On the Syntax and Semantics of State Machines

Michelle L. Crane
February 6, 2006

A depth paper submitted to the

School of Computing
Queen's University
Kingston, Ontario, Canada

in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Copyright ©2005 Michelle Love Crane

Abstract

Within Model Driven Development (MDD), state machines are a common mechanism for modelling behaviour. The development of a formal semantics for 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. Research on state machine semantics is severely complicated by the fact that there are multiple suggested approaches to formalizing state machine semantics. In addition, there are currently several well-documented state machine dialects, each subtly different from the others.

According to the research literature, the most popular state machine dialects are classical and Rhapsody statecharts and UML state machine diagrams. These three dialects appear to be very similar; however, there are several key syntactic and semantic differences. The first half of this paper presents the results of a comparative study of these three dialects with the help of several illustrative examples. We also present a classification of the differences, together with a comprehensive overview.

The second half of this paper is the result of a comparative literature review 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.

Acknowledgements

```
10   This is the depth that never ends.
20   It goes on and on my friends.
30   Someone started writing it not knowing what it was,
40   And they'll continue writing it forever just because
50   GOTO 10
```

On a more serious note, I would like to thank Val for her continued support and encouragement.

I would also like to thank Juergen for being Juergen.

On a much more serious note, I would like to acknowledge the invaluable assistance of Bran Selic from IBM Rational Software Canada with respect to the UML 2.0 specification.

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.

Contents

Abstract i

Acknowledgements ii

Table of Contents iii

List of Tables v

List of Figures v

1 Introduction 1
- 1.1 Organization of Paper . . . 3

2 State Machines 3
- 2.1 Finite State Machines . . . 3
- 2.2 State Machine Dialects . . . 4
  - 2.2.1 Classical Statecharts . . . 5
  - 2.2.2 UML State Machines . . . 5
  - 2.2.3 Rhapsody Statecharts . . . 6
- 2.3 Related Work . . . 6

3 Detailed Comparison of State Machine Dialects 7
- 3.1 Synchrony Hypothesis . . . 8
  - 3.1.1 Synchrony and Zero Time . . . 8
  - 3.1.2 Synchrony and Simultaneous Events . . . 8
- 3.2 Priorities of Conflicting Transitions . . . 9
- 3.3 Order of Execution of Actions . . . 10
- 3.4 Fork and Join . . . 10
- 3.5 Junction . . . 12
- 3.6 Conditional . . . 12
- 3.7 Choice . . . 13
- 3.8 Summary Table . . . 14
- 3.9 Implications of Differences . . . 15

4 Semantic Approaches to UML State Machines 18
- 4.1 Related Work . . . 18

5 Categorization of Semantic Approaches 19
- 5.1 Mathematical Models . . . 19
- 5.2 Rewriting Systems . . . 20
- 5.3 Translation Approaches . . . 21
- 5.4 Overlap in Reference Set . . . 22

6 Comparison of Semantic Approaches **22**
6.1 UML Coverage 24
6.2 Analysis 30
6.3 Tool Support 31
6.4 Comparison Summary 32

7 Conclusion **33**
7.1 Future Work 33

References **35**

List of Tables

1 Summary of Differences 17
2 Categorization of semantic approach references 23
3 UML feature coverage legend for Tables 4, 5 and 6 25
4 Coverage of mathematical model approaches 25
5 Coverage of rewriting approaches 26
6 Coverage of translation approaches 26
7 Sub-category coverage of UML features 29
8 Tool support for analyis 32

List of Figures

1 Example finite state machine 3
2 State machine that is well-formed but interpreted differently 4
3 Example classical Harel statechart 5
4 State machine with potentially simultaneous events 9
5 State machine with conflicting transitions 9
6 Transition with a list of actions [43] 10
7 This UML fork would be ill-formed in Rhapsody 11
8 This classical fork would be ill-formed in both UML and Rhapsody 11
9 This classical join would be ill-formed in UML 11
10 This UML join would be ill-formed in Rhapsody 11
11 This classical junction can be made compatible to UML 12
12 This classical junction would be ill-formed in UML and Rhapsody 12
13 Conditional construct supported by classical and Rhapsody dialects 13
14 UML supports the same static choice by using the junction pseudostate 13
15 UML supports dynamic choice 13
16 Dynamic choice can be simulated in Rhapsody 13
17 Primary categorization of semantic approaches for UML state machines 19
18 Overlap between primary sub-categories 22
19 UML state machine features, ordered by coverage lavel 28
20 Histogram showing which versions of UML are supported 30

1 Introduction

Model Driven Development (MDD) is a software development process that has been gaining in popularity. MDD focuses on the models, or abstractions of the software system, rather than on the final programs [106]; these models are transformed into code. Executable models are a key component of MDD, as well as such concepts as automatic transformation of models, validation of models, and standardization to enable interoperability of different MDD tools (e.g., OMG's Model Driven Architecture (MDA) [85] initiative). Within MDD, state machines are a common mechanism for modelling the behaviour of model elements. Although descended from a fairly well-understood model of computation, i.e., finite state machines, today's state machine dialects suffer from the lack of a clear, concise and comprehensive semantics. There are literally dozens of approaches suggested in the research literature for formalizing the semantics of current state machine dialects. However, no one formal semantics approach has the properties mentioned above and widespread acceptance in the field.

Research on state machine semantics is severely complicated by the following:

1. There are multiple suggested approaches employing a diverse set of formalisms, including: transition systems, abstract state machines, Petri nets, graph rewriting, term rewriting, and translation to model checking languages, specification languages or even axiomatic systems. In addition, these approaches vary widely in terms of which state machine features they cover, the analyses they support, and how well they are supported by tools.

2. There are currently several state machine dialects, e.g., UML state machines, classical statecharts and RHAPSODY statecharts. Each of these dialects is subtly different from the others. For instance, classical statecharts is the only dialect that allows for the handling of simultaneous events, UML is the only dialect that permits dynamic choice, etc. These differences are such that a semantic approach tailored to one dialect cannot be automatically applied to another dialect.

Defining the semantics of a language involves defining/understanding the syntax of a language, choosing a semantic domain that is understood, and performing a mapping from the syntax to the semantic domain [46]. The main purpose of this paper is to survey approaches to formalizing state machines, i.e., to examine different ways of mapping from the language of state machines, specifically state machines in the Unified Modeling Language (UML), to various mathematical and non-mathematical formalisms.

In the interests of clarity, we define and use the following terminology in this paper:

state machine In the purest sense, the term 'state machine' refers to the actual model of computation, e.g., finite state machine. We refer to both the model of computation and fragments presented in figures (regardless of the dialect), e.g., "The state machine in Fig. x".

dialect We discuss three separate realizations/interpretations of state machines. Different state machine dialects differ in how certain syntactic constructs are represented. They may also differ in how certain semantic concepts are handled.

statecharts Harel invented a specific version of state machines called 'statecharts'; this term refers to Harel's interpretation of the model of computation (used in both the classical and RHAPSODY dialects). In addition, the term 'statecharts' also refers to the representations (or diagrams) for these two dialects.

state machine diagram In general, this term refers to the representation of a state machine. With respect to the three dialects discussed in this paper, UML is the only one that uses this term; the classical and RHAPSODY dialects use the term 'statecharts' to refer to diagrams.

notation This term is reserved for the symbols used in state machines or in a particular dialect to graphically represent the various parts of a machine, e.g., initial state, junction, etc.

formalism We use this term to refer to the underlying semantic methodology (semantic domain) to which state machines are being mapped, e.g., transition system, model checking language, etc.

approach This term refers to a particular mapping of (elements of) a specific state machine dialect to a specific formalism. An approach may refer to several works by the same group of authors.

We focus on the UML dialect for the following reasons:

- UML has become the *de facto* industry standard for modelling software.
- There exists a considerable body of work published on UML, especially with respect to its various diagram types, and semantics of those diagram types.
- Unlike other current state machine dialects (e.g., STATEMATE and RHAPSODY), UML state machines are not tied to any specific development process or tool.

To understand the syntax of this dialect, we compare UML state machines with classical and RHAPSODY statecharts. The first part of this paper provides an in-depth comparison of these three dialects.

The second part of this paper provides a high-level categorization and detailed comparison of 26 different semantic approaches. Each of these approaches provides a mapping from the syntax of UML state machines to some understood semantic domain.

1.1 Organization of Paper

This paper is organized as follows: Section 2 briefly describes state machines and three common dialects; in addition, this section motivates the requirement to study the differences between the three dialects. Section 3 contains a detailed comparison of syntactic constructs and semantic concepts, which differ between the three dialects; this section includes a tabular summary. Section 4 motivates the study of different semantic approaches to UML state machines and discusses why the semantics of classical statecharts cannot necessarily be applied to UML state machines. Section 5 divides the surveyed approaches into three primary categories. Section 6 then compares the semantic approaches along several secondary dimensions. Section 7 presents conclusions and future work.

2 State Machines

2.1 Finite State Machines

A finite state machine (FSM) is a model of computation that "specifies the sequence of states an object goes through during its lifetime in response to events" [13, Chapter 2]. An FSM is essentially an abstract machine that consists of a set of states, an initial state, an input alphabet of events and a transition function mapping current states and event symbols to next states [49]. The term 'finite state machine' refers to the model of computation but not the diagram representing it; instead, the traditional name for a diagram representing a FSM is 'state diagram' or 'state transition diagram'.

FSMs are very useful for representing reactive systems, more so than linear or textual representations, which are more suitable to transformational systems [50]. Ostensibly, an FSM produces output only when it reaches a final state; however, there are many variations, such as Mealy and Moore machines. Mealy machines can produce output along any transition [50], while Moore machines can produce output at any state [39]. The FSMs that we refer to in this context are Mealy-Moore machines, a combination of these two variants.

Figure 1 shows part of a simple finite state machine with three states and five transitions. If the state machine were in state A, an event b would cause it to move to state B, while an event c would cause it to move to state C. Similarly, if the machine were in state B, event a would cause it to move to state A and event d would cause it to move to state C, etc.

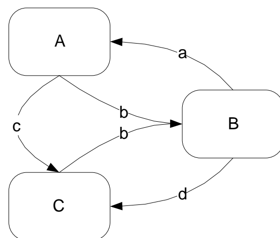
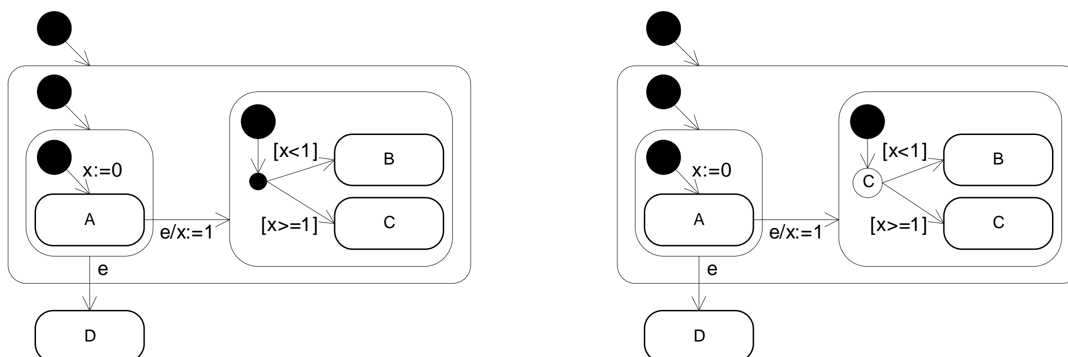


Figure 1: Example finite state machine from [39]

2.2 State Machine Dialects

Three popular state machine dialects, as represented in the research literature, are the Unified Modeling Language state machine diagrams (as specified in UML 2.0 [91]), classical Harel statecharts (implemented in STATEMATE [43, 45]), and a newer object-oriented version of Harel's statecharts (implemented in RHAPSODY [42]). These three dialects appear to be very similar. For instance, at first glance, a model written in one dialect could be easily ported to one of the other two dialects. However, there are some subtle syntactic and semantic differences between the dialects, which can lead to pitfalls.Consider, for example, the state machines shown in Figure 2. The two machines are identical, except for the symbol used to represent static choice. Figure 2(a) makes use of a junction (small filled circle); this machine is well-formed in the classical and UML dialects. Figure 2(b) shows a conditional construct (circled 'C'), which is used by both the classical and RHAPSODY dialects. Ignoring the different notation for a minute, this model is well-formed in all three dialects. However, the behaviour exhibited by the state machine is different for all three dialects. When the state machine first starts, it moves to state A, at which point the variable $x = 0$. All three dialects agree on this point. What they do not agree on is what happens when event e occurs. In the classical dialect, the state machine moves to state D. In UML, the state machine moves to state B. Finally, in RHAPSODY, the state machine moves to state C.



(a) Junction (small filled circle) used for static choice. Model is well-formed in the classical and UML dialects

(b) Conditional (circled 'C') used for static choice. Model is well-formed in the classical and RHAPSODY dialects

Figure 2: Ignoring dialect differences, this model is well-formed in all three dialects, but is interpreted differently in all three. The classical state machine moves to D because the priority of conflicting transitions is handled differently (see Section 3.2). In UML, the junction is a static choice, i.e., the guards are evaluated with the information available at the beginning of the entire transition. Here, $x = 0$, so the state machine moves to B. In RHAPSODY, the conditional is also a static choice, but the fact that it is enclosed in a composite state causes it to behave as a dynamic choice (see Section 3.7). The initial transition is a 'microstep'; variables are evaluated at the beginning of each microstep. $x = 1$ when the conditional is reached and the state machine moves to C

The fact that there can be three distinct interpretations of one state machine indicates that there is a lack of standardization between the three dialects. It also indicates that the

task of transforming, or porting, a model from one dialect to another may not be straightforward. Therefore, it is worthwhile to study the syntactic and semantic differences between these three popular dialects.

2.2.1 Classical Statecharts

In the late 1980's Harel defined a "visual formalism for describing states and transitions in a modular fashion, enabling clustering, orthogonality, and refinement, and encouraging 'zoom' capabilities...between levels of abstraction" [39]. These new *statecharts* were essentially state transition diagrams with the addition of hierarchy (also known as depth), orthogonality (also known as concurrency) and broadcast communications [39, 75]. Other publications by Harel and other authors quickly followed, defining a preliminary semantics for the statecharts dialect [44, 96]. Far from being a final product, the statecharts dialect evolved over the years, spawning many variants. In fact, as of 1994, there were at least 20 variants of these statecharts [116]. In 1996, Harel revisited his statecharts, modifying some of the previous semantics [40, 43]. These statecharts are often referred to in the research literature as simply 'statecharts', 'Harel statecharts', or 'classical statecharts'. Because of the fact that the semantics of statecharts has evolved over the years, and the fact that there are so many variants, it is necessary to define unambiguously to which statecharts we refer. For the purposes of this paper, the term 'classical statecharts' will be used to represent Harel's original statecharts syntax with the newest semantics, as documented in [40, 43, 45]. Although Harel himself states that there is no official semantics for his statecharts [43], classical statecharts are actually implemented in I-Logix's STATEMATE tool [54], to which Harel has contributed.

Figure 3 shows a classical statechart fragment from [39]. It demonstrates how the use of hierarchy (implemented with an OR state) can simplify a finite state machine; this statechart is in fact behaviourally equivalent to the finite state machine shown in Figure 1.

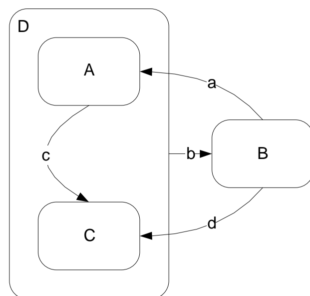


Figure 3: Example classical Harel statechart showing use of hierarchy, from [39]. This statechart is behaviourally equivalent to the finite state machine shown in Fig. 1

2.2.2 UML State Machines

The Unified Modeling Language has become the *de facto* industry standard for general-purpose modelling; it can be used for "specifying, constructing and documenting the artifacts of a system" [88, Part I]. The UML is a visual modelling language; different diagram

types (sub-languages) can be used to model various parts of the system under consideration. These diagram types can be sub-divided into structural and behavioural views. In addition, behavioural diagrams can be further sub-divided into inter-object and intra-object behavioural views. UML state machine diagrams[1] are one diagram type that can be used to model intra-object behaviour, i.e., how individual model elements behave. The syntax and semantics of UML state machines have remained reasonably consistent throughout UML's history, although there are occasionally minor modifications. We concern ourselves with the version of UML state machines given in the latest draft of the UML 2.0 Superstructure specification [91].

2.2.3 Rhapsody Statecharts

UML state machine diagrams are an object-based variant of classical statecharts [91, 87, 36]. An alternative object-based variant is one to which Harel himself has contributed: the statechart dialect implemented in I-Logix's RHAPSODY tool [53]. This dialect was created after the introduction of UML 1.1. Actually, the RHAPSODY dialect is more closely related to UML than to its classical ancestor. In fact, there was cooperation between the RHAPSODY and UML development teams, resulting in cross-pollination between the two dialects [41, 107]. For the purposes of this paper, we concern ourselves with RHAPSODY as it is documented in [41, 42].

2.3 Related Work

The UML 2.0 Semantics Project[2] is an international collaboration including IBM (Canada, Germany, Israel), Queen's University (Canada), the Technical University of Munich (Germany), and the Technical University of Braunschweig (Germany). The purpose of this project is to define a formal semantics of UML 2.0. Under the auspices of this project, we have initiated an effort to survey, categorize and compare semantic approaches for formalizing state machine behaviour. In order to critique these approaches, we needed a detailed understanding of the syntax and intended semantics of state machines. During our literature review, it became apparent that classical, UML and RHAPSODY state machines could not automatically be considered equivalent, even though at first glance, they appear almost identical.

Unfortunately, although there is much research relating to these dialects, there is no definitive comparison between them. The most detailed comparison is a bulleted list in an older UML specification [87, Section 2.12.5.4], which is not even included in the new UML 2.0 specification. Other sources offer one- or two-line high-level comparisons between classical statecharts and UML state machines, without going into great detail. The bulk of the research presented in the first half of this paper is thus a result of detailed inspection of

[1] The newest versions of the UML 2.0 specification [91], [91] use the term 'state machine diagram'; previous versions of the standard [89], [87] use the term 'statechart diagram'.

[2] `http://www.cs.queensu.ca/~stl/internal/uml2/`

the UML specification [91], as well as key documents relating to the classical [39, 40, 43, 45] and RHAPSODY [41, 42] dialects.

3 Detailed Comparison of State Machine Dialects

In general, the three state machine dialects (classical, UML and RHAPSODY) are similar. Basically, statecharts, or state machine diagrams, are directed graphs, consisting of states and transitions between them. Transitions may have labels of the form event[guard]/action. All three dialects support both orthogonal (AND) and sequential (OR) composite states.

These basic similarities aside, there are several syntactic and semantic differences between the three dialects. The syntactic differences concern how various syntactic constructs are represented and their well-formedness constraints, while the semantic differences are caused by variations in basic semantic concepts. These differences can be divided into three categories, based on the type and severity of errors that they can cause when porting state machines from one dialect to another. Note that a particular syntactic construct or semantic concept can result in differences in more than one category.

Notation A construct may be common to all three dialects and yet be represented by alternative notation. For example, a final state in UML is represented as "a circle surrounding a smaller solid filled circle" [91], while the classical and RHAPSODY dialects make use of a circled 'T'. This category is the least critical; after a simple notation translation, a state machine would be compatible with the target dialect(s).

Well-Formedness Differences in this category are more important; they result in models that are well-formed in one or two dialects, but not in all three. For instance, a construct may not be available in a particular dialect, or a dialect may enforce additional or different constraints on a common construct. A state machine could be checked for compatibility with simple syntax or well-formedness checking. Translation and re-working of a state machine may make it compatible with the target dialect(s); however, not all state machines can be made fully compatible with all dialects. For example, event triggers are not permitted after pseudostates in UML; however, it may be possible to re-work the state machine to conform to this restriction. On the other hand, simultaneous events cannot be handled simultaneously in UML; it may not be possible to re-work a classical state machine to mimic this behaviour without using simultaneous events.

Executable Behaviour This is the most critical category of differences, and the most insidious. A state machine may be well-formed in more than one dialect and yet not behave exactly the same. This type of incompatibility would not be found by simple syntax or well-formedness checking. In essence, an incompatible state machine would 'compile', but its executable behaviour would be other than expected, sometimes the opposite of the intended behaviour.

In order to more fully understand these categories and the potential problems associated with each, we now examine several syntactic constructs and semantic concepts in detail. We start with the semantic concepts because, in general, they affect multiple constructs and the overall understanding of the models. Then, the more interesting syntactic constructs are examined.

3.1 Synchrony Hypothesis

3.1.1 Synchrony and Zero Time

The *(perfect) synchrony hypothesis* [11] states that a system must react immediately to external events and that the corresponding output must occur at the same time [116]. The zero-time assumption follows from the synchrony hypothesis and implies that transitions take zero time to execute [87]. In general, classical statecharts support both the synchrony hypothesis and the zero-time assumption [116, 87].[3]

In UML, a transition *may* take time [87], although no assumptions are actually made, allowing for models with either zero- or fixed-execution time [91, Section 13.3.30].[4] The Rhapsody dialect mirrors that of UML in that a "step does not necessarily take zero time" [42]. Therefore, with respect to the zero-time assumption, it is theoretically possible that both the UML and Rhapsody dialects adhere to the synchrony hypothesis.

3.1.2 Synchrony and Simultaneous Events

By the synchrony hypothesis, classical statecharts must be able to react immediately to external events. This is possible, based on the fact that different events may occur simultaneously, and be acted upon simultaneously, in classical statecharts [76]. However, neither the UML nor Rhapsody dialects support the synchrony hypothesis in this regard. Instead, both dialects adhere to the concept of run-to-completion (RTC), which means that each event is handled completely before the next event is processed.[5]

It is thus impossible in a UML or Rhapsody state machine for different events to be handled simultaneously.[6] For example, consider the state machine in Figure 4. Assume that the state machine is currently in states A and C and that events e1 and e2 occur simultaneously. If this were a classical statechart, then both events would be handled simultaneously (since they do not conflict) and the machine would move to states B and D in one step.

[3]Note that classical statecharts semantics, as implemented in Statemate, supports two time models: asynchronous and synchronous. Only the asynchronous time model supports zero-time transitions [43].

[4]This is what is known as a 'semantic variation point' in the UML specification. These "explicitly identify the areas where the semantics are intentionally under-specified to provide leeway for domain-specific refinements of the general UML semantics" [91, Section 6.5.1]. In other words, users may make their own choices at these semantic variation points; as long as their choices are documented, the model is still considered to conform to the UML specification.

[5]In UML, "event occurrences are detected, dispatched, and then processed...one at a time" [91, Section 15.3.12]. In Rhapsody, events are handled "one by one, in order" [41].

[6]It is however, possible for the same event to be handled simultaneously in different regions of an orthogonal composite state.

However, in the other two dialects, only one event can be handled at a time. Therefore, the state machine would next move to either states A and D or B and C, depending on which event was handled.

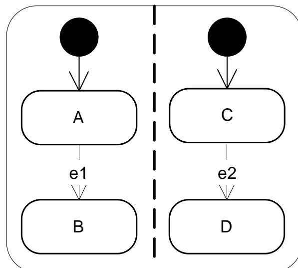


Figure 4: State machine with potentially simultaneous events

3.2 Priorities of Conflicting Transitions

It is possible in all three dialects to have conflicting transitions, i.e., a set of enabled transitions that cannot all be fired due to conflict in their results. For example, consider the state machine in Figure 5. Assume that the machine is currently in state B and that event e is generated. The two transitions enabled by this event are considered to be in conflict because their effects conflict. For instance, if the transition into state D is taken, the state machine moves to state D, and the transition into state C cannot be taken.

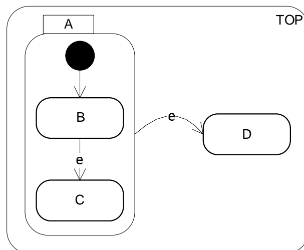


Figure 5: State machine with conflicting transitions

One of the most serious differences between the UML/Rhapsody and classical dialects is the handling of conflicting transitions. In classical statecharts, the *scope* of a transition is the lowest OR-state neither exited nor entered by that transition [43, 76]. Priority is given to the transition with the highest scope. In the case of the state machine in Figure 5, the scope of the transition from B to C is state A, while the scope of the transition from A is D is the state TOP. Since priority is given to the transition with the highest scope, the latter event is handled; therefore, the state machine moves to state D.

In UML, a "transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states" [91, Section 15.3.12]. In Rhapsody, lower level states also get priority [41]. In this case, the transition from B to C originates

from state B, which is a substate of state A, the origin of the transition from A to D. Since priority is given to the substates, the former event is handled; therefore, the state machine moves to state C in both UML and RHAPSODY.

The rationale behind the different priority schemes is not well-documented, although it has been suggested that the lowest-first priority scheme espoused by both UML and RHAPSODY is more object-oriented. In other words, this priority scheme allows substates to override superstates in a way that is similar to how subclass operations/methods can override those of the superclass [42].

3.3 Order of Execution of Actions

In all three dialects, is it possible to list multiple actions (or behaviours) on a transition between two states, as shown in Figure 6. Assume that the state machine is in state A, $x = 0$, and event e occurs. In classical statecharts, actions on a transition are executed in parallel, rather than in sequence [43]. Therefore, at state B, $x = 1$ and $y = 0$, because both actions were executed simultaneously. In UML however, the behaviour expression "may be an action sequence comprising a number of distinct actions" and "behaviors are executed in sequence following their linear order" [91, Section 15.3.14]. Similarly, in RHAPSODY, "actions are guaranteed to be performed in sequential order" [42]. For both UML and RHAPSODY therefore, at state B, $x = 1$ and $y = 5$.

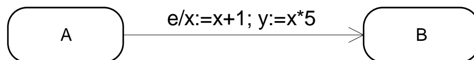


Figure 6: Transition with a list of actions [43]

3.4 Fork and Join

Fork and join constructs are common to all three dialects, although the symbols are slightly different in classical/RHAPSODY than in UML. Published work on the classical and RHAPSODY dialects show forks and joins as simply arrows with either multiple sources or multiple targets. The UML specification shows separate fork and join constructs, which break the transitions into incoming and outgoing transitions.

In addition to the dialect differences between the dialects, there are several well-formedness differences. For example, actions (or any labelling) are not permitted on the outgoing transitions of a fork in RHAPSODY. Thus, the UML state machine in Figure 7 would be ill-formed in RHAPSODY, even with the alternate notation taken into account.

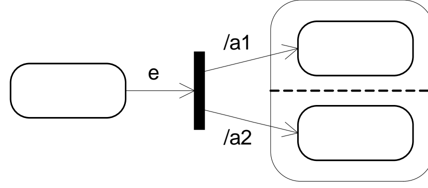


Figure 7: This UML fork would be ill-formed in Rhapsody

As another example, the classical statechart in Figure 8 would be ill-formed in both UML and Rhapsody. In the first place, Rhapsody does not allow the labelling of transitions leaving a fork. UML does allow the placement of actions on these transitions, but not event triggers. However, there is a much more fundamental semantic difference between the classical and the other two dialects. In the classical dialect, the fork transition would only be taken if all three events `e`, `e1` and `e2` were to occur simultaneously, which is possible since the classical dialect allows for simultaneous events. On the other hand, both UML and Rhapsody adhere to the RTC assumption; therefore, only one event can be handled at a time.

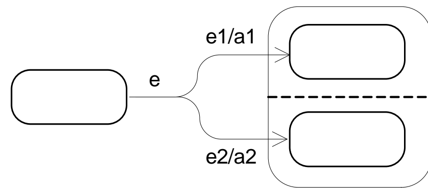


Figure 8: This classical fork would be ill-formed in both UML and Rhapsody

The classical statechart in Figure 9 would be ill-formed in UML because UML does not allow for event triggers after the join pseudostate. In addition, the obvious solution of simply moving the event trigger to the incoming transitions would not work; UML does not allow for event triggers incoming to join pseudostates. In fact, joins are not explicitly triggered in UML; they are only used with completion events [107], i.e., events that are automatically triggered when the last state(s) in orthogonal regions are completed. Finally, the UML state machine in Figure 10 would be ill-formed in Rhapsody, since Rhapsody does not allow for any labels on transitions coming into a join.

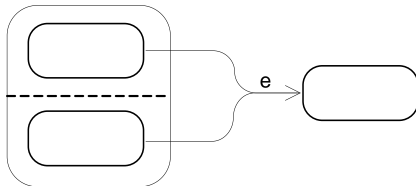


Figure 9: This classical join would be ill-formed in UML

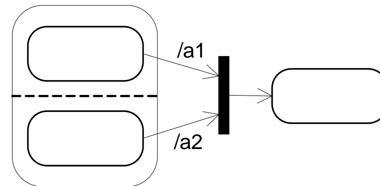


Figure 10: This UML join would be ill-formed in Rhapsody

3.5 Junction

Junction constructs are common to all three dialects, although there are some well-formedness differences. For example, the classical statechart in Figure 11 is ill-formed in UML. However, it is possible to make the state machine compatible by simply moving the event trigger to the transitions coming into the junction because UML allows for event triggers on transitions coming into junctions. In fact, each incoming transition may even have a different event trigger.

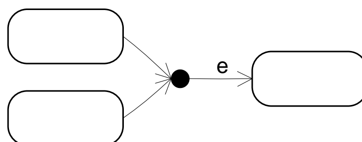


Figure 11: This classical junction can be made compatible to UML

In addition, the RTC assumption also affects the compatibility of the junction construct. For example, the classical statechart in Figure 12 is ill-formed in both UML and RHAPSODY. The transition in question will only be triggered if both events `e1` and `e2` occur at the same time, which is possible with classical statecharts but not with UML or RHAPSODY. In addition, UML does not allow for event triggers on transitions outgoing from a pseudostate. Finally, RHAPSODY does not allow for more than one outgoing transition from a junction.

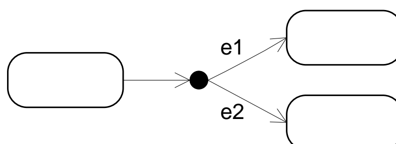


Figure 12: This classical junction would be ill-formed in UML and RHAPSODY

3.6 Conditional

Classical and RHAPSODY statecharts support a specific conditional construct, shown in Figure 13. This construct simply represents a *static* choice, i.e., the guards on the outgoing transitions are evaluated before the transition is taken. The conditional construct no longer exists in UML,[7] but its semantics are identical to those of the standard junction pseudostate, as shown in Figure 14.

[7]The conditional construct was removed from UML 1.3, since it is equivalent to a junction [29, Section 3.4.3].

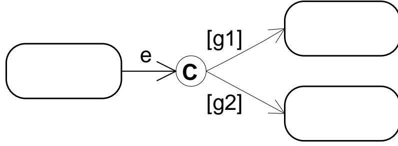


Figure 13: Conditional construct supported by classical and RHAPSODY dialects

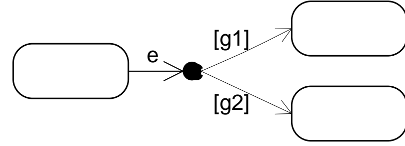


Figure 14: UML supports the same static choice by using the junction pseudostate

3.7 Choice

UML does allow for a *dynamic* choice pseudostate, which is not equivalent to the classical/RHAPSODY conditional construct. Consider the UML state machine in Figure 15. When the state machine starts, it moves to state A and $x = 0$. When event e occurs, the action on the transition is executed *before* the guards on the outgoing transitions are evaluated. The state machine will thus move to state C.

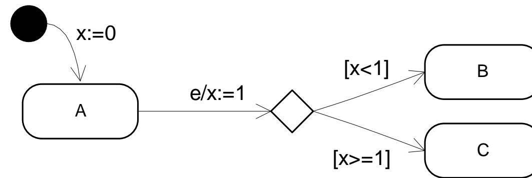


Figure 15: UML supports dynamic choice

Although neither the classical nor RHAPSODY dialects support this dynamic choice construct, it is possible to simulate it at least in RHAPSODY. Consider the RHAPSODY statechart in Figure 16. In this case, the fact that RHAPSODY makes use of *microsteps* [42] comes into play. The default, or initial, transition is considered a microstep. Attributes are assigned their values at the beginning of each microstep, so the assignment $x := 1$ is executed as the state machine enters the composite state. Once the conditional is reached, $x = 1$, so the state machine would move to state C.

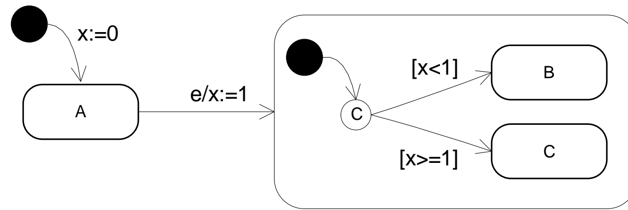


Figure 16: Dynamic choice can be simulated in RHAPSODY. State machine from [42]

It is very important to note that even if the conditional in Figure 16 were replaced by UML's static choice construct (junction), the state machine would not behave identically in UML as it does in RHAPSODY. UML does not make use of microsteps, and the action along

the transition would not be considered when the guards are evaluated [107]. If this state machine were to be evaluated in UML, it would move to state B.

3.8 Summary Table

Table 1 summarizes the findings of this section, as well as results for some other syntactic constructs. The left-hand columns of the table summarize the syntactic and semantic differences. UML 2.0 is used as the baseline, with classical and RHAPSODY both being compared to it. The right-hand columns indicate in which potential problem categories each construct and concept fall:

- The notation category indicates differences which can be easily managed, i.e., a state machine in one dialect can be easily ported to the other dialects with a simple notation translation.
- Differences in the well-formedness category are more serious. Sometimes it will be possible to modify a state machine to make it compatible to another dialect, e.g., the UML state machine in Figure 14 represents the classical/RHAPSODY statechart in Figure 13. Unfortunately, not all state machines can be made compatible, e.g., the classical statechart in Figure 9 cannot be translated into an equivalent UML state machine.
- Finally, differences in execution behaviour are the most serious of all. This is not because they imply that a state machine cannot be ported to another dialect, but because a state machine designed with constructs/concepts from this category can be well-formed in more than one dialect and yet behave differently in each. The state machines in Figure 2 are prime examples of this particular pitfall.

Obviously, problems caused by well-formedness differences can also cause problems in execution behaviour. For example, a UML state machine with deferred events[8] would be ill-formed in the other two dialects. However, if the deferred events were simply removed, the state machine would not behave as expected. In this case, the execution behaviour problem would not be indicated in Table 1, since the well-formedness problem itself alerts modellers of the mismatch and thus encourages them to ensure that a ported state machine is well-formed and behaves as expected. Instead, the behavioural problems indicated in Table 1 are *in addition* to any notation or well-formedness problems for that construct/concept, and not caused by them.

Not only does this table present a comprehensive summary of the differences between the three dialects, but it also brings to light several facts, such as:

[8]Normally, when an event occurs, it either matches the event trigger on some transition and is handled, or it does not match any trigger and is ignored. However, the use of deferred events allows a state to recognize certain events (which do not trigger transitions in that state) and postpone responding to them [91, Section 15.3.11].

- Rhapsody is much closer, syntactically and semantically, to UML than to its classical ancestor, especially with respect to behavioural semantics. This means that state machines can be more easily ported between UML and Rhapsody than between either of these dialects and classical statecharts.

- UML is the only dialect that allows for dynamic choice.

- Many of the well-formedness and execution behaviour differences between classical and UML/Rhapsody are indirectly caused by the fact that UML and Rhapsody do not support simultaneous events or actions, e.g., with respect to do-activities, forks, joins, junctions, and event triggers.

- Although the priority scheme between the classical and UML/Rhapsody dialects is inverted, it does not cause any notation or well-formedness problems. In other words, the fact that a state machine would behave differently due to the opposite priority schemes would not be found by a syntax or well-formedness checker.

3.9 Implications of Differences

There are currently at least three popular dialects for modelling state machines: UML state machine diagrams, classical statecharts and Rhapsody statecharts. Modellers may adhere to MDD without being restricted to one particular dialect. In general, the similarities between classical statecharts, UML state machine diagrams, and the statecharts implemented by the Rhapsody tool are enough to suggest to the non-expert that a state machine modelled in one dialect can be interpreted in the other dialects. Unfortunately, this is not necessarily the case; there are enough syntactic and semantic differences between the dialects to cause problems when sharing models.

Some problems are caused by simple notation differences and can be solved with a translation. Some problems cause well-formedness issues; occasionally, these problems can be solved with translation or re-working of the model. Occasionally, these problems cannot be solved, but at least their presence can be identified by syntax or well-formedness checks. Finally, some problems cannot be identified by such checks; these are the most insidious problems and result in well-formed state machines which behave differently in different dialects.

The comparative study described in this section is of interest to modellers, tool developers, and end users of statecharts and state machine diagrams for the following reasons:

- Modellers should be aware of how their state machines will be interpreted in different dialects. This is especially important with respect to execution behaviour issues, where a modeller might be expecting a different behaviour than that exhibited by a state machine. In the same vein, state machines can be used as a communication medium between modellers and their customers, or end users. Users may interpret these state machines differently, based on an alternate dialect with which they are familiar. Indeed, the users may not even be aware that their interpretation is different, leading to a modeller/customer disconnect, which may not be noticed.

- Similarly, state machines might be shared between modellers, or ported from one modelling environment to another. If the participants are not aware of the potential problems of dialect, well-formedness and execution behaviour, these state machines cannot be shared or ported accurately.

- Finally, tool developers should also be aware of these differences and potential problems in order to gear their tools to particular dialects. Tool developers may also offer import/export capabilities; our work indicates the parts of a state machine that must be translated or otherwise modified. In addition, the development of syntax and well-formedness checkers can benefit from knowledge of these differences.

Table 1: Summary of differences between classical, UML and Rhapsody state machine dialects. Left-hand columns summarize syntactic and semantic differences. Right-hand columns indicate the severity of problems caused by these differences

Construct/Concept	UML	Class.	Rhap.	Note	Notation	Well-Form.	Behaviour
Syntax							
States							
entry/exit actions	●	⊙	●	1			✓
do-activity	●	⊙	⊙	2		✓	
deferred events	●	⊗	⊗			✓	
Pseudostates							
initial	●	●	●	3			
final	●	●	●	4	✓		
fork	●	⊙	⊙	5	✓	✓	
join	●	⊙	⊙	5	✓	✓	
shallow history	●	⊙	⊗	6		✓	
deep history	●	⊙	⊙	6		✓	
junction (static)	●	●	⊙	7		✓	✓
conditional (static)	N/A	⊕	⊕	8	✓		✓
choice (dynamic)	●	⊗	⊗	9		✓	
Transitions							
event trigger	●	⊙	⊙	10		✓	
action (behaviour)	●	⊙	●	1			✓
completion	●	⊘	⊘	11			✓
Semantics							
simultaneous events	N/A	⊕	N/A	12		✓	✓
simultaneous actions	N/A	⊕	N/A	13			✓
priority	●	⊙	●	14			✓

1 Multiple actions are permitted on a transition (or as entry/exit actions) in all formalisms; see Section 3.3 for how execution of these actions differs.
2 Classical and Rhapsody offer a 'static reaction' construct, which may also have triggers and guards. In addition, Classical statecharts allow multiple (potentially simultaneous) static reactions for a particular state [45, Sect. 6.1.1].
3 Called 'default' [43].
4 Called 'termination connector'; symbol is a circled 'T' in classical/Rhapsody statecharts [45, 41].
5 Symbol is slightly different. See Section 3.4.
6 UML allows history in orthogonal states. Rhapsody does not support shallow history.
7 Not used for static choice in Rhapsody. See Section 3.5.
8 Equivalent to junction; removed from UML [29, Sect. 3.4.3]. See Section 3.6.
9 See Section 3.7.
10 Classical allows conjunction and negation of triggers [117], as well as disjunction. UML does not permit conjunction or negation [117, 87]. Rhapsody does not support conjunction [41] or disjunction [52], or presumably, negation.
11 Completion events and transitions are not mentioned in classical or Rhapsody statecharts; although null transitions are permitted.
12 See Section 3.1.
13 See Section 3.3.
14 See Section 3.2.

Legend for Left-Hand Columns

Symbol	Description
●	supported, with little or no difference from UML 2.0
⊙	supported, with considerable difference from UML 2.0
⊗	definitely not supported (direct evidence)
⊘	presumably not supported (indirect evidence)
⊕	not supported by UML, but supported by other formalism(s)
N/A	not applicable

4 Semantic Approaches to UML State Machines

Of the three dialects examined, only one, UML, has become the *de facto* industry standard for general purpose modelling. A common criticism of UML is that its semantics, especially with respect to behaviour, are inadequate. The UML 2.0 specification [91] contains much detailed prose about semantics; however, it does not adopt a formal description 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. For instance, a formal semantics could highlight problems in the standard and enable the development of powerful and interoperable analysis and transformation tools for UML models.

The remainder of this paper provides a categorization and comparison of 26 different approaches to formalizing the semantics of state machines, specifically UML 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 research 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 underlying semantic formalism, analysis goal, or state machine feature and determine which approaches, or types of approach, are most suitable to their needs.

Specific technical details of the surveyed formalisms will not be provided. Readers are encouraged to consult the referenced papers for more information about the approaches. Alternatively, a technical report [24] provides a high-level overview of each approach.

4.1 Related Work

UML state machine diagrams are an object-based variant of classical statecharts, which have evolved over the years since Harel first introduced the dialect [39]. Although much research has been devoted to the semantics of classical statecharts ([45],[77], and [96] 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,[9] there are other factors which make the two state machine dialects 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 [91]; classical statecharts still allow simultaneous processing of events. As another example, the implicit priority system between the two dialects 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'. Due to these differences, existing semantic approaches for classical statecharts are not directly applicable to UML state machines.

[9] 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 [40]. This removed many of the paradoxes (such as instantaneous states and self-triggering transitions) discussed in [116].

There exists much published research relating specifically to semantic approaches to UML in general and to state machines in particular. However, there has been little work published on the categorization and/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. Fortunately, most research contains a 'related work' section, and several of these publications have been particularly useful. For instance, [26] provides an orthogonal division of related work, including categories such as 'level of UML coverage' and 'loose' vs. 'precise' semantics. [51] 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 an appropriate formalism). Finally, as one of the more recent publications, [58] provides an excellent overview of many different approaches.

5 Categorization of Semantic Approaches

The state machine formalisms 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 6. There are three broad categories of underlying formalism, each with several sub-categories, as shown in Figure 17.

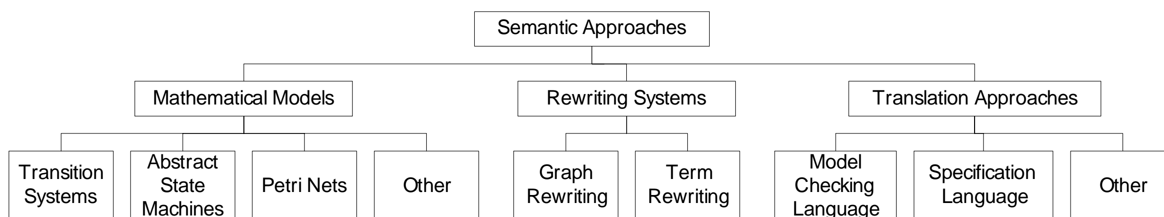


Figure 17: Primary categorization of semantic approaches for UML state machines

5.1 Mathematical Models

This category comprises semantic approaches that 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" [112] 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) [38] basically consists of a set of states and an iteratively applied update rule [60] and can be used, for instance, for the operational description of algorithms [58]. Although ASMs can be considered transition systems [14], we have kept the two formalisms separate, as in [112]. 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. Control flow is modelled through the use of tokens. Transitions are enabled when there are a sufficient number of tokens in the places 'before' them. 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 [82] exist.

Other Mathematical This sub-category holds other mathematical approaches that do not make use of transition systems, ASMs or Petri nets, e.g., basic sets-and-relations formalisms.

5.2 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 [112] 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" [112]. Graph rewriting approaches are a natural fit for state machines since there is no need to make the leap from graphical notation to a 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).

5.3 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 [79] or PROMELA/SPIN [48]. 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 [23, 108]. 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 [109] or PVS [92].

Other Translation This sub-category is a catch-all, currently containing one approach that translates state machines to concurrent regular expressions ([57]) and two approaches that translate state machines into axiomatic systems ([69] 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 could be:

Programming Languages Approaches here would essentially generate code from UML statechart diagrams. [105, 64] 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 state machine diagrams to internal representations of tools, such as Rose RT, RHAPSODY, etc. The last two are large, commercial, tools, but there exist many less well-known tools which allow for code generation, simulation, and analysis of UML state machine diagrams.

5.4 Overlap in Reference Set

The primary categorization provided here is not orthogonal. For instance, graph rewriting can be considered mathematically precise [112]. In addition, some approaches fit comfortably into more than one category, especially since several make use of more than one underlying formalism. Figure 18 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.

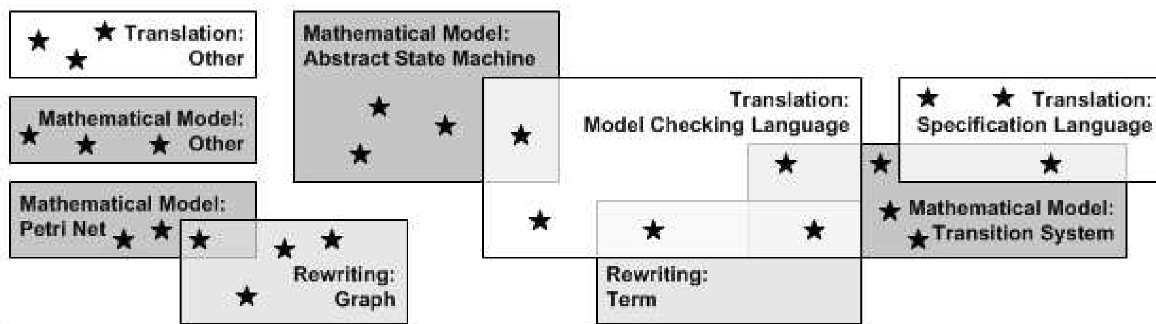


Figure 18: Overlap between primary sub-categories, based on current set of references. Each ★ represents a surveyed approach

Our set of references contains 39 approach-specific publications, covering 26 separate semantic approaches to formalizing UML state machines. These references are listed in Table 2, 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 18.

6 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. We 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.

Table 2: Categorization of semantic approach references

Category/Sub-Category		**References** (one approach per cell)					
Mathematical Models							
	Abstract State Machines	[14]	[23] [108] [22]	[58]	[60]		
	Transition Systems	[26]	[31]	[35] [71] [72]	[67]	[99] [97]	[117]
	Petri Nets	[8]	[10] [82]	[63]			
	Other	[17] [16]	[33]	[81]			
Rewriting Systems							
	Graph Rewriting	[8]	[30]	[36] [65] [66]	[112]		
	Term Rewriting	[67]	[73] [74]				
Translation Approaches							
	Specification Languages	[5] [110]	[99]	[120] [119]			
	Model Checking Languages	[23] [108] [22]	[35] [71]	[67]	[73] [74]	[105] [64]	
	Other	[4]	[57]	[69]			

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 state machine 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.

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.

6.1 UML Coverage

The viability of a particular semantic approach should depend in part upon how well it addresses the features of UML state machines. Tables 4, 5 and 6 show in detail how well each approach covers specific UML state machine features. Table 4 details the mathematical model approaches, Table 5 the rewriting approaches, and Table 6 the translation approaches.

Table 3 provides the legend for these three tables. Entries marked with ● indicate that the formalism supports the specific state machine feature with little or no difference from the description in the UML 2.0 standard. Entries marked with ⊙ indicate that the feature is supported; however, there is a significant difference from the description in the standard. For example, [97][99] supports initial states but only one initial state per state machine is permitted, while in UML, each sequential composite state can have an initial state and each orthogonal composite state can have multiple initial states.

Entries marked with ⊗ indicate that the documentation for the specific approach clearly states that the state machine feature is not supported. On the other hand, ⊘ indicates that there is indirect evidence that a particular feature is not covered. For instance, [97][99] clearly state that forks and joins are not supported. The same documentation makes absolutely no mention of the concept of priority between conflicting transitions, thereby leading to the implication that a specific priority scheme is not supported.

Finally, entries marked with ○ indicate that there is simply not enough information to make even an educated guess about the coverage of certain state machine features. Several publications were written at such a high level that little or no coverage information could be determined.

The left-hand column of Tables 4, 5 and 6 list most of the features of UML 2.0 state machines. This list includes most of the features of *behavioral state machines* according to [91] and gives a general idea of which features are best covered by our set of semantic approaches. The following features are not included: submachines and 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.).

Table 3: UML feature coverage legend for Tables 4, 5 and 6

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

Table 4: State machine feature coverage of mathematical model approaches. The legend is listed in Table 3. Several publications were written at such a high level that little or no coverage information could be determined

	Mathematical Model Approaches															
	ASMs				Transition Systems						Petri Nets			Other		
Feature	[14]	[23] [108] [22]	[58]	[60]	[26]	[31]	[35] [71] [72]	[67]	[97] [99]	[117]	[8]	[10] [82]	[63]	[16] [17]	[33]	[81]
States																
entry/exit actions	●	●	●	●	○	●	⊗	○	⊗	●	○	●	○	○	●	⊗
internal transitions	●	⊗	●	●	○	○	⊗	○	⊗	○	○	●	○	○	●	⊗
sequential (OR)	●	●	●	●	○	●	●	●	●	●	○	⊗	⊘	○	●	●
orthogonal (AND)	●	●	●	●	○	●	●	●	⊗	●	○	⊗	⊘	○	●	●
do-activity	●	●	●	●	○	⊗	⊗	○	⊗	⊗	○	●	○	○	●	⊗
deferred events	●	⊘	●	⊗	○	⊗	⊗	○	●	⊗	○	●	○	○	●	⊗
Pseudostates																
initial	●	●	●	●	○	●	●	●	⊙	⊗	○	●	●	○	●	●
final	●	○	●	●	○	●	○	○	⊙	⊗	○	●	●	○	●	○
fork/join	⊗	⊘	●	⊘	○	○	●	○	⊗	⊗	○	⊗	○	○	●	●
history	●	⊘	●	⊗	○	⊗	⊗	●	⊗	●	○	⊗	○	○	●	⊗
junction	●	⊘	●	⊘	○	○	⊗	○	⊙	⊗	○	⊗	○	○	⊗	⊗
choice	○	⊘	⊙	⊘	○	⊗	⊗	○	⊗	⊗	○	⊗	○	○	⊘	⊗
Transitions																
event trigger	●	●	●	⊙	○	●	●	●	●	⊙	○	●	●	○	●	●
guard	●	●	●	●	○	●	●	●	●	⊗	○	⊗	⊘	○	●	●
action/behavior	●	⊘	●	⊙	○	●	●	●	●	●	○	●	●	○	●	●
priority scheme	●	●	●	○	○	●	●	●	⊘	●	○	○	○	○	●	●
interlevel trans.	●	●	●	⊗	○	●	●	●	●	●	○	⊗	○	○	●	●
Miscellaneous																
completion event	●	●	●	●	○	●	○	○	⊗	⊗	○	●	○	○	⊗	○

Table 5: State machine feature coverage of rewriting approaches

	Rewriting Approaches					
	Graph				Term	
Feature	[8]	[30]	[36] [65] [66]	[112]	[67]	[73] [74]
States						
entry/exit actions	○	○	○	⊘	○	●
internal transitions	○	○	○	⊘	○	●
sequential (OR)	○	⊗	●	●	●	●
orthogonal (AND)	○	⊗	●	●	●	●
do-activity	○	○	⊗	⊘	○	●
deferred events	○	○	⊗	⊗	○	●
Pseudostates						
initial	○	○	●	●	●	●
final	○	○	●	⊘	○	●
fork/join	○	⊗	○	⊘	○	●
history	○	⊗	⊗	⊗	●	●
junction	○	⊗	○	⊘	○	⊙
choice	○	⊗	○	⊘	○	○
Transitions						
event trigger	○	⊙	●	●	●	●
guard	○	⊗	⊗	●	●	●
action/behavior	○	⊙	⊗	●	●	●
priority scheme	○	○	●	●	●	●
interlevel trans.	○	⊗	●	●	●	●
Miscellaneous						
completion event	○	⊘	●	○	○	●

Table 6: State machine feature coverage of translation approaches

	Translation Approaches										
	Specification			Model Checking					Other		
Feature	[5] [110]	[99]	[119] [120]	[23] [108] [22]	[35] [71]	[64] [105]	[67]	[73] [74]	[4]	[57]	[69]
States											
entry/exit actions	●	⊗	○	●	⊗	●	○	●	○	⊘	●
internal transitions	⊗	⊗	○	⊗	⊗	○	○	●	○	⊘	○
sequential (OR)	●	●	●	●	●	●	●	●	○	●	●
orthogonal (AND)	●	⊗	●	●	●	●	●	●	○	●	●
do-activity	●	⊗	○	●	⊗	●	○	●	○	⊘	○
deferred events	●	●	○	⊘	⊗	○	○	●	○	⊘	⊗
Pseudostates											
initial	●	⊙	●	●	●	●	●	●	○	●	○
final	●	⊙	●	○	○	●	○	●	○	○	○
fork/join	●	⊗	○	⊘	●	●	○	●	○	●	○
history	●	⊗	○	⊘	⊗	●	●	●	○	○	⊗
junction	●	⊙	○	⊘	⊗	⊙	○	⊙	○	○	○
choice	●	⊗	○	⊘	⊗	○	○	○	○	●	○
Transitions											
event trigger	●	●	●	●	●	●	●	●	●	●	○
guard	●	●	●	●	●	●	●	●	●	●	⊙
action/behavior	●	●	⊙	⊘	●	●	●	●	⊙	●	○
priority scheme	⊗	⊘	○	●	●	●	●	●	○	⊗	○
interlevel trans.	⊗	●	●	●	●	●	●	●	○	●	○
Miscellaneous											
completion event	●	⊗	○	●	○	●	○	●	○	○	○

An overall examination of the information presented in Tables 4, 5 and 6 suggests the following conclusions:

- ASM approaches offer better coverage most other mathematical formalisms.
- For the most part, the graph rewriting approaches do not provide very good coverage of state machine features.
- Model checking approaches offer above-average coverage, especially with respect to the features dealing with transitions.
- Those features related to pseudostates are the least well covered, compared to features related to actual states or transitions.

Figure 19 also lists the UML state machine features and gives a general idea of which features are best covered by our set of semantic approaches.

The following conclusions can be drawn from the information summarized in Figure 19:

- 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% (18/26) 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% (4/26) 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% (8/26) 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 19 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 29 with respect to dynamic choice.

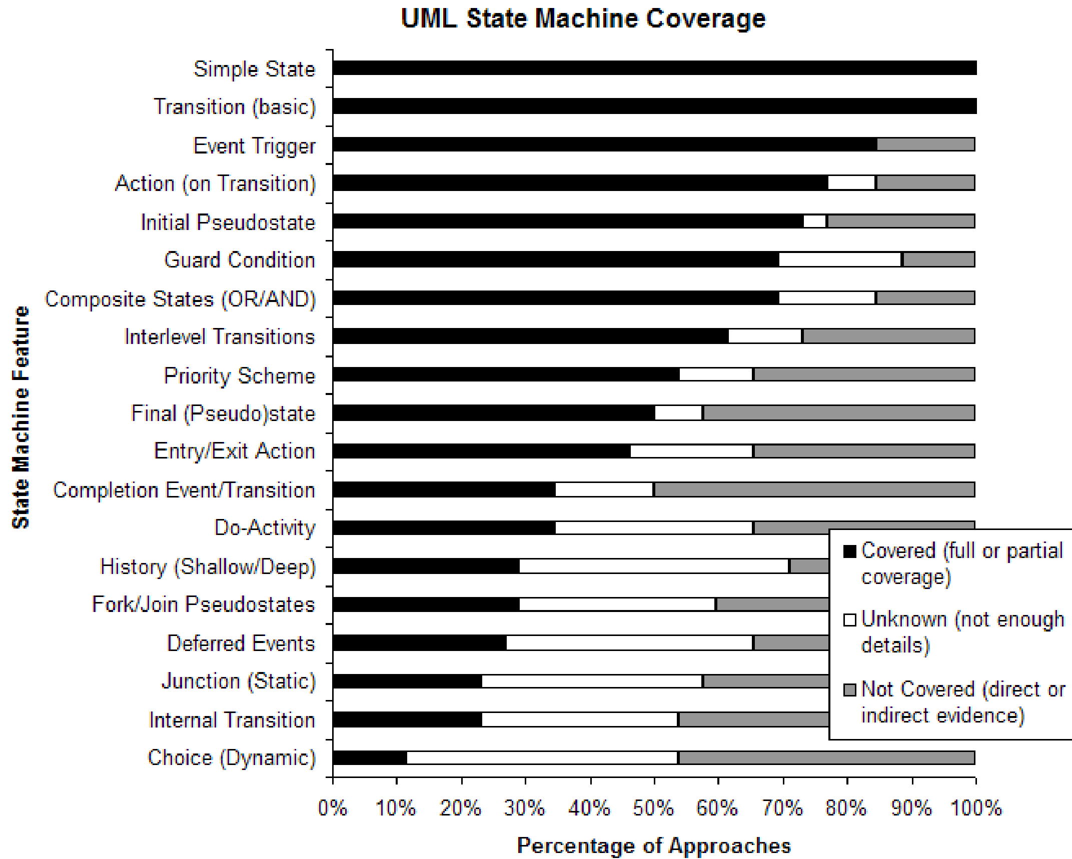


Figure 19: UML state machine features, ordered by coverage level. The features at the top are specifically covered by all or a majority of approaches; those at the bottom are specifically covered by few approaches. Most approaches are quite explicit with respect to which features they do or do not cover (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)

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 7 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.

The following conclusions can be determined by the information summarized in Table 7:

- 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

Table 7: 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%

case that researchers have simply not expanded the formalism to handle these more interesting features. While the lack of coverage by these approaches does not necessarily indicate a fundamental limitation, but more like a conscious decision by the authors to emphasize the treatment of different diagram types rather than all state machine features, it does appear that Petri nets and graph rewriting are not particularly suitable for handling the more interesting UML state machine features.

- While Petri net approaches cover the least number of features shown in Table 7, ASM approaches cover the most. At least one ASM approach claims to cover all 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 [58] does not allow for variables, which means that choice pseudostates are no different from junction pseudostates. The other approach [57] 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. [14] and [74, 73] treat junction as syntactic sugar. [99] 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 ([74, 73] and [105, 64]) draw junction as a diamond, i.e., the symbol for choice.[10]

- Priority is reasonably well covered, at least by most of the sub-categories. Two approaches ([31] and [35, 71]) make use of a parameterized priority scheme, which allows for either a bottom-up or top-down priority.

- There is exactly one approach [58] which handles all five of these features; however, as discussed above, the support for dynamic choice does not allow for variables. However,

[10] 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.

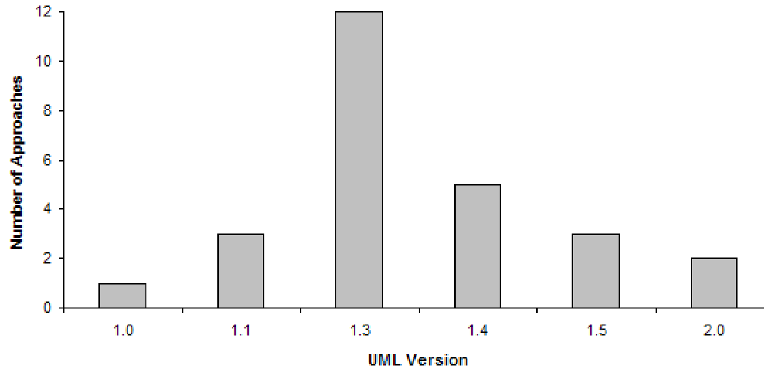


Figure 20: Histogram showing which versions of UML are supported

this approach is one of only a handful which support a great majority of the state machine features. In addition to [58], four other approaches cover a majority of the features: [14] (Abstract State Machine, missing fork/join); [33] (Other Math, missing junction/choice); [73, 74] (Term Rewriting and Model Checking Language, missing choice, junctions translated to simpler constructs); and [105, 64] (Model Checking Language, missing internal transitions, choice, syntactic difference with junction).

Another way of looking at coverage is to consider the version of UML that approaches are covering. The histogram in Figure 20 indicates that a majority of approaches refer to UML 1.3, i.e., 1999-2000. Although there are only two approaches that cover UML 2.0 ([33] and [120, 119]), 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.

6.2 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: [108].

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: [108].

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 [105, 64]. Another form of consistency checking is suggested by [69], 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], [35], [67], [73, 74], [105, 64], [108].

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], [58].

It is interesting to note that 35% (10/26) of the approaches surveyed propose some type of analysis that can be performed on or with UML state machines. Even more usefully, all but one of these approaches provide for some sort of tool support.

6.3 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 8 lists which tools have been used by which approaches, and for which types of analysis.

Table 8: 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 [32]		[58]
	JACK editor [15]		[35]
	F-Developer [4]		[4]
Syntax (vs. metamodel)	unnamed	XASM [3]	[108]
Well-formedness (vs. OCL)	unnamed	XASM [3]	[108]
Consistency	unnamed	GreatSPN-to-PROD [27]	[10]
	HUGO [105]	SPIN [48]	[105, 64]
	F-Verifier [4]	HOL [2]	[4]
Model checking	unnamed	SMV [79]	[108]
	unnamed	SMV [79]	[67]
	unnamed	PROD [111]	[10]
	JACK [15]		[35]
	HUGO [105]	SPIN [48] UPPAAL [70]	[105, 64]
	vUML [74]	SPIN [48]	[74, 73]
Animation/Simulation	Moses [32] F-Prototyper [4]	 ML [93]	[58] [4]

6.4 Comparison Summary

Sections 5 and 6 represent 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 with transitions between them. The same argument can be made for Petri nets as well.

- 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 language, 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.

7 Conclusion

MDD is a software development process that focuses on the models of a software system. These models are transformed into code. The concepts of executable models, automatic transformation, and validation are key components of MDD. Statecharts, or state machines, are a common mechanism for modelling the behaviour of model elements.

There are currently three state machine dialects well-represented in the research literature. Although very similar, there are enough subtle syntactic and semantic differences between the three dialects to make it difficult to correctly interpret a state machine unless the underlying dialect is well-understood. This paper helps to address this difficulty by clearly documenting many syntactic and semantic differences between the dialects. The results of this research are useful to both modellers and customers in fostering better communication. Also, tool developers can make use of this research to create better modelling, analysis, and transformation tools.

A common complaint about the Unified Modelling Language is that it does not include a formal semantics. This deficiency can be understood, given the fact that there are thirteen diagram types in UML 2.0. However, even UML state machines, which are based on the well-understood notion of finite state machines, cannot claim to be completely formalized.

We have gathered information about 26 different attempts to formalize UML state machines from the research literature. We have categorized them according to the underlying formalism and compared them along several secondary dimensions, most notably, UML state machine coverage. This particular research can be used as a useful starting point with respect to learning about the semantics of state machines. The results of this research can be used to guide the reader to relevant publications, based on particular formalisms, analysis goals or state machines features. In addition, this research raises several interesting questions, suitable for future work.

7.1 Future Work

With respect to the comparison of state machine dialects, the following topics could be considered for future work:

- Continue adding to the dialects covered, especially those dialects employed by specific tools.

- With an eye to creating an algorithm for porting a state machine from one dialect to another, compile a detailed list of constraints that must be met in order for a

state machine to be considered portable. For example, a classical statechart employing simultaneous events cannot be ported to the other two dialects.

- Create workable algorithms for porting between dialects. For example, a dynamic choice in a UML state machine can be simulated in Rhapsody with a junction pseudostate and the addition of a new composite state.

With respect to semantic approaches to formalizing UML state machines, there are a number of open research questions and issues:

- 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 [17] 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?

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] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

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

[13] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[14] 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.

[15] 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).

[16] 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.

[17] 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.

[18] 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`.

[19] 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.

[20] S. Christensen, J.B. Joergensen, and L.M. Kristensen. Design/CPN - a computer tool for coloured Petri nets. In *Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 1997.

[21] 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.

[22] 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.

[23] 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.

[24] M.L. Crane and J. Dingel. On the semantics of UML state machines: Categorization and comparison. Technical Report 2005-501, School of Computing, Queen's University, 2005.

[25] 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)*, volume 3713 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2005.

[26] 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.

[27] 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.

[28] B.P. Douglass. UML statecharts. Whitepaper, I-Logix.

[29] B.P. Douglass. *Real Time UML*. Object Technology Series. Addison-Wesley, third edition, 2004.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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.

[37] M. Gogolla, P. Ziemann, and S. Kuske. Towards an integrated graph based semantics for UML. *Electronic Notes in Theoretical Computer Science*, 72(3):1–16, 2003.

[38] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Brger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

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

[40] 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.

[41] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

[42] D. Harel and H. Kugler. The RHAPSODY semantics of statecharts (on, on the executable core of the UML) (preliminary version). In *SoftSpez Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer-Verlag, 2004.

[43] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[44] D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64. Computer Society Press of the IEEE, 1987.

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

[46] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer Magazine*, 37(10):64–72, 2004.

[47] B. Hnatkowska and Z. 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.

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

[49] D. Howe (editor). The free on-line dictionary of computing. `http://www.foldoc.org/`.

[50] C. Huizing and W.P. de Roever. Introduction to design choices in the semantics of statecharts. *Information Processing Letters*, 37(4):205–213, 1991.

[51] 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.

[52] I-Logix. *Rhapsody 6.0 User Guide*.

[53] I-Logix. Rhapsody web site. `http://www.ilogix.com/rhapsody/rhapsody.cfm`.

[54] I-Logix. Statemate web site. `http://www.ilogix.com/statemate/statemate.cfm`.

[55] I-Logix. *Tutorial for Rhapsody in J (Release 4.1 MR2)*, 2003.

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

[57] 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.

[58] 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.

[59] R. Joehanes. Incorporating UML state charts into Bandera. Master's thesis, Kansas State University, Department of Computing and Information Science, 2002.

[60] 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.

[61] S. Kent, A. Evans, and B. Rumpe. UML semantics FAQ. In *Object-Oriented Technology, ECOOP'99 Workshop Reader*, volume 1743 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 1999.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

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

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statecharts semantics. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 120–129. ACM Press, 2000.

[76] Erich M. *Semantics and Verification of Statecharts*. PhD thesis, Christian-Albrechts University of Kiel, 2000. Bericht Nr. 2011.

[77] 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.

[78] R.C. Martin. UML tutorial: Finite state machines. Engineering Notebook Column, C++ Report, 1998.

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

[80] S. Meng, Z. Naixiao, and B.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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] OMG. MDA web site. `http://www.omg.org/mda/`.

[86] OMG. UML specification. Document ad/99-06-08, Object Management Group, 1997. Version 1.3.

[87] OMG. UML specification. Document formal/03-03-01, Object Management Group, 2003. Version 1.5.

[88] OMG. UML 2.0 infrastructure specification. Document ptc/03-09-15, Object Management Group, 2004.

[89] OMG. UML 2.0 superstructure specification. Document ptc/03-08-02, Object Management Group, 2004.

[90] OMG. UML 2.0 superstructure specification. Document ptc/04-10-02, Object Management Group, 2004.

[91] OMG. Unified Modeling Language: Superstructure version 2.0. Document formal/05-07-04, Object Management Group, 2005.

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

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

[94] 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.

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

[96] 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.

[97] G. Reggio. Metamodelling behavioural aspects: the case of the UML state machines (complete version). Technical Report DISI-TR-02-3, DISI-Università di Genova, Italy, 2002.

[98] 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.

[99] 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.

[100] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[101] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2005.

[102] B. Rumpe. A note on semantics (with an emphasis on UML). In *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 177–197. Technische Universität München, TUM-I9813, 1998.

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

[104] O. Rysavý. A survey on approaches to formal representation of UML. Technical report, Brno University of Technology, 2003.

[105] 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.

[106] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[107] B. Selic. Personal Communication, 2005.

[108] 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.

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

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

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

[112] 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.

[113] 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.

[114] D. Varró. Automated formal verification of visual modeling langauges by model checking. *Software Systems Modelling*, 3:85–113, 2004.

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

[116] 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.

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

[118] R. Wieringa and J. Broersen. Minimal transition system semantics for lightweight class- and behavior diagrams. In *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München TUM-I9803, 1998.

[119] 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.

[120] 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.