC*. IEEE Computer Society, 147–152. DOI: https://doi.org/10.1109/CMPSAC.2002.1044545

[192] Colin F. Snook and Michael J. Butler. 2006. UML-B: Formal modeling and design aided by UML. *Trans. Softw. Eng. Methodol.* 15, 1 (2006), 92–122. DOI: https://doi.org/10.1145/1125808.1125811

[193] Colin F. Snook and Michael J. Butler. 2008. UML-B and Event-B: An integration of language and tools. In *SE*. Acta Press, 598–177.

[194] Colin F. Snook, Vitaly Savicks, and Michael J. Butler. 2010. Verification of UML models by translation to UML-B. In *FMCO (LNCS)*, Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.), Vol. 6957. Springer, 251–266. DOI: https://doi.org/10.1007/978-3-642-25271-6_13

[195] Marc Spielmann. 2000. Model checking abstract state machines and beyond. In *ASM (LNCS)*, Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele (Eds.), Vol. 1912. Springer, 323–340. DOI: https://doi.org/10.1007/3-540-44518-8_18

[196] J. Michael Spivey. 1992. *Z Notation - A Reference Manual* (2nd. ed.). Prentice Hall.

[197] Harald Störrle and Alexander Knapp. 2016. Discovering timing feature interactions with timed UML 2 interactions. In *MoDeVVa@MoDELS (CEUR Workshop Proceedings)*, Michalis Famelis, Daniel Ratiu, and Gehan M. K. Selim (Eds.), Vol. 1713. CEUR-WS.org, 10–19. Retrieved from http://ceur-ws.org/Vol-1713/MoDeVVa_2016_paper_2.pdf.

[198] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *CAV (LNCS)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 709–714. DOI: https://doi.org/10.1007/978-3-642-02658-4_59

[199] Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. 2011. A state/event-based model-checking approach for the analysis of abstract system properties. *Sci. Comput. Program.* 76, 2 (2011), 119–135. DOI: https://doi.org/10.1016/j.scico.2010.07.002

[200] Yann Thierry-Mieg and Lom-Messan Hillah. 2008. UML behavioral consistency checking using instantiable Petri nets. *Innov. Syst. Softw. Eng.* 4, 3 (2008), 293–300. DOI: https://doi.org/10.1007/s11334-008-0065-0

[201] Issa Traoré. 2000. An outline of PVS semantics for UML statecharts. *J. Univ. Comput. Sci.* 6, 11 (2000), 1088–1108. Retrieved from http://www.jucs.org/jucs_6_11/an_outline_of_pvs.

[202] Jan Trowitzsch and Armin Zimmermann. 2005. Real-time UML state machines: An analysis approach. In *Workshop on Object Oriented Software Design for Real Time and Embedded Computer Systems (Net.ObjectDays 2005)*.

[203] Jan Trowitzsch, Armin Zimmermann, and Günter Hommel. 2005. Towards quantitative analysis of real-time UML using stochastic Petri nets. In *IPDPS*. IEEE Computer Society. DOI: https://doi.org/10.1109/IPDPS.2005.441

[204] Ninh-Thuan Truong and Jeanine Souquières. 2006. Verification of UML model elements using B. *J. Inf. Sci. Eng.* 22, 2 (2006), 357–373. Retrieved from http://www.iis.sinica.edu.tw/page/jise/2006/200603_08.html.

[205] Andrew C. Uselton and Scott A. Smolka. 1994. A compositional semantics for statecharts using labeled transition systems. In *CONCUR (LNCS)*, Bengt Jonsson and Joachim Parrow (Eds.), Vol. 836. Springer, 2–17. DOI: https://doi.org/10.1007/978-3-540-48654-1_2

[206] Michael von der Beeck. 1994. A comparison of statecharts variants. In *FTRTFT (LNCS)*, Hans Langmaack, Willem P. de Roever, and Jan Vytopil (Eds.), Vol. 863. Springer, 128–148. DOI: https://doi.org/10.1007/3-540-58468-4_163

[207] Michael von der Beeck. 2002. A structured operational semantics for UML-statecharts. *Softw. Syst. Model.* 1, 2 (2002), 130–141. DOI: https://doi.org/10.1007/s10270-002-0012-8

[208] Michael von der Beeck. 2006. A formal semantics of UML-RT. In *MoDELS (LNCS)*, Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (Eds.), Vol. 4199. Springer, 768–782. DOI: https://doi.org/10.1007/11880240_53

[209] Ji Wang, Wei Dong, and Zhichang Qi. 2002. Slicing hierarchical automata for model checking UML statecharts. In *ICFEM (LNCS)*, Chris George and Huaikou Miao (Eds.), Vol. 2495. Springer, 435–446. DOI: https://doi.org/10.1007/3-540-36103-0_45

[210] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. 2009. SPASS version 3.5. In *CADE (LNCS)*, Renate A. Schmidt (Ed.), Vol. 5663. Springer, 140–145. DOI: https://doi.org/10.1007/978-3-642-02959-2_10

[211] Michael Westergaard. 2013. CPN tools 4: Multi-formalism and extensibility. In *Petri Nets (LNCS)*, José Manuel Colom and Jörg Desel (Eds.), Vol. 7927. Springer, 400–409. DOI: https://doi.org/10.1007/978-3-642-38697-8_22

[212] Claes Wohlin and Rafael Prikladnicki. 2013. Systematic literature reviews in software engineering. *Inf. Softw. Technol.* 55, 6 (2013), 919–920. DOI: https://doi.org/10.1016/j.infsof.2013.02.002

[213] Wing Lok Yeung, Karl R. P. H. Leung, Ji Wang, and Wei Dong. 2005. Improvements towards formalizing UML state diagrams in CSP. In *APSEC*. IEEE Computer Society, 176–184. DOI: https://doi.org/10.1109/APSEC.2005.70

[214] Sergio Yovine. 1997. KRONOS: A verification tool for real-time systems. *Int. J. Softw. Tools Technol. Transf.* 1, 1–2 (1997), 123–133. DOI: https://doi.org/10.1007/s100090050009

[215] Xuede Zhan. 2007. A formal testing framework for UML statecharts. In *SNPD*, Wenying Feng and Feng Gao (Eds.). IEEE Computer Society, 882–887. DOI: https://doi.org/10.1109/SNPD.2007.432

[216] Xuede Zhan and Huaikou Miao. 2004. An approach to formalizing the semantics of UML statecharts. In *ER (LNCS)*, Paolo Atzeni, Wesley W. Chu, Hongjun Lu, Shuigeng Zhou, and Tok Wang Ling (Eds.), Vol. 3288. Springer, 753–765. DOI: https://doi.org/10.1007/978-3-540-30464-7_56

[217] Shao Jie Zhang and Yang Liu. 2010. An automatic approach to model checking UML state machines. In *SSIRI*. IEEE Computer Society, 1–6. DOI: https://doi.org/10.1109/SSIRI-C.2010.11

[218] Karolina Zurowska and Jürgen Dingel. 2012. Symbolic execution of communicating and hierarchically composed UML-RT state machines. In *NFM (LNCS)*, Alwyn Goodloe and Suzette Person (Eds.), Vol. 7226. Springer, 39–53. DOI: https://doi.org/10.1007/978-3-642-28891-3_6

Received 22 July 2021; revised 26 August 2022; accepted 16 December 2022