On the Semantics of UML State Machines: Categorization and Comparison

Technical Report 2005-501

Michelle L. Crane and Juergen Dingel
School of Computing
Queen's University
Kingston, Ontario, Canada
crane@cs.queensu.ca
dingel@cs.queensu.ca

Abstract

Within Model Driven Development (MDD), state machines are a common mechanism for modelling behaviour. The development of a formal semantics for UML state machines continues to be a very active and important area of research, because the development of interoperating MDD tools requires a precise, unambiguous, yet readable account of the meaning of the diagrams. This paper is the result of a comparative literature survey on approaches to formally capture the semantics of UML state machines; it categorizes and compares 26 different approaches. As a primary categorization, we use the underlying formalism of the approaches, e.g., mathematical models, rewriting systems and translation approaches. We also compare the approaches along several secondary dimensions, such as coverage of state machine features, analysis and tool support. The purpose of this paper is to provide an overview of the state of the art in the field and to identify open questions and promising topics for future research.

Contents

Abstract i

Table of Contents ii

List of Tables iii

List of Figures iii

1 Introduction 1

2 Related Work 2

3 Categorization of Semantic Approaches 3

3.1 Mathematical Models 3
3.2 Rewriting Systems 4
3.3 Translation Approaches 5
3.4 Overlap in Reference Set 6

4 Comparison of Semantic Approaches 7

4.1 UML Coverage 8
4.2 Analysis 12
4.3 Tool Support 13

5 Future Work and Conclusions 14

References 16

Appendices 24

A Overlap (Detailed) 24

B Details on Approaches 26

List of Tables

1 Categorization of semantic approach references . . . 6
2 Sub-category coverage of UML features . . . 10
3 Tool support for analyis . . . 13
4 Legend mapping alphabetic references to numeric references . . . 25
5 UML state machine characteristic coverage legend . . . 26

List of Figures

1 Primary categorization of semantic approaches for UML state machines . . . 3
2 Overlap between primary sub-categories . . . 6
3 UML state machine features . . . 9
4 Histogram showing which versions of UML are supported . . . 12
5 Overlap between primary sub-categories (more detailed) . . . 24

1 Introduction

Model Driven Development (MDD) is a software development process that has been gaining in popularity. MDD focuses on the models of a software system, which are transformed into code. Within MDD, state machines are a common mechanism for modelling the behaviour of model elements. The Unified Modeling Language (UML) has become the *de facto* industry standard for general purpose modelling. UML is a visual modelling language with many diagram types. One of these, UML statechart diagrams, can be used to model intra-object behaviour, i.e., how individual model elements behave.

A common criticism of UML is that its semantics, especially with respect to behaviour, are inadequate. The UML 2.0 standard [63] contains much detailed prose about semantics; however, it does not adopt a formal notation to achieve a higher level of precision and clarity. Nonetheless, it is recognized that a formal, unambiguous, yet readable account of UML's semantics would be very beneficial for UML and MDD by, for instance, highlighting problems in the standard and enabling the development of powerful and interoperable analysis and transformation tools for UML models.

There exists much published research relating specifically to semantic approaches applied to UML in general and to state machines in particular. However, there has been little work published on the categorization or comparison of these approaches. The wide variety of approaches (using, for instance, Petri nets, labeled transition systems, concurrent regular expressions, rewriting, and specification languages such as Z) complicates this task, but also makes it all the more useful.

Providing an adequate, yet readable, formal account of UML represents a substantial challenge to the formal methods and semantics research communities and brings up several questions such as: 'Which formalisms are most suitable to handle feature X in diagram type Y?' 'Which formalisms support abstraction, analysis, and automation?' 'Which formalism is most accessible to average users?' This paper attempts to provide a somewhat comprehensive and comparative overview of the state of the art in formalizing the behaviour of UML state machines. Its goal is to facilitate future research on this topic by providing a starting point and by formulating some useful observations and open questions.

Under the aegis of the UML 2.0 Semantics Project,[1] this paper provides a categorization and comparison of 26 different approaches to the semantics of state machines. The approaches are grouped into primary categories, based on their underlying formalism. These groups of approaches are then compared against several secondary dimensions. The purpose of this paper is to provide a useful starting point with respect to learning about the semantics of state machines. Readers will be able to focus on a particular formalism, analysis goal, or state machine feature and determine which approaches, or types of approach, are most suitable to their needs.

This paper assumes familiarity with UML state machines and their features. Moreover, it will not provide any technical details on the surveyed formalisms themselves. Readers are

[1] http://www.cs.queensu.ca/stl/internal/uml2

encouraged to consult [63] for details about UML state machines and the referenced papers[2] for more information on the approaches.

This paper is organized as follows: Section 2 briefly discusses related work and why the semantics of classical statecharts cannot necessarily be applied to UML state machines. Section 3 divides the surveyed approaches into three primary categories. Section 4 then compares the semantic approaches along several secondary dimensions. Section 5 discusses conclusions and future work.

2 Related Work

UML statechart diagrams are an object-oriented variant of classical statecharts, which have evolved over the years since Harel first introduced the formalism [32]. Although much research has been devoted to the semantics of classical statecharts ([34],[55], and [68] among many others), these approaches cannot be simply applied to the semantics of UML statechart diagrams. Even though the step semantics of classical statecharts has evolved from a 'current step' to 'next step' philosophy,[3] there are other factors which make the two statechart formalisms less than perfectly compatible with each other. For instance, the run-to-completion assumption of UML states that an event can only be dispatched when the processing of the previous event has been completed [63]; classical statecharts still allow simultaneous processing of events. As another example, the implicit priority system between the two formalisms is inverted. In UML, the priority of conflicting transitions is determined by the source of the transitions and is 'bottom-up'. In classical statecharts, priority is determined by the overall scope of the transitions and is 'top-down'. These and other syntactic and semantic differences between the two formalisms are detailed in [22]. Due to these differences, existing semantic approaches for classical statecharts are not directly applicable to UML state machines.

As stated earlier, there are few, if any, publications devoted to the categorization and/or comparison of state machine semantic approaches. However, most research contains a 'related work' section, and several of these publications have been particularly useful. For instance, [23] provides an orthogonal division of related work, including categories such as 'level of UML coverage' and 'loose' vs. 'precise' semantics. [37] discusses semantic approaches for UML as a whole, with categories such as 'naive set-theoretic', 'meta-modelling' and 'translation'. [5] offers three categories of approach for applying a mathematical basis to object-oriented (OO) models: 'supplemental' (replacing informal notation with formal); 'OO-extension' (extending existing formal method to object-orientation); and 'method-integration' (integrating OO notation with appropriate formalism). Finally, as one of the

[2]In the main body of this paper, we have limited the number of references per approach to two; more references are listed in Appendix B.

[3]Initial versions of the semantics allowed changes that occurred in a given step to take effect in the same step; Harel subsequently changed this so that changes did not take effect until the next step [33]. This removed many of the paradoxes (such as instantaneous states and self-triggering transitions) discussed in [82].

more recent publications, [40] provides an excellent overview of many different approaches.

3 Categorization of Semantic Approaches

The state machine formalizations can be distinguished in many different ways: mathematical vs. non-mathematical, textual vs. graphical, theoretical vs. practical application, etc. Inspired by comparative surveys of semantics for programming languages, our initial intention was to categorize the approaches based on the type of semantics, e.g., denotational, operational or axiomatic; however, it turned out that an overwhelming majority of the approaches we surveyed (24/26) were operational in nature. We have therefore chosen to use the underlying formalism of the approaches as a primary dimension, with other interesting dimensions to be used for comparison in Section 4. There are three broad categories of underlying formalism, each with several sub-categories, as shown in Figure 1.

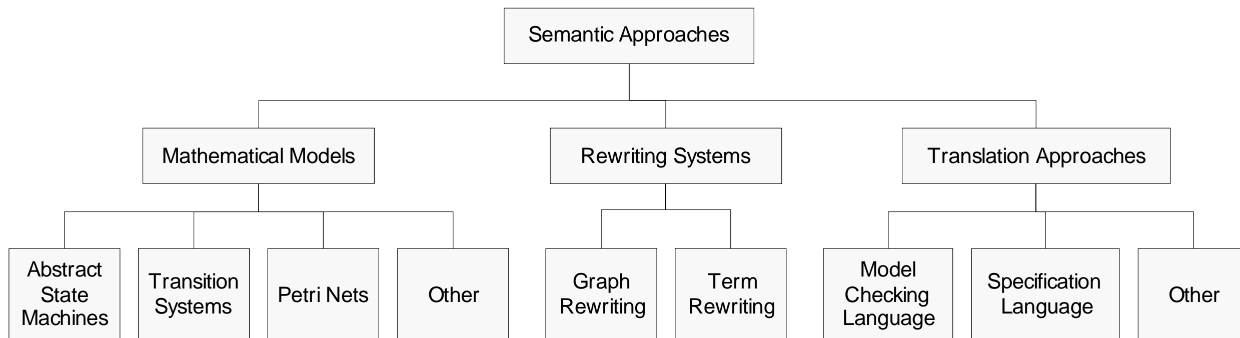

Figure 1: Primary categorization of semantic approaches for UML state machines

3.1 Mathematical Models

This category comprises semantic approaches which are based directly on standard mathematical concepts and notations. The advantage of using a mathematical notation is that it encourages precision and attention to detail, making it more likely that the resulting semantics is complete and unambiguous. In principle at least, the notation should be accessible to anybody with a standard mathematical background. However, it appears that most approaches in this category "fail to provide a high level of abstraction that can be properly understood" [79] by users. In other words, the user is often expected to digest a prohibiting amount of detail and notation.

Transition Systems In general, a transition system is a graph in which nodes represent states and edges represent transitions between them. There are different flavours of transition system, including Labeled Transition System (LTS), Kripke structure and Symbolic Transition System.

Abstract State Machines An Abstract State Machine (ASM) basically consists of a set of states and an iteratively applied update rule [41] and can be used, for instance, for the operational description of algorithms [40]. Although ASMs can be considered transition systems [12], we have kept the two formalisms separate, as in [79]. The syntax of ASMs is reminiscent of a simple imperative programming language which makes them quite accessible to users with a programming background. Analysis of ASMs is also possible with tool support.

Petri Nets Petri nets are a well-studied and intuitive formalism that is both graphical and mathematical. They consist of places, transitions, and arcs connecting them. Flow through nets makes use of the concept of tokens. The execution of a Petri net involves the firing of enabled transitions; tokens on places 'before' the transition are consumed and tokens on places 'after' the transition are created. Numerous editing and analysis tools are available and various extensions for different domains such as Generalized Stochastic Petri Nets for performance analysis [59] exist.

Other Mathematical This sub-category holds other mathematical approaches which do not make use of transition systems, ASMs or Petri nets. Example approaches include coalgebraic representations and basic sets-and-relations formalisms.

3.2 Rewriting Systems

This category is geared towards pure rewriting systems, such as graph rewriting or term rewriting. Rewriting systems can be considered as mathematical models [79] although we have chosen to keep them as a separate category. A rewrite system typically consists of a set of rewrite rules. Each rewrite rule consists of a left- and a right-hand side. The execution of a rewrite system involves the repeated application of the rules to some 'configuration'. In each application, an occurrence of the left-hand side of a rule in the configuration is replaced by the right-hand side. The execution terminates when no matching rule can be found anymore. Rewrite systems are also well-studied and various kinds of tool support are available.

Graph Rewriting Graph rewriting (also called graph transformation) "provides a mathematically precise and visual specification technique by combining the advantages of graphs and rules into a single computational paradigm" [79]. Graph rewriting approaches are a natural fit for state machines since there is no need to make the leap from graphical notation to textual/mathematical formalism.

Term Rewriting Term rewriting is a similar concept to graph rewriting, except that the rewrite rules are performed on terms rather than graphs. In the context of UML state machines, a term represents a configuration (e.g., set of active states) and a rewrite rule describes the relation between terms (e.g., transitions between state configurations).

3.3 Translation Approaches

This category contains approaches which rely on translating a UML state machine into some other formal language, such as a specification language, the input language to a model checker, or a programming language. Some of the approaches in this category can also be classified as either mathematical models or rewriting systems. What distinguishes this category from the other two is that these approaches are typically motivated not only by a desire to formalize but also to analyze automatically, e.g., with model checking, theorem proving, simulation, etc.

Model Checking Languages Model checking is a well-researched dynamic analysis method in which systems are modelled as finite state models. Temporal logic can then be used to define properties and the models are checked to verify whether these properties hold. Approaches listed in this sub-category typically have model checking as their final goal and they transform UML state machines into a language designed for such analysis, such as SMV or PROMELA/SPIN. A disadvantage of this sub-category is that the semantic model and the verification model are not the same, because model checking languages are not truly formal languages [21, 75]. For example, an approach may use LTS to define the semantic model and then translate that to PROMELA for the validation model.

Specification Languages Several approaches attempt to inject formalism into UML state machines by translating them into an already formalized specification language, such as Z or PVS. Other specification languages that we have seen mention of include: CASL [72], LOTOS [35], Object-Z [42], and RSL [58].

Other Translation This sub-category is a catch-all, currently containing one approach that translates state machines to concurrent regular expressions ([39]) and two approaches that translate state machines into axiomatic systems ([49] and [4]); these last two are the only non-operational semantic approaches that we discovered.

We are restricting ourselves to these three sub-categories; however, other appropriate sub-categories would be:

Programming Languages Approaches here would essentially generate code from UML statechart diagrams. [74, 44] is an example of this category, translating to Java.

Internal Representations Although little published research exists for this sub-category, it is nevertheless interesting. Here, approaches translate UML statechart diagrams to internal representations of tools, such as Rose RT, Rhapsody, etc. These two last are large, commercial, tools, but there exist many less well-known tools which allow for code generation, simulation, and analysis of UML statechart diagrams.

3.4 Overlap in Reference Set

The primary categorization provided here is not orthogonal. For instance, graph rewriting can be considered mathematically precise [79]. In addition, some approaches fit comfortably into more than one category, especially since several make use of more than one underlying formalism. Figure 2 shows the number of approaches in each sub-category and their overlaps. These overlaps represent our current set of references; additional references may cause new overlaps. Figure 5 in Appendix A shows a larger, more detailed, version of this overlap diagram.

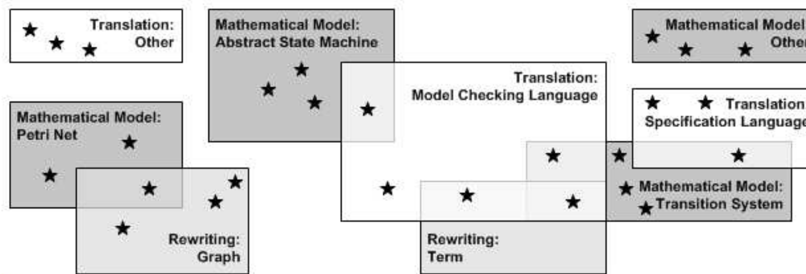

Figure 2: Overlap between primary sub-categories, based on current set of references. Each ★ represents a surveyed approach

Our set of references contains 35 primary approach-specific publications, covering 26 separate semantic approaches to formalizing UML state machines. These references are listed in Table 1, according to how they relate to the primary categorization. Note that some approaches are listed more than once, reflecting the overlap between sub-categories as shown in Figure 2. In addition to the primary publications (maximum of two per approach), several approaches also have secondary references; these are included in the approach details in Appendix B.

Table 1: Categorization of semantic approach references

Category/Sub-Category	**References** (one approach per cell)					
Mathematical Models						
Abstract State Machines	[12]	[21][75]	[40]	[41]		
Transition Systems	[23]	[26]	[30][51]	[47]	[72][70]	[83]
Petri Nets	[8]	[10][59]	[43]			
Other	[15][14]	[28]	[57]			
Rewriting Systems						
Graph Rewriting	[8]	[25]	[31][45]	[79]		
Term Rewriting	[47]	[53][54]				
Translation Approaches						
Specification Languages	[5]	[72]	[85][84]			
Model Checking Languages	[21][75]	[30][51]	[47]	[53][54]	[74][44]	
Other	[4]	[39]	[49]			

4 Comparison of Semantic Approaches

In addition to separating the semantic approaches along a primary dimension, i.e., underlying formalism, we also compared the approaches along several secondary dimensions. Due to space considerations, we will restrict ourselves to the following dimensions, which are of particular relevance to MDD:

UML Coverage All of the approaches are geared towards UML state machines; however, they vary widely as to which of the state machine features are actually covered.

Analysis Some approaches focus solely on providing a semantics for state machines; others provide a semantics and then continue on with analysis, such as model checking.

Tool Support Many approaches make use of or refer to some type of tool support. Some make use of pre-existing tools, e.g., for graph transformation or Petri net analysis, while others include the development of specific tools.

Obviously, there are many other dimensions along which the approaches can be compared. Additional dimensions of interest to MDD include:

Integration Some approaches focus specifically on statechart diagrams, while others are geared towards UML models in general, with only minor attention being paid to the state machines. Some of these approaches discuss model integration, i.e., how the meaning of different diagrams can be combined; others examine the issue of semantics for the entire set of diagrams from a higher level.

Multiple State Machines Most approaches dealt single state machines, i.e., no communication between objects was necessary. A very few approaches made use of communicating state machines, representing multiple objects.

Understandability In order for a user to fully make use of the benefits of a formalism, they are often required to learn the intricacies of that formalism. For example, Z specifications are all but incomprehensible to the novice user. Some approaches focus on formalisms which are more accessible to novices, such as ASMs or graph grammars. Other approaches use formalisms which require more expert knowledge, such as Z specifications, transition systems or axiomatic systems.

Additional dimensions not specifically related to MDD requirements include:

Intermediate Formalism Several approaches made use of an intermediate formalism or format, such as representing state machines as Extended Hierarchical Automata.

UML Version Which version of UML was used for an approach is relevant; various changes to the UML standard over the years might make an older approach more or less suitable with respect to the UML 2.0 specification.

Level of Detail Approaches can be compared in terms of how detailed their publications are—detailed enough to reproduce the work or very high level.

Examples Some approaches made very good use of illustrative examples; others had no examples at all. A few approaches exclusively used examples to illustrate their approach; others made use of examples in addition to a formal explanation.

Potential Impact This dimension simply refers to how well-cited a particular approach is. For example, some approaches are obviously key works, as they are continually cited by other authors, e.g., [51], [31] and [54]. A measure of popularity is no indication of the overall quality or merit of a particular approach, but it can be a measure of acceptance and impact within the community. In addition, the confidence in a particular approach can be affected by where its references are published. For instance, an unpublished manuscript may inspire less confidence than a publication in a well-known journal.

4.1 UML Coverage

The viability of a particular semantic approach should depend in part upon how well it addresses the features of UML state machines. Figure 3 lists most of the features of UML 2.0 state machines. This list includes most of the features of *behavioral state machines* according to [63] and gives a general idea of which features are best covered by our set of semantic approaches. The following features are not included: submachines, entry/exit pseudostates (submachines are syntactic sugar—they can be represented by replacing the submachine state with the machine that it represents); terminate pseudostates (new to UML 2.0); object creation/destruction and state machine extension (more complicated issues and not mentioned in many approaches); specific details with respect to event triggers, guards or actions (approaches either do not mention these details, or impose various restrictions, such as only call events, only signal events, only one event per transition, only one action per transition, guards with no conditions dependent on attributes, etc.).

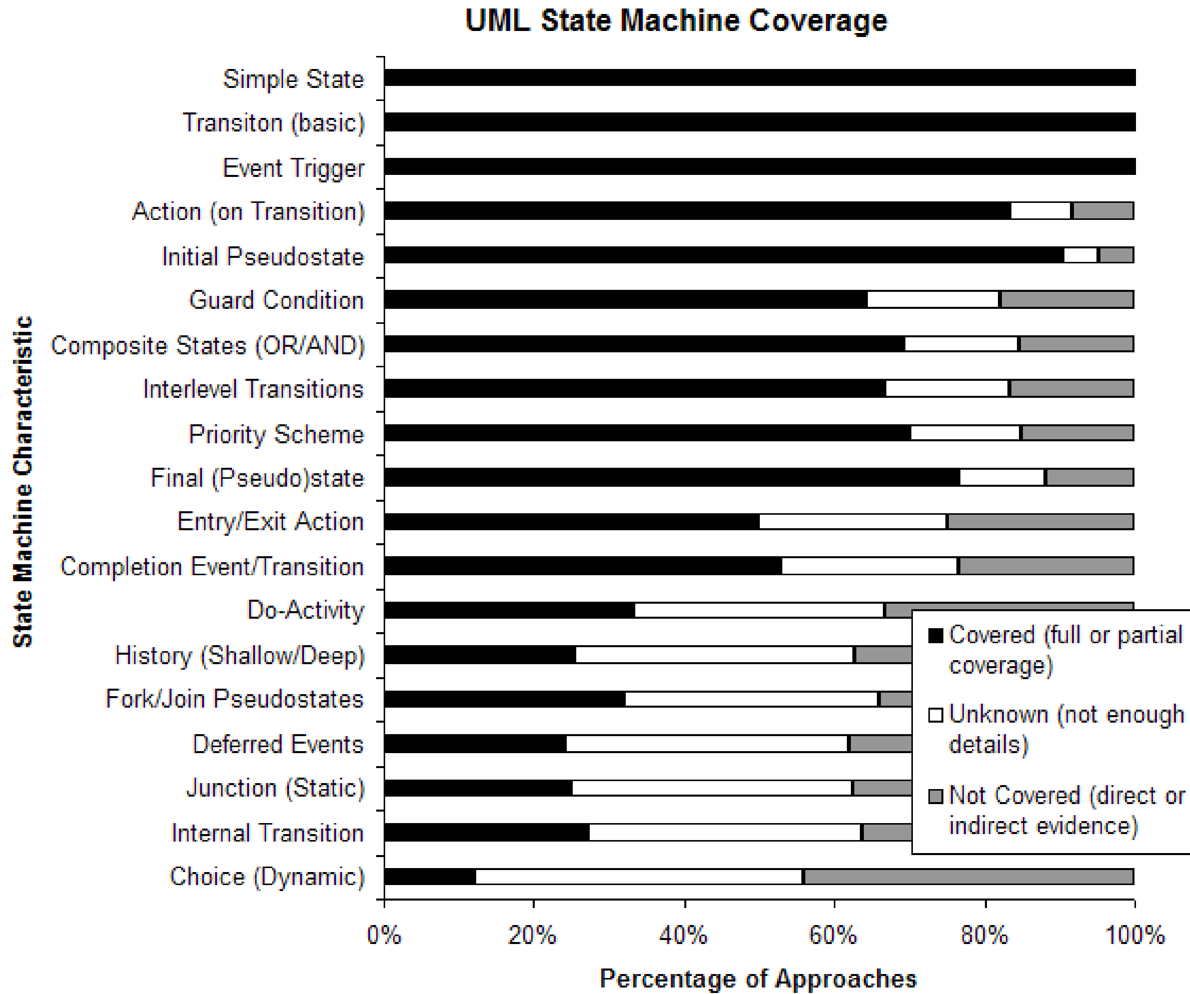

Figure 3: UML state machine features, ordered by coverage level. The features at the top are specifically covered by all or a majority of approaches; those at the bottom are specifically covered by few approaches. Most approaches are quite explicit with respect to which features they do or do not cover (shown by the black and grey segments in the figure. However, some approaches are too high-level or do not give enough detail (as shown by the white segments in the figure). Specific coverage information for each approach is detailed in Appendix B

The following conclusions can be drawn from the information summarized in Figure 3:

- While all approaches support the concept of simple states and basic transitions with some type of event trigger, not all approaches extend to more complicated transitions containing guards and/or actions.
- Almost 70% of approaches allow for composite states, i.e., AND- and OR- states. Interestingly enough, almost every approach which attempted to deal with OR- states also dealt with AND- states.

- 15% of approaches did not allow for any composite states, negating the state machine concepts of hierarchy and concurrency. Eliminating composite states also eliminates support of interlevel transitions (transitions between levels in a state machine hierarchy), completion events/transitions (triggered when a composite state has completed its execution) and history (only used with composite states).

- Almost 30% of approaches allowed for shallow and/or deep history; only one of these approaches supported shallow history but not deep history.

- The features in the bottom half of Figure 3 can be seen to represent the more 'complicated' features. Between 30% and 50% of approaches specifically do not support these features. Another 30% to 40% do not provide enough detail to determine whether or not they provide support; however, the probability is low. It should be noted that while some approaches claim to cover a particular feature, close inspection of the papers often casts some doubt. This may be due to a lack of detail or clarity in the description of the treatment of that feature. Consider, for instance, the discussion on page 11 with respect to dynamic choice.

In addition to examining how the approaches as a whole fared with respect to the coverage of UML state machine features, we were also interested in how families of approaches measured up. Instead of looking at all features however, we chose five of the more interesting features. Table 2 displays the results of this comparison of primary sub-categories vs. specific features. Four of the features are purely syntactic: composite states (OR- and/or AND-states), history pseudostates (shallow and/or deep), junction (static choice) and choice (dynamic choice). The fifth feature, priority, refers to whether or not the family of approaches deals with the concept of the implicit priority scheme of UML state machines. In short, the priority of conflicting transitions in UML state machines is determined by the source state of the transition (bottom-up). On the other hand, the classical statecharts formalism uses a top-down priority, based on the scope of the transition.

Table 2: Sub-category coverage of UML features

Sub-Category	Composite	History	Junction	Choice	Priority
ASM	●	◐	◐	◔	◕
Transition System	◕	◔	◔	○	◕
Petri Net	○	○	○	○	○
Graph Rewriting	◐	○	○	○	◐
Term Rewriting	●	◕	◔	○	●
Model Checking	●	●	◐	○	●
Specification	◐	◔	◐	◔	○

Legend

Symbol	Coverage (by Approaches)
○	0%
◔	> 0% and ≤ 25%
◐	> 25% and ≤ 50%
◕	> 50% and ≤ 75%
●	100%

The following conclusions can be determined by the information summarized in Table 2:

- None of the Petri net approaches addresses any of these five features. Further examination of these approaches indicates that they are approaches which formalize UML

models as a whole, of which the state machines are but a small part. It might be the case that researchers have simply not expanded the formalism to handle these more interesting features. Consequently, the lack of coverage by these approaches does not necessarily point to a fundamental limitation, but more likely to a conscious decision by the authors to emphasize the treatment of different diagram types rather than all state machine features.

- Neither graphical approaches, i.e., Petri nets and Graph Rewriting, seem particularly suitable for modelling these specific features. However, intuition suggests that they would be more suitable for modelling a visual language.

- While Petri net approaches provide the least amount of coverage, ASM approaches provide the most, with at least some of the ASM approaches claiming to cover each of these five features.

- Regardless of the type of approach used, dynamic choice is poorly addressed across the board. Only two approaches actually claim to support choice. However, one approach [40] does not allow for variables, which means that choice pseudostates are no different from junction pseudostates. The other approach [39] claims to support choice but there are not enough details to determine whether or not this choice is actually static or dynamic.

- Junction is another feature which is not particularly well covered by any family of approach. Many approaches do not refer to this feature at all. [12] and [54, 53] treat junction as syntactic sugar. [72] models a restricted junction, i.e., a junction is used to eliminate multiple transitions with the same trigger leaving one state. In addition, two approaches ([54, 53] and [74, 44]) draw junction as a diamond, i.e., the symbol for choice.[4]

- Priority is reasonably well covered, at least by most of the sub-categories. Two approaches ([26] and [30, 51]) make use of a parameterized priority scheme, which allows for either a bottom-up or top-down priority.

- There is exactly one approach [40] which handles all five of these features; however, as discussed above, the support for dynamic choice does not allow for variables. However, this approach is one of only a handful which support a great majority of the state machine features. In addition to [40], four other approaches cover a majority of the features: [12] (Abstract State Machine, missing fork/join); [28] (Other Math, missing junction/choice); [53, 54] (Term Rewriting and Model Checking Language, missing choice, junctions translated to simpler constructs); and [74, 44] (Model Checking Language, missing internal transitions, choice, syntactic difference with junction).

[4]It should be noted that junction is represented by a filled circle. Up until UML 2.0, choice was represented by an empty circle; it is now represented by a diamond. The diamond is used in activity diagrams; before UML 2.0, activity diagrams were considered a special type of statechart diagram.

Another way of looking at coverage is to consider the version of UML that approaches are covering. The histogram in Figure 4 indicates that a majority of approaches refer to UML 1.3, i.e., around 1999-2000. Although there are only two approaches that cover UML 2.0 ([28] and [85, 84]), many of the older approaches are still applicable because there have been few significant changes to the syntax and semantics of state machines. Minor changes include the replacement of 'event' by 'event occurrence' and 'action' by 'behavior'. A few new constructs have also been added, but they relate to the concept of sub-machines, which are not normally considered by the semantic approaches. The essence of state machines has remained unchanged.

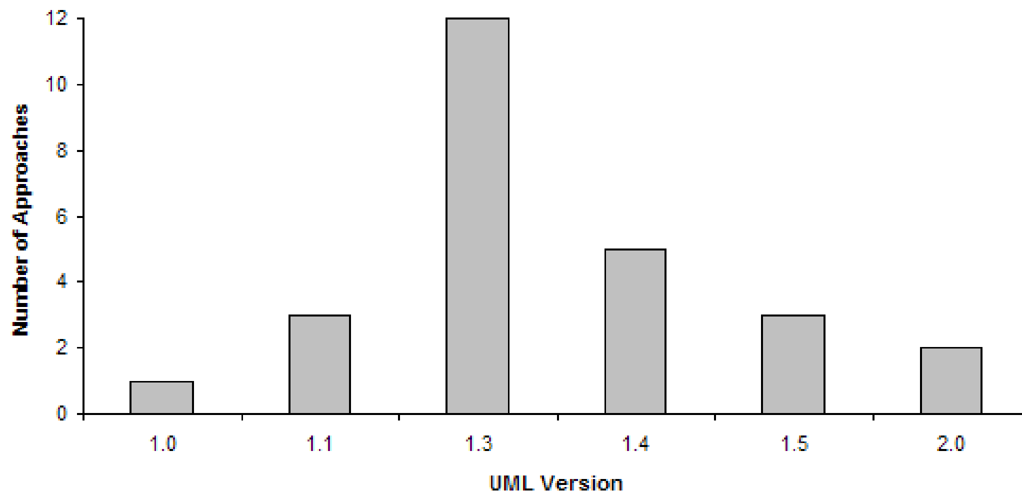

Figure 4: Histogram showing which versions of UML are supported

4.2 Analysis

In our reference set of approaches, the following types of analysis for UML state machines were discussed:

Syntax Checking Syntax checking can be performed against the UML meta-model, e.g., confirming that each transition has at least one source state and target state. Approach: [75].

Well-Formedness Checking Well-formedness (also known as static semantics) checking can be performed against the UML specification's Object Constraint Language (OCL) constraints, e.g., checking that transitions leaving pseudostates do not have triggers (events). Approach: [75].

Consistency Checking There are several ways of checking consistency. It is possible to check that a state machine satisfies assertions on its related class diagram, e.g., [4].

More common is consistency checking of a state machine against interaction diagrams, e.g., [10] and [74, 44]. Another form of consistency checking is suggested by [49], where the state machines of various interacting classes can be checked for consistency with each other.

Model Checking Model checking is a dynamic analysis, performed on finite state models of systems. In this context, it can be used to determine whether key properties (expressed in temporal logic) hold for all executions of a particular state machine, e.g., liveness, reachability, deadlock, fairness, etc. One advantage of model checking is its ability to return a counter-example when a property is violated. Another advantage is the fact that it is a mature field, with many well-designed tools. Approaches: [10], [30], [47], [53, 54], [74, 44], [75].

Animation Although not a formal analysis method, animation (simulation, execution) of a state machine can be used by the developer to ensure that a particular state machine behaves as expected. Approaches: [4], [40].

4.3 Tool Support

Although it is possible to perform some analysis of state machines manually (e.g., syntax checking), it is obviously much more convenient to automate analysis tasks. Several approaches make use of pre-existing tools, for instance, by extending or adapting a current tool. Several other approaches design their own specific tools or toolsets.

Table 3 lists which tools have been used by which approaches, and for which types of analysis. More information about how approaches uses these tools can be found in Appendix B.

Table 3: Tool support for analysis. Some approaches adapt or extend a pre-existing tool, while others create tools specific to their approach

Analysis	Tool	Based On/Using	Approach
Editing	Moses [27]		[40]
	JACK editor [13]		[30]
	F-Developer [4]		[4]
Syntax (vs. metamodel)	unnamed	XASM [3]	[75]
Well-formedness (vs. OCL)	unnamed	XASM [3]	[75]
Consistency	unnamed	GreatSPN-to-PROD [24]	[10]
	HUGO [74]	SPIN [36]	[74, 44]
	F-Verifier [4]	HOL [2]	[4]
Model checking	unnamed	SMV [56]	[75]
	unnamed	SMV [56]	[47]
	unnamed	PROD [78]	[10]
	JACK [13]/AMC		[30]
	HUGO [74]	SPIN [36], UPPAAL [50]	[74, 44]
	vUML [54]	SPIN [36]	[54, 53]
Animation/Simulation	Moses [27]		[40]
	F-Prototyper [4]	ML [65]	[4]

5 Future Work and Conclusions

This paper represents an attempt to categorize and compare the numerous approaches that exist for formalizing UML state machines. By no means can it be considered a complete overview; although we have made an attempt to cover a broad range of approaches, there are simply too many approaches in the research literature to cover them all. Moreover, our categorization is also not definitive. Alternative ways to group the approaches, perhaps using different dimensions, are conceivable.

In addition to the conclusions already listed, we suggest that several other conclusions can be drawn from this research:

- The translation of state machines to model checking languages is a popular approach. In this case, the result may not be as formal as a purely mathematical approach, but the end result is a system which can be analyzed automatically.
- Transition systems, and especially Abstract State Machines, are another popular formalism. These systems intuitively match the concept of state machines, i.e., states of some sort with transitions between them.
- While all approaches handle the basic concept of states and transitions, very few approaches handle the entire range of UML state machine features. In fact, there exists some doubt as to whether any one approach can formalize all of a state machine's features.
- Because UML is a visual notation, our original intuition was that graphical approaches, i.e., graph rewriting and Petri nets, would prove to be most useful. However, neither of these sub-categories fares well in terms of coverage with respect to UML state machine features.

Apart from the above direct observations, there are a number of open research questions and issues that can be distilled from the results of our study:

- Petri net approaches provide the least amount of coverage. Does this indicate an inherent limitation in the formalism (doubtful) or have the current approaches simply not taken the formalism as far as possible?
- Dynamic choice is the least covered state machine feature. Are there any approaches that successfully cover dynamic choice? Is this because it is difficult to capture formally, or because it is perceived as not interesting, i.e., infrequently used by modellers?
- As already mentioned, some of the state machine features are nothing more than syntactic sugar; e.g., entry/exit actions could be replaced with actions along the incoming/outgoing transitions (which would also eliminate the necessity for internal transitions). Exactly which features can be eliminated as syntactic sugar? What is the "core" of the UML state machine?

- The integration of different models is a crucial research issue for MDD. However, it currently seems poorly studied (with [15] as a notable exception). The question is, once suitable semantics for the different diagram types in UML have been found, how can they be integrated such that a single picture of the entire system emerges that encompasses all the different views? Which formalisms lend themselves to such an integration? How can the relationships between entities in different diagrams be specified?

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada, Communications and Information Technology Ontario, and the IBM Centers for Advanced Studies.

References

[1] Omega web site. `http://www-omega.imag.fr/`.

[2] *University of Cambridge Computer Laboratory: The HOL System Description, revised edition*, 1991.

[3] M. Anlauff. XASM- an extensible, component-based abstract state machines language. In *Proceedings of Abstract State Machine Workshop*, 2000.

[4] T. Aoki, T. Tateishi, and T. Katayama. An axiomatic formalization of UML models. In *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with UML 2001*, volume P-7 of *LNI*, pages 13–28. German Informatics Society, 2001.

[5] D.B. Aredo. Semantics of UML statecharts in PVS. Research Report 299, Department of Informatics, University of Oslo, 2000.

[6] ARTIS s.r.l. Artifex 3.1 - tutorial, 1994. Torino, Italy.

[7] L. Baresi, A. Orso, and M. Pezzè. Introducing formal methods in industrial practice. In *Proceedings of the 20th International Conference on Software Engineering*, pages 55–66. ACM Press, 1997.

[8] L. Baresi and M. Pezzè. On formalizing UML with high-level Petri nets. In *Proceedings of Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 271–300. Springer, 2001.

[9] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, pages 187–196, 2000.

[10] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable Petri net models. In *Proceedings of the 3rd International Workshop on Software and Performance (WOSP'02)*, pages 35–45. ACM Press, 2002.

[11] R.V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison Wesley, 2000.

[12] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2000.

[13] A. Bouali, S. Gnesi, and S. Larosa. The integration project for the JACK environment, Bull. *EATCS*, 54:207–223, 1994. `http://rep1.iei.pi.cnr.it/projects/JACK` (The Home Page of JACK).

[14] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. Systems, views and models of UML. In *The Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, 1998.

[15] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–366. Springer, 1997.

[16] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. The design of distributed systems - an introduction to FOCUS. Technical Report SFB 342/2/92 A, Technische Universität München, 1993. `http://www4.informatik.tu-muenchen.de/reports/TUM-I9202.ps.gz`.

[17] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN-1.7 - graphical editor and analyzer for timed and stochastic Petri nets. *Performance Evaluation*, 24(1-2):47–68, 1995.

[18] S. Christensen, J.B. Joergensen, and L.M. Kristensen. Design/CPN - a computer tool for coloured Petri nets. *Lecture Notes in Computer Science*, 1217, 1997.

[19] G. Ciardo, J. Muppala, and K. Trivedi. SPNP: Stochastic Petri net package. In *Proceedings of the 3rd International Workshop on Petri Nets and Performance*, pages 142–151, 1989.

[20] K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan, 2000.

[21] K. Compton, J.K. Huggins, and W. Shen. A semantic model for the state machine in the Unified Modeling Language. In *Dynamic Behaviour in UML Models: Semantic Questions, Workshop Proceedings, UML 2000 Workshop*. Ludwig-Maximilians-Universität München, Institut für Informatik, 2000.

[22] M.L. Crane and J. Dingel. UML vs. Classical vs. Rhapsody statecharts: Not all models are created equal. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005) (to appear)*, 2005.

[23] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in real-time UML. In *Formal Methods for Components and Objects*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer, 2003.

[24] S. Donatelli and L. Ferro. Validation of GSPN and SWN models through the PROD tool. In *Proceedings of the 12th International Conference on Modeling Techniques and Tools*, volume 2324 of *Lecture Notes in Computer Science*. Springer, 2002.

[25] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 2000)*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2000.

[26] R. Eshuis and R. Wieringa. Requirements-level semantics for UML statecharts. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems FMOODS*, pages 121–145. Kluwer Academic Publishers, 2000.

[27] R. Esser and J.W. Janneck. Moses: A tool suite for visual modeling of discrete-event systems. In *Symposia on Human-Centric Computing*, pages 272–279. IEEE Computer Society, 2001.

[28] H. Fecher, M. Kyas, and J. Schönborn. Semantic issues in UML 2.0 state machines. Technical Report 0507, Christian-Albrechts-Universität zu Kiel, 2005.

[29] V.K. Garg and M.T. Ragunath. Concurrent regular expressions and their relationship to Petri nets. *Theoretical Computer Science*, 96:285–304, 1992.

[30] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statecharts diagrams kernel and its extension to multicharts and branching time model-checking. *The Journal of Logic and Algebraic Programming*, 51:43–75, 2002.

[31] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In *Proceedings of the Workshop on Precise Semantics for Modelling Techniques (PSMT'98)*, pages 55–72. Technische Universität München, TUM-I9803, 1998.

[32] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[33] D. Harel. Some thoughts on statecharts, 13 years later. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 226–231. Springer, 1997.

[34] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: the STATEMATE Approach*. McGraw-Hill, 1998.

[35] Bogumila Hnatkowska and Zbigniew Huzar. Transformation of dynamic aspects of UML models into LOTOS behaviour expressions. *International Journal of Applied Mathematics and Computer Science*, 11(2):537–556, 2001.

[36] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[37] H. Hussmann. Loose semantics for UML, OCL. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology (IDPT 2002)*. Society for Design and Process Science, 2002.

[38] J.W. Janneck and P.W. Kutter. Mapping automata - simple abstract state machines. Technical Report 49, Computer Engineering and Networks Laboratory, ETH Zurich, 1998.

[39] S. Jansamak and A. Surarerks. Formalization of UML statechart models using concurrent regular expressions. In *Proceedings of the 27th Australasian Computer Science Conference (ACSC 2004)*, volume 26 of *CRPIT*, pages 83–88. Australian Computer Society, 2004.

[40] Y. Jin, R. Esser, and J.W. Janneck. A method for describing the syntax and semantics of UML statecharts. *Software and Systems Modeling*, 3(2):150–163, 2004.

[41] J. Jürjens. A UML statecharts semantics with message-passing. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02)*, pages 1009–1013. ACM Press, 2002.

[42] S-K. Kim and D. Carrington. A formal metamodeling approach to a transformation between the UML state machine and object-Z. In *Proceedings of the International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, pages 548–560. Springer, 2002.

[43] P. King and R. Pooley. Using UML to derive stochastic Petri net models. In *Proceedings of the 15th UK Performance Engineering Workshop (UKPEW'99)*, pages 45–56. Department of Computer Science, The University of Bristol, 1999.

[44] A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In *Proceeding of the 5th Workshop on Tools for System Design and Verification (FM-TOOLS 2002)*, Report 2002-11. Institut für Informatik, Universität Augsburg, 2002.

[45] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In *Proceedings of the 4th International Conference on the Unifed Modeling Language (UML 2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2001.

[46] S. Kuske, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28. Springer, 2002.

[47] G. Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In *Proceedings of the 3rd International Conference on the Unified Modeling

Language (UML 2000), volume 1939 of *Lecture Notes in Computer Science*, pages 528–540. Springer, 2000.

[48] K. Lano. Logical specification of reactive and real-time systems. *Journal of Logic and Computation*, 8(5):679–711, 1998.

[49] K. Lano, J. Bicarregui, and A. Evans. Structured axiomatic semantics for UML models. In *Rigorous Object-Oriented Methods (ROOM 2000)*. Electronic Workshops in Computing (eWiC), 2000.

[50] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[51] D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[52] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 331–347. Kluwer, 1999.

[53] J. Lilius and I. Porres Paltor. Formalising UML state machines for model checking. In *Proceedings of The Unified Modeling Languge (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer, 1999.

[54] J. Lilius and I. Porres Paltor. vUML: A tool for verifying UML models. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 255–258. IEEE Computer Society, 1999.

[55] A. Maggiolo-Schettini and A. Peron. A graph rewriting framework for statecharts semantics. In *Proceedings of the International Conference on Graph Grammars (GRAGRA)*, volume 1996 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1996.

[56] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

[57] S. Meng, Z. Naixiao, and L.S. Barbosa. On semantics and refinement of UML statecharts: a coalgebraic view. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 164–173. IEEE Computer Society, 2004.

[58] Sun Meng, Zhang Naixiao, and Bernhard K. Aichernig. The formal foundations in RSL for UML statechart diagrams. Technical Report 299, The United Nations University International Institute for Software Technology (UNU/IIST), 2004.

[59] J. Merseguer, J. Campos, S. Bernardi, and S. Donatelli. A compostional semantics for UML state machines aimed at performance evaluation. In *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 295–302. IEEE Computer Society Press, 2002.

[60] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Proceedings of the Asian Computing Science Conference (ASIAN '97)*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 1997.

[61] P.D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In *Proceedings of TAPSOFT '97*, volume 1214 of *Lecture Notes in Computer Science*. Springer, 1997.

[62] OMG. OMG UML specification. Technical Report ad/99-06-08, Object Management Group, 1997. Version 1.3.

[63] OMG. UML 2.0 superstructure specification. Technical Report ptc/04-10-02, Object Management Group, 2004.

[64] S. Owre, N. Shankar, J. Rushby, and D.W. Stringer-Calvert. PVS language reference, version 2.3. Technical report, Computer Science Laboratory, 1999.

[65] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[66] M. Pezzè. Cabernet: A customizable environment for the specification ande analysis of real-time systems. Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 1994.

[67] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.

[68] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS'91)*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer, 1991.

[69] G. Reggio. An "extreme" approach to metamodelling. Technical report, DISI, Università di Genova, Italy, 2002.

[70] G. Reggio. Metamodelling behavioural aspects: the case of the UML state machines (complete version). Technical report, DISI - Università di Genova, Italy, 2002.

[71] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. A CASL formal definition of UML active classes and associated state machines. Technical Report DISI-TR-99-16, DISI-Università di Genova, Italy, 1999.

[72] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines - a lightweight formal approach. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 2000.

[73] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249:3–80, 2000.

[74] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. In *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier, 2001. Issue 3.

[75] W. Shen, K. Compton, and J. K. Huggins. A toolset for supporting UML static and dynamic model checking. In *Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC 2002)*, pages 147–152. IEEE Computer Society, 2002.

[76] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[77] I. Traoré. An outline of PVS semantics for UML statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, 2000.

[78] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Techical Report Series B, Number 13, Helsinki University of Technology, 1995.

[79] D. Varró. A formal semantics of UML Statecharts by model transition systems. In *Proceeding of the 1st International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 378–392. Springer, 2002.

[80] D. Varró. Towards symbolic analysis of visual modelling languages. In *Proceedings of the International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *Electronic Notes in Theoretical Computer Science*, pages 57–70. Elsevier, 2002.

[81] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, 2002.

[82] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 1994.

[83] M. von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.

[84] X. Zhan and H. Miao. An approach to formalizing the semantics of UML statecharts. In *Proceeding of the 23rd International Conference on Conceptual Modeling (ER 2004)*, volume 3288 of *Lecture Notes in Computer Science*, pages 753–765. Springer, 2004.

[85] X. Zhan, H. Miao, and L. Liu. Formalizing the semantics of UML statecharts with Z. In *Proceedings of the 4th International Conference on Computer and Information Technology (CIT '04)*, pages 1116–1121. IEEE Computer Society, 2004.

A Overlap (Detailed)

Similar to Figure 2, Figure 5 shows the overlap between primary sub-categories; however, this figure notes exactly which references populate which overlapping sections. This overlap diagram also contains more information, such as a 'Translation to: Programming Language' primary sub-category, as well as a 'Meta-Modelling' secondary dimensions. Alphabetized references are used in the figure; their correspondence with the numerical references used in this document is listed in Table 4.

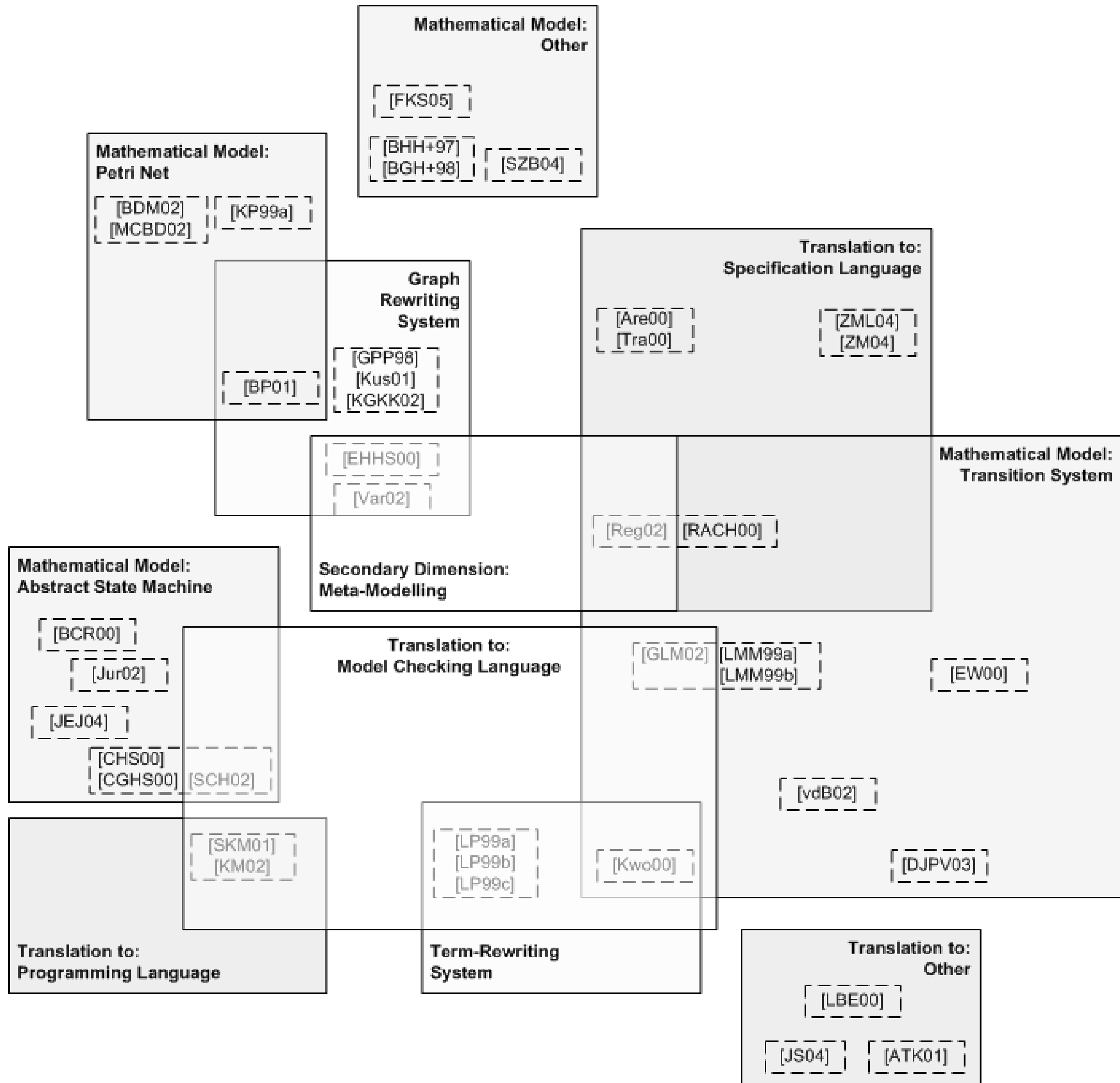

Figure 5: Overlap between primary sub-categories, based on current set of references. Approaches are indicated with alphabetic references; Table 4 shows the mapping between these alphabetic references and the numeric references used elsewhere in this document

Table 4: Legend mapping alphabetic references to numeric references

Alphabetic	Numeric	Alphabetic	Numeric
[Are00][Tra00]	[5][77]	[JEJ04]	[40]
[ATK01]	[4]	[JS04]	[39]
[BCR00]	[12]	[Jur02]	[41]
[BDM02][MCBD02]	[10][59]	[KP99]	[43]
[BHH+97][BGH+98]	[15][14]	[Kwo00]	[47]
[BP01]	[8]	[LBE00]	[49]
[CHS00][CGHS00][SCH02]	[21][20][75]	[LP99a][LP99b]	[54][53]
[DJPV03]	[23]	[RACH00][Reg02]	[72][70]
[EHHS00]	[25]	[SKM01][KM02]	[74][44]
[EW00]	[26]	[SZB04]	[57]
[FKS05]	[28]	[Var02]	[79]
[GLM02][LMM99a][LMM99b]	[30][51][52]	[vdB02]	[83]
[GPP98][Kus01][KGKK02]	[31][45][46]	[ZML04][ZM04]	[85][84]

B Details on Approaches

The material presented thus far is a high-level compilation of details gleaned from the publications for each approach. This appendix presents each approach in more detail, providing a brief summary of the approach, as well as a detailed examination of how each approach handles the UML state machine characteristics listed in Figure 3. The legend in Table 5 explains the symbols used to describe how well each characteristic is supported.

Table 5: UML state machine characteristic coverage legend

Symbol	Description
●	supported, with little or no difference from UML 2.0 specification
⊙	supported, with considerable difference from UML 2.0 specification
⊗	definitely not supported (direct evidence)
⊘	presumably not supported (indirect evidence)
○	unknown; not enough evidence to determine

The following points should be noted:

1. The level of coverage has been determined based on the information in the cited references. Where more than one reference is provided, the 'best' coverage level is used.
2. The provided information includes the following:
   - List of publications: the first one or two publications are primary references and have been used in the categorization and comparison presented in the main document. Supporting, or secondary, publications are listed in italics.
   - Brief summary: a high-level synopsis of the approach is presented; readers are encouraged to consult the cited publications for more information.
   - Coverage table: the level of coverage for each UML state machine characteristic is presented.
3. Support (●), or partial support (⊙), was either explicitly noted in the publication, or implicitly derived from provided examples.
4. Definite lack of support (⊗) was explicitly noted in the publication.
5. Assumed lack of support (⊘) was derived from information in the publication. This determination is subjective and is based on the author's opinion at the time of reading.
6. A lack of information (○) does not imply that the characteristic is not covered; it simply means that there is insufficient evidence in the publication to make a determination. Several publications are written at such a high level that absolutely no coverage information could be inferred.

An Axiomatic Formalization of UML Models	[4]
T. Aoki and T. Tateishi and T. Katayama	

This approach suggests that some ambiguity in UML models is caused by the fact that multiple models (i.e., diagrams) are built to describe a single system. The authors recommend an axiomatic approach that allows them to check consistency between diagrams, for instance, between a class diagram and a statechart diagram. Specifically, they make use of a distinct unification step in the analysis phase where a class diagram and statechart diagram are linked with a *unification mapping*. The behaviour of the state machine can be checked against assertions on the class diagram using a "typical invariant assertion method" [4]. A system called F-Developer provides support to automatically generate the axiomatic system.

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	○	
orthogonal (AND)	○	
do-activity	○	
deferred events	○	
Pseudostates		
initial	○	
final	○	
fork/join	○	
history (shallow & deep)	○	
junction	○	
choice	○	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	⊙	only one action per transition but can cause several events to occur
priority scheme	○	
interlevel transitions	○	
Miscellaneous		
completion event/transition	○	

Semantics of UML Statecharts in PVS	[5]
D.B. Aredo	
An outline of PVS semantics for UML statecharts	[77]
I. Traoré	

This approach provides a general schema for translating UML state machines into PVS-SL, the specification language of the PVS [64] framework. The translation takes into account the abstract syntax of state machines, as well as their well-formedness constraints. The advantage of using the PVS framework is that it includes tools for rigorous analysis of PVS specifications, including type checking, theorem proving, and even model checking. Although short, this paper suggests a PVS-SL specification of a reasonable subset of state machine features and introduces the concepts of formalizing the semantics of these machines.

Feature		Comment
States		
entry/exit actions	●	
internal transitions	⊗	according to author, should be able to extend
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	●	
Pseudostates		
initial	●	
final	●	
fork/join	●	
history (shallow & deep)	●	
junction	●	
choice	●	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	⊗	according to author, should be able to extend
interlevel transitions	⊗	according to author, should be able to extend
Miscellaneous		
completion event/transition	●	

On Formalizing UML with High-Level Petri Nets	[8]
L. Baresi and M. Pezzè	

[8] extends the research in [7] by modifying the Customized Rules Approach to an object oriented notation, specifically UML. The concept of customized rules is that users ascribe desired semantics in a familiar notation and benefit from a formal simulation and analysis engine, which they do not necessarily need to master. The paper illustrates through an example the automatic mapping from a UML model (of which statecharts are simply one part) to high-level Petri nets. Because UML is a graphical notation, graph grammars are employed to define its syntax and semantics. With respect to state machines, state are modeled as Petri net places, while transitions are modeled as Petri net transitions. The paper contains a detailed example based on the dining philosopher's problem; it is modeled by various UML diagrams and then mapped to a high-level Petri net. Various tools for Petri nets can be used for execution [18, 6] and analysis and model checking [18, 66]. The analyses that can be performed on these Petri nets include: absence of deadlocks, boundedness, mutual exclusion, fairness, net execution, reachability, and model checking. Ultimately, the user decides what properties are desirable for the UML specification. These properties are then defined in terms of the Petri net. The net is then analyzed and the results are translated back to UML notation for the user to examine.

Note: article too high level to determine any coverage information.

Feature		
States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	○	
orthogonal (AND)	○	
do-activity	○	
deferred events	○	
Pseuodstates		
initial	○	
final	○	
fork/join	○	
history (shallow & deep)	○	
junction	○	
choice	○	
Transitions		
event trigger	○	
guard condition	○	
action (behavior)	○	
priority scheme	○	
interlevel transitions	○	
Miscellaneous		
completion event/transition	○	

From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models	[10]
S. Bernardi and S. Donatelli and J. Merseguer	
A Compositional Semantics for UML State Machines Aimed at Performance Evaluation	[59]
J. Merseguer and J. Campos and S. Bernardi and S. Donatelli	

This approach transforms state machines into General Stochastic Petri Nets (GSPNs) [59] and sequence diagrams into labeled GSPNs (LGSPNs) [10]. It then combines these into an analyzable GSPN that includes the behaviour of the state machines and their interactions based on the sequence diagrams [10]. This GSPN can be analyzed for correctness (e.g., determine if there is a sequence such that the state machine can complete execution) and performance analysis, based on the specific system being modelled (e.g., time to reseat after a fault occurs). The following tools are used for this approach: GreatSPN is a tool for modelling, validating and performance evaluation of distributed systems using Generalized Stochastic Petri Nets [17]. The PROD [78] model checker is a reachability analyzer/model checker for Predicate/Transition nets. Finally, the GreatSPN-to-PROD tool [24] translates GSPNs into high-level Petri nets that are the input of the PROD checker.

States		
entry/exit actions	●	
internal transitions	●	
sequential (OR)	⊗	
orthogonal (AND)	⊗	
do-activity	●	
deferred events	●	
Pseuodstates		
initial	●	
final	●	
fork/join	⊗	
history (shallow & deep)	⊗	
junction	⊗	
choice	⊗	
Transitions		
event trigger	●	
guard condition	⊗	only static analysis, so don't need to evaluate guards
action (behavior)	●	
priority scheme	○	
interlevel transitions	⊗	since no composite states
Miscellaneous		
completion event/transition	⊗	

Modeling the Dynamics of UML State Machines	[12]
E. Börger and A. Cavarra and E. Riccobene	

This approach makes use of Abstract State Machines (ASMs) to represent state machines and takes a very detailed look at the more complicated parts of UML state machines, such as the event handling, run-to-completion step, internal (do-) activities, etc. Its best contributions are its attempts at making explicit the various 'semantic variation points' in the UML standard, as well as a detailed explanation on representing state machines as ASMs.

States		
entry/exit actions	●	
internal transitions	●	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	●	
Pseuodstates		
initial	●	
final	●	
fork/join	⊗	can be defined in terms of more basic constructs; allow only incoming (outgoing) transitions that end (start) on the boundary
history (shallow & deep)	●	
junction	●	
choice	○	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	●	

Systems, Views and Models of UML	[14]
R. Breu and R. Grosu and F. Huber and B. Rumpe and W. Schwerin	
Towards a Formalization of the Unified Modeling Langauge	[15]
R. Breu and U. Hinkel and C. Hofmann and C. Klein and B. Paech and B. Rumpe and V. Thurner	

This approach looks at the whole of UML, using a mathematical formalism based on the theory of streams and stream processing functions [16]. In order to adapt this technique to object-orientation, the authors augment the framework with system models [15]. This approach presents a very high view of the formalism of UML and does not offer many specific details. It is included here mainly because of its ancestral relation to the system model prepared as part of the UML 2 Semantics Project.

Note: article too high level to determine any coverage information.

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	○	
orthogonal (AND)	○	
do-activity	○	
deferred events	○	
Pseuodstates		
initial	○	
final	○	
fork/join	○	
history (shallow & deep)	○	
junction	○	
choice	○	
Transitions		
event trigger	○	
guard condition	○	
action (behavior)	○	
priority scheme	○	
interlevel transitions	○	
Miscellaneous		
completion event/transition	○	

A Semantic Model for the State Machine in the Unified Modeling Language	[21]
K. Compton and J. Huggins and W. Shen	
A Toolset for Supporting UML Static and Dynamic Model Checking	[75]
W. Shen and K. Compton and J. Huggins	
An Automatic Verification Tool for UML	[20]
K. Compton and Y. Gurevich and J. Huggins and W. Shen	

[21] presents an Abstract State Machine (ASM) model of the UML state machine, which is used by [75] to conduct both static and dynamic analysis of state machines. The latter publication discussions a prototype toolset based on XASM [3], an extensible ASM. This toolset can be used to perform well-formedness checking. In addition, the toolset can be used to convert a state machine into an ASM specification and then perform model checking (based on the SMV [56] model checker). One of the most interesting aspects of this approach is that it makes use of the XMI format so that the toolset is compatible with commercial UML CASE tools.

States		
entry/exit actions	●	
internal transitions	⊗	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	⊘	
Pseuodstates		
initial	●	
final	○	
fork/join	⊘	
history (shallow & deep)	⊘	
junction	⊘	
choice	⊘	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	⊘	discusses only exit and entry actions along a transition
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	●	

Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML W. Damm and B. Josko and A. Pneuli and A. Votintseva	[23]

This approach defines semantics for a specific subset of UML geared towards real-time concepts, such as dynamic object creation/destruction, dynamically changing communication topologies, etc. [23]. The semantics for this 'kernel language' associates each model with a symbolic transition system. Although the article is quite detailed with respect to its approach, it is also quite useful in terms of its 'related works' section, which contains several orthogonal ways of categorizing semantics approaches. Another significant contribution to this work is its commercial viability; the Omega project builds on the semantic foundations in this paper; its goal is to "provide formal foundations, methods and tools for formal specifications and verification of real-time systems within UML" [1]

Note: article too high level to determine any coverage information.

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	○	
orthogonal (AND)	○	
do-activity	○	
deferred events	○	
Pseudostates		
initial	○	
final	○	
fork/join	○	
history (shallow & deep)	○	
junction	○	
choice	○	
Transitions		
event trigger	○	
guard condition	○	
action (behavior)	○	
priority scheme	○	
interlevel transitions	○	
Miscellaneous		
completion event/transition	○	

Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML G. Engels and J.H. Hausmann and R. Heckel and S. Sauer	[25]

The authors pursue a metamodelling approach, where the static meta model (based on class diagrams) is extended by a dynamic model specified with a basic form of collaboration diagrams. The collaboration diagrams are used to specify the operational semantics of the state machines. By using diagrams with which the user is already familiar, this approach can be considered more accessible. These collaboration diagrams are "formalized as graph transformation rules for specifying the operational semantics" [25]. The use of graph rewriting allows for a mathematical rigorous, yet more user-friendly, formal semantics. Although the paper provides a detailed explanation, along with excellent examples, the approach covers a very basic subset of UML state machines.

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	⊗	composite states not considered
orthogonal (AND)	⊗	composite states not considered
do-activity	○	
deferred events	○	
Pseuodstates		pseudostates not considered
initial	○	
final	○	
fork/join	⊗	
history (shallow & deep)	⊗	
junction	⊗	
choice	⊗	
Transitions		
event trigger	⊙	call events only
guard condition	⊗	
action (behavior)	⊙	call actions only
priority scheme	○	
interlevel transitions	⊗	
Miscellaneous		
completion event/transition	⊘	assume no, since only call events considered

Requirements-level Semantics for UML Statecharts	[26]
R. Eshuis and R. Wieringa	

This approach makes use of Labeled Transition Systems (LTSs)/Kripke structures to define a formal semantics for UML state machines. This is the only approach in our reference set that specifically focuses on requirements-level vs. implementation level semantics. This focus enables the authors to make certain assumptions about the system being modelled, for instance, perfect technology, instantaneous (lossless) communication, etc. However, they also make the assumption that the system reacts to *all* events in put as opposed to reacting to one event; this assumption seems to defy the UML run-to-completion assumption. In addition, this approach also allows for instantaneous states, which also does not conform to the UML specification. That said, the greatest contribution of this article may be the fact that they do distinguish between the concept of requirements-level vs. implementation-level semantics. The latter requires the assumptions of imperfect technology, non-instantaneous (lossy) communication, actions taking time and limited concurrency [26]; these concepts represent the future of research in this field. Very few of the approaches survey handle any of these assumptions, let alone all of them. It could be said that the assumptions of implementation-level semantics represent what needs to be researched in this field.

States		
entry/exit actions	●	
internal transitions	○	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊗	according to authors, omitted to simplify; could be added
deferred events	⊗	according to authors, omitted to simplify; could be added
Pseuodstates		
initial	●	
final	●	
fork/join	○	
history (shallow & deep)	⊗	according to authors, omitted to simplify; could be added
junction	○	
choice	⊗	according to authors, omitted to simplify; could be added
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	●	

Semantic Issues in UML 2.0 State Machines	[28]
H. Fecher and M. Kyas and J. Shönborn	

This paper presents a very detailed examination of a pre-release version of the UML 2.0 specification of state machines. State machines are basically defined in terms of sets and relations and most features are considered. The major contribution of this work is that through such a detailed examination of the specification, they are able to ask interesting questions. For example, they propose several state machines with conflicting transitions for which it is unclear how to assign priority using the current specification.

States		
entry/exit actions	●	
internal transitions	●	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	●	
Pseuodstates		
initial	●	
final	●	
fork/join	●	
history (shallow & deep)	●	
junction	⊗	
choice	⊘	no mention, but assume no since junction not supported
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	⊗	

Modular Semantics for a UML Statechart Diagrams Kernel and its Extension to Multicharts and Branching Time Model-checking S. Gnesi and D. Latella and M. Massink	[30]
Towards a Formal Operational Semantics of UML Statechart Diagrams D. Latella and I. Majzik and M. Massink	[52]
Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker *D. Latella and I. Majzik and M. Massink*	[51]

This approach makes use of slightly modified Extended Hierarchical Automata (EHA) as an intermediate model, which is then transformed into Kripke structures for the semantic model of a subset of UML state machines. The approach is similar to one for classical statecharts [60] but takes into account the different priority scheme for UML state machines. The resulting semantics has only three rules (progress, composition and stuttering), which simplifies the correctness proofs of these rules [51]. [30] continues the work in [51], expanding on the kernel, extending it to include multiple state machines, which are modelled by a Labeled Transition System (LTS). [30] also introduces the JACK [13] environment, which can be used to edit the specifications (LTS expressed in the FC2 format) as well as model check them with the included AMC model checker.

States		
entry/exit actions	⊗	abstracted away
internal transitions	⊗	irrelevant when no entry/exit actions
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊗	
deferred events	⊗	
Pseudostates		
initial	●	
final	○	
fork/join	●	
history (shallow & deep)	⊗	
junction	⊗	no branch transitions
choice	⊗	no branch transitions
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	

State Diagrams in UML: A Formal Semantics using Graph Transformations M. Gogolla and F. Parisi Presicce	[31]
A Formal Semantics of UML State Machines Based on Structured Graph Transformation S. Kuske	[45]
An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformaiton *S. Kuske and M. Gogolla and R. Kollmann and H.-J. Kreowski*	[46]

[31] discusses how the inherent hierarchy in state machines can be normalized into flattened graphs with the use of graph transformation rules. The technique is quite intuitive, especially as the user does not need to know any underlying mathematical formalism. [45] continues by applying graph transformation techniques to define a formal semantics of the state machines. System configurations are represented as graphs and firing of transitions in the state machine correspond to the application of graph transformation rules [45].

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊗	according to authors, left out for space
deferred events	⊗	according to authors, left out for space
Pseuodstates		
initial	●	
final	●	
fork/join	○	
history (shallow & deep)	⊗	according to authors, should be able to extend
junction	○	
choice	○	
Transitions		
event trigger	●	
guard condition	⊗	according to authors, left out for space
action (behavior)	⊗	according to authors, left out for space
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	

Formalization of UML Statechart Models Using Concurrent Regular Expressions S. Jansamak and A. Surarerks	[39]

This approach provides transformation rules for formalizing state machines as Concurrent Regular Expressions (CREs) [29]. CREs are regular expressions with the addition of several operators: interleaving, alpha-closure, synchronous composition and renaming. Once a model's state machines are expressed as CREs, simple "inconsistency checking" [39] can be performed, i.e., state machines of different (interacting) objects can be checked to ensure there exists no inconsistency between them.

States		
entry/exit actions	⊘	
internal transitions	⊘	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊘	
deferred events	⊘	
Pseudostates		
initial	●	
final	○	
fork/join	●	
history (shallow & deep)	○	
junction	○	
choice	●	not enough details to confirm dynamic choice
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	⊗	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	

A Method for Describing the Syntax and Semantics of UML Statecharts [40]
Y. Jin and R. Esser and J.W. Janneck

Although an Abstract State Machine (ASM) approach, [40] deliberately separates the syntax and semantics of state machines. The syntax is represented by the Graph Type Definition Language (GTDL), which is a small domain-specific language and part of the Moses tool suite [27]. Well-formedness rules are represented as predicates over the abstract syntax of these graphs. The semantics is then represented as Object Mapping Automata (OMA) [38], which are a specialized version of ASMs. The Moses tools suite allows users to edit, simulate and automatically analyze such a system [40].

States		
entry/exit actions	●	
internal transitions	●	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	●	
Pseuodstates		
initial	●	
final	●	
fork/join	●	
history (shallow & deep)	●	
junction	●	
choice	⊙	no variables, so not really dynamic choice
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	●	

A UML Statecharts Semantics with Message-passing J. Jürjens	[41]

This approach makes use of Abstract State Machines (ASMs) to give a formal semantics of UML state machines; its major contribution is that state machines may communicate through message passing. The paper is quite short, but covers a good subset of UML state machine characteristics. It is expected that future work will continue to move towards executable UML modelling, i.e., allowing whole systems, rather than single diagrams, to be modelled [41].

States		
entry/exit actions	●	
internal transitions	●	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	⊗	according to author, should be able to extend
Pseuodstates		
initial	●	
final	●	
fork/join	⊘	transitions cannot cross borders
history (shallow & deep)	⊗	according to author, should be able to extend
junction	⊘	only one target or source
choice	⊘	only one target or source
Transitions		
event trigger	⊙	only one trigger
guard condition	●	
action (behavior)	⊙	only one action
priority scheme	○	
interlevel transitions	⊗	
Miscellaneous		
completion event/transition	●	

Using UML to Derive Stochastic Petri Net Models	[43]
P. King and R. Pooley	

The goal of [43] is to conduct performance analysis using stochastic Petri nets to represent a UML model (of which the statechart is just one part) of a particular example. They map the UML model to a variant of stochastic Petri nets, based on the Stochastic Petri Net Package (SPNP) tool [19]. The sample model has a very simple state machine and it is unknown whether or not this method can be scaled to cover more state machine characteristics. The performance analysis is specific to the example presented, e.g., throughput on a two-phase protocol.

Note: article too high level to determine much coverage information.

States		
entry/exit actions	⊘	
internal transitions	⊘	
sequential (OR)	⊘	
orthogonal (AND)	⊘	
do-activity	⊘	
deferred events	⊘	
Pseuodstates		
initial	●	
final	●	
fork/join	○	
history (shallow & deep)	○	
junction	○	
choice	○	
Transitions		
event trigger	●	
guard condition	⊘	
action (behavior)	●	
priority scheme	○	
interlevel transitions	○	
Miscellaneous		
completion event/transition	○	

Model Checking and Code Generation for UML State Machines and Collaborations A. Knapp and S. Merz	[44]
Model Checking UML State Machines and Collaborations T. Schäfer and A. Knapp and S. Merz	[74]

This approach showcases HUGO [74], a novel tool for state machines, which allows animation, model checking and the generation of Java code. Instead of model checking against properties expressed in temporal logic (which requires knowledge of the underlying model checker [44]), HUGO's model checking checks state machines against interaction diagrams. This results in a consistency check of state machines against specifications expressed as collaboration or sequence diagrams [44]. Other model checking is also possible, such as checking for the absence of deadlock, as well as more complicated properties expressed in temporal logic. HUGO makes use of the SPIN [36] model checker and its PROMELA input language, although it also has a back end for the real-time model checker UPPAAL [50]. In addition, state machines can be transformed into Java code, with a separate object being created for each state in the machine. One of the more interesting things about this approach is that it does not seem to rely on a mathematical formalism; states are modeled by individual PROMELA processes, with additional processes created to dispatch events and handle transitions [74]. This approach also manages to cover most of the UML state machine characteristics, with the exception of internal transitions, deferred events, and choice.

Feature	Support	Comment
States		
entry/exit actions	●	
internal transitions	○	according to authors, not required for model checker
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	○	
Pseudostates		
initial	●	
final	●	
fork/join	●	
history (shallow & deep)	●	
junction	⊙	drawn with diamond
choice	○	according to authors, implementation is obvious
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	●	

Rewrite Rules and Operational Semantics for Model Checking UML Statecharts G. Kwon	[47]

This approach makes use of a *conditional term rewriting system* and Kripke structures in order to translate state machines into the SMV [56] input language. Sets of active states are translated into terms and transition labels are translated into conditional rewrite rules [47]. The semantics for the conditional term rewriting are based on Kripke structures and these semantics are then translated into the SMV model checker's input language. This approach has some support for composite states but does not permit the "clean structure of hierarchy" to be violated by interlevel transitions [47].

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	○	
deferred events	○	
Pseuodstates		
initial	●	
final	○	
fork/join	○	
history (shallow & deep)	●	no mention of deep history
junction	○	
choice	○	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	

Structured Axiomatic Semantics for UML Models	[49]
K. Lano and J. Bicarregui and A. Evans	

This approach examines UML as a whole, representing UML models as "theories in extended first -order set theory" [49]. These theories, expressed in Real-time Action Logic (RAL) [48] can be used to represent classes, instances, associations, etc. State machines are formalized as extensions of the theories for classes. The approach is quite detailed, formalizing key parts of the UML, e.g., classes from the Core package. The most interesting contribution is the concept of *reductive transformations*, which are essentially algorithms which can be used to eliminate both nesting (OR-states) and concurrency (AND-states).

States		
entry/exit actions	●	reductive transformation to remove
internal transitions	○	
sequential (OR)	●	reductive transformation to remove
orthogonal (AND)	●	reductive transformation to remove
do-activity	○	
deferred events	⊗	
Pseudostates		
initial	○	
final	○	
fork/join	○	
history (shallow & deep)	⊗	
junction	○	
choice	○	
Transitions		
event trigger	○	
guard condition	⊙	no conditions dependent on attributes
action (behavior)	○	
priority scheme	○	
interlevel transitions	○	
Miscellaneous		
completion event/transition	○	

The Semantics of UML State Machines	[53]
J. Lilius and I. Porres Paltor	
vUML: a Tool for Verifying UML Models	[54]
J. Lilius and I. Porres Paltor	

This approach is a two-phased approach. The first phase consists of formalizing the structure of UML state machines, while the second focuses on the formalization of the operational semantics of state machines. The authors make use of a term rewriting system, where terms represent active state configurations with transitions mapping between these terms. For the formalization in the second phase the execution of a state machine is described in terms of a hypothetical machine with an event queue, event dispatch mechanism and event processor [53]. Each object in the UML model executes a run-to-completion algorithm. Not only does this approach cover most of the UML state machine characteristics, but it is also very nicely supported by the vUML tool [54]. vUML is a tool for model checking state machines; its appeal is that it can model check the same model that is being used to analyze, design, document and implement the target software system [53]. The tool is not used to model check a single state machine; rather, it can be used to model check the interactions between state machines, as defined in collaboration diagrams.

States		
entry/exit actions	●	
internal transitions	●	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	●	
deferred events	●	
Pseuodstates		
initial	●	
final	●	
fork/join	●	
history (shallow & deep)	●	
junction	⊙	don’t deal with directly but translate into simpler; drawn as diamond
choice	○	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	●	

Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach	[72]
G. Reggio and E. Astesiano and C. Choppy and H. Hussmann	
Metamodelling Behavioural Aspects: the Case of the UML State Machines (Complete Version)	[70]
G. Reggio	

[72] works towards a formalization of state machines of active classes by making use of Labeled Transition Systems (LTS) and conditional algebraic specifications written in the Common Algebraic Specification Language (CASL), which is part of the CoFI initiative [61]. The state machines considered are relatively primitive, with no AND-states and few pseudostates; however, the greatest contribution of this work could be considered the detailed examination of the UML 1.3 specification [62]. The paper raises several interesting questions, especially regarding the destruction of objects, how threads are used, etc. [70] continues the research into active classes by defining the LTS in an object-oriented way, i.e., with metamodelling. The approach, called 'Extreme Metamodelling' [69], makes use of GML as a "visual OO notation for presenting metamodels" [70]. Based on the previous work in [72, 71], metamodelling state machines is simply a matter of metamodelling the LTS used to define their semantics.

States		
entry/exit actions	⊗	translate to actions along incoming/outgoing transitions
internal transitions	⊗	irrelevant when no entry/exit actions
sequential (OR)	●	
orthogonal (AND)	⊗	
do-activity	⊗	
deferred events	●	
Pseudostates		
initial	⊙	only one initial state in entire machine
final	⊙	only one final state in entire machine
fork/join	⊗	
history (shallow & deep)	⊗	according to authors, should be able to extend
junction	⊙	restricted: only used to eliminate multiple transitions with the same trigger leaving a state
choice	⊗	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	⊘	not mentioned
interlevel transitions	●	
Miscellaneous		
completion event/transition	⊗	since assumption that all state machines eventually reach a solitary final state

On Semantics and Refinement of UML Statecharts: A Coalgebraic View [57]
S. Meng and Z. Naixiao and L.S. Barbosa

This approach discusses UML state machines represented as coalgebras. In fact, the authors make use of the claim that Labeled Transition Systems (LTS) can be naturally represented as coalgebras [73]. They do this by building on the work of [52] and presenting a coalgebraic representation of statecharts which supports that paper's operational semantics. The main contribution of this approach is the discussion of equivalence between statecharts and refinement of statecharts. They provide a mathematical look at equivalence and refinement, including several refinement laws.

Note: this approach builds directly on [52], so assume the same level of coverage.

States		
entry/exit actions	⊗	abstracted away
internal transitions	⊗	irrelevant when no entry/exit actions
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊗	
deferred events	⊗	
Pseuodstates		
initial	●	
final	○	
fork/join	●	
history (shallow & deep)	⊗	
junction	⊗	no branch transitions
choice	⊗	no branch transitions
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	

A Formal Semantics of UML Statecharts by Model Transition Systems [79]
D. Varró

This approach combines both metamodelling and graph transformation techniques to formally define the dynamic behaviour of UML state machines. The approach makes use of Extended Hierarchical Automata (EHA) as discussed in [51], but only as an “alternate structural representation” of state machines. The intermediate syntax is actually not required; the underlying formalism using dynamic attributes and relations manipulated by graph productions could be applied directly to the UML metamodel of statecharts. However, the addition of this intermediate step allows flexibility, e.g., the semantics could be applied to state machine variants, such as classical statecharts. The main contribution of this approach is that it keeps the syntactic and well-formedness concepts in the metamodel, while the dynamic semantics are specified by graph transformation rules [79]. In other work by the same author, the results were tested within the VIATRA tool [81]. In addition, the semantics were transformed [80] into SAL specifications [9] which could be used for various verification techniques, including model checking.

States		
entry/exit actions	⊘	
internal transitions	⊘	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊘	
deferred events	⊗	according to authors, omitted for space reasons
Pseudostates		
initial	●	
final	⊘	
fork/join	⊘	
history (shallow & deep)	⊗	according to authors, omitted for space reasons
junction	⊘	
choice	⊘	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	

A Structured Operational Semantics for UML-statecharts	[83]
M. von der Beeck	

[83] makes use of a term syntax as an intermediate formalism, which the author claims is "more similar" to the original UML state machine syntax than the Extended Hierarchical Automata (EHA) used by some other approaches. The semantics is defined in two phases: first, an auxiliary semantics is defined which deals only with the processing of single events [83]. This first phase is inspired by Plotkin's Structured Operational Semantics (SOS) [67] approach and essentially maps the syntax terms to sets of Labeled Transition Systems (LTS). The second phase uses the auxiliary semantics to create a semantics dealing with sequences of events; this phase makes use of Kripke structures, which are appropriate for modeling the concept that the output of one step serves as part of the input of the next step [83].

States		
entry/exit actions	●	
internal transitions	○	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	⊗	
deferred events	⊗	
Pseudostates		
initial	⊗	
final	⊗	
fork/join	⊗	no branching
history (shallow & deep)	●	
junction	⊗	
choice	⊗	
Transitions		
event trigger	⊙	no time or change events
guard condition	⊗	
action (behavior)	●	
priority scheme	●	
interlevel transitions	●	
Miscellaneous		
completion event/transition	⊗	

An Approach to Formalizing the Semantics of UML Statecharts	[84]
X. Zhan and H. Miao	
Formalizing the Semantics of UML Statecharts with Z	[85]
X. Zhan and H. Miao and L. Liu	

This approach formalizes UML state machines in Z [76]. [85] introduces Z-schemas for state machines, including requirements with respect to well-formedness, compound transitions, priority among conflicting transitions, etc. [84] extends this work, adding discussions about how to generate test cases (for a class) from a UML state machine that makes use of hierarchy (OR-states) and concurrency (AND-states). The authors demonstrate how to translate state machines into FREE [11] models, which essentially move actions out of states. According to [11], models which follow the FREE conventions are testable.

States		
entry/exit actions	○	
internal transitions	○	
sequential (OR)	●	
orthogonal (AND)	●	
do-activity	○	
deferred events	○	
Pseuodstates		
initial	●	
final	●	
fork/join	○	
history (shallow & deep)	○	
junction	○	
choice	○	
Transitions		
event trigger	●	
guard condition	●	
action (behavior)	⊙	only generation of events
priority scheme	○	
interlevel transitions	●	
Miscellaneous		
completion event/transition	○	