UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal

Michelle L. Crane and Juergen Dingel

School of Computing, Queen's University
Kingston, Ontario, Canada
`{crane,dingel}@queensu.ca`

Abstract. State machines, represented by statecharts or statechart diagrams, are an important formalism for behavioural modelling. According to the research literature, the most popular statechart formalisms appear to be Classical, UML, and that implemented by RHAPSODY. These three formalisms seem to be very similar; however, there are several key syntactic and semantic differences. These differences are enough that a model written in one formalism could be ill-formed in another formalism. Worse, a model from one formalism might actually be well-formed in another, but be interpreted differently due to the semantic differences. This paper summarizes the results of a comparative study of these three formalisms with the help of several illustrative examples. Then, we present a classification of the differences together with a comprehensive overview.

1 Introduction

Model driven development (MDD) is a software development process that has been gaining in popularity in recent years. MDD focuses on the models, or abstractions of the software system, rather than on the final programs [20]; these models are transformed, automatically or manually, 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 initiative). Within MDD, state machines are a popular way of modelling the behaviour of systems.

With respect to state machines, the most popular formalisms, as represented in the research literature, are UML statechart diagrams (as specified in UML 2.0 [18]), Classical Harel statecharts (implemented in STATEMATE [9, 11]), and a newer object-oriented version of Harel's statecharts (implemented in RHAPSODY [8]). These three formalisms appear to be very similar. For instance, at first glance, a model written in one formalism could be easily ported to one of the other two formalisms. However, there are some subtle syntactic and semantic differences between the formalisms which can lead to pitfalls. Consider, for example, the state machines shown in Fig. 1. The two machines are identical, except for the notation used to represent static choice. Fig. 1(a) makes use of a junction (small filled circle); this machine is well-formed in the Classical and

UML formalisms. Fig. 1(b) shows a condition construct (circled 'C'), which is used by both the Classical and RHAPSODY formalisms. Ignoring the notation difference for a minute, this model is well-formed in all three formalisms. However, the behaviour exhibited by the state machine is different for all three. When the state machine first starts, it moves to state A, at which point the variable $x = 0$. All three formalisms agree on this point. What they do not agree on is what happens when event e occurs. In the Classical formalism, the state machine moves to state D. In the UML formalism, the state machine moves to state B. Finally, in the RHAPSODY formalism, the state machine moves to state C.
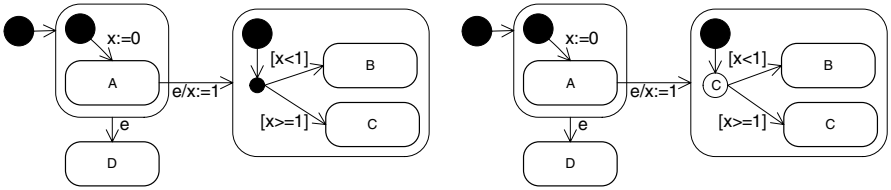
(a) Junction (small filled circle) used for static choice. Model is well-formed in Classical and UML formalisms

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

Fig. 1. Ignoring notation differences, this model is well-formed in all three formalisms, 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. State machine inspired by [8]

The fact that there can be three distinct interpretations of one state machine indicates that there is a lack of standardization between the three formalisms. It also indicates that the task of transforming, or porting, a model from one formalism to another may not be straightforward. Therefore, it is worthwhile to study the syntactic and semantic differences between the most popular formalisms. In this paper, we present a detailed comparison of these three formalisms, including several illustrative examples. Our results are of interest to modellers, customers, and tool developers because they summarize the differences between the three most popular formalisms and thus help to avoid the pitfalls of incorrectly in-

terpreted models. On the one hand, modelling and transformation tools must correctly implement the syntax and semantics of a formalism (or more than one, if the tool is expected to import/export models). On the other hand, the modellers and customers who make use of models to communicate must also be conversant in these details in order to communicate effectively.

This paper is organized as follows: Section 2 briefly describes state machines and the three formalisms. Section 3 contains a detailed comparison of syntactic constructs and semantic concepts which differ between the three formalisms, while a tabular summary is presented in Section 4. Section 5 discusses related work. Finally, Section 6 contains the conclusion and contributions of this work.

2   State Machines, Statecharts and Statechart Diagrams

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, together with its responses to those events" [2, Ch. 2]. FSMs are very useful for representing reactive systems. 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'.

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" [5]. 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 [5, 14]. Other publications by Harel and other authors quickly followed, defining a preliminary semantics for the statecharts formalism [10, 19]. Far from being a final product, the statecharts formalism evolved over the years, spawning many variants. In fact, as of 1994, there were at least 20 variants of these statecharts [22]. In 1996, Harel revisited the formalism, modifying some of the previous semantics [6, 9]. 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 which statecharts we refer to. 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 [6, 9, 11]. Although Harel himself states that there is no official semantics for his statecharts [9], Classical statecharts are actually implemented in I-Logix's STATEMATE tool, to which Harel has contributed.

The Unified Modeling Language (UML) 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" [17, 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 statechart diagrams are one diagram type that can be used to model intra-object behaviour, i.e., how individual model elements behave. A statechart diagram is used to represent a state machine. 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 latest draft of the UML 2.0 Superstructure specification [18].

UML statechart diagrams are an object-based variant of Classical statecharts [18, 16, 4]. An alternative object-based variant is one to which Harel himself has contributed: the statechart formalism implemented in I-Logix's Rhapsody tool. This formalism was created after the introduction of UML 1.1. Actually, the Rhapsody formalism is more closely related to the UML formalism than to its Classical ancestor. In fact, there was cooperation between the Rhapsody and UML development teams, resulting in cross-pollination between the two formalisms [7, 21]. For the purposes of this paper, we concern ourselves with Rhapsody as it is documented in [7, 8].

3   Detailed Comparison

In general, all three formalisms are similar. Basically, statecharts[1] are directed graphs, consisting of states and transitions between them. Transitions may have labels of the form event[guard]/action. All three formalisms support both orthogonal (AND) and sequential (OR) composite states.

These basic similarities aside, there are several syntactic and semantic differences between the three formalisms. 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 statecharts from one formalism 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 formalisms and yet be represented with alternative notation. For example, a final state in UML is represented as "a circle surrounding a smaller solid filled circle" [18], while the Classical and Rhapsody formalisms make use of a circled 'T'. This category is the least critical; after a simple notation translation, a model would be compatible with the target formalism(s).

Well-Formedness Differences in this category are more important; they result in models that are well-formed in one or two formalisms, but not in all three. For instance, a construct may not be available in a particular formalism, or a formalism may enforce additional or different constraints on a common

---

[1] In the interests of simplicity, we refer to the diagrams of all three formalisms as *statecharts* and use the term *state machine* when referring to the model of computation.

construct. A model could be checked for compatibility with simple syntax or well-formedness checking. Translation and re-working of a model may make it compatible with the target formalism(s); however, not all models can be made fully compatible with all formalisms. For example, event triggers are not permitted after pseudo-states 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 model may be well-formed in more than one formalism 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 model 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. Several of the more interesting syntactic constructs are then examined.

3.1 Synchrony Hypothesis

Synchrony and Zero Time The *(perfect) synchrony hypothesis* [1] states that a system must react immediately to external events and that the corresponding output must occur at the same time [22]. The zero-time assumption follows from the synchrony hypothesis and implies that transitions take zero time to execute [16]. In general, Classical statecharts support both the synchrony hypothesis and the zero-time assumption [22, 16].[2]

In UML, a transition *may* take time [16], although no assumptions are actually made, allowing for models with either zero- or fixed-execution time [18, Sect. 13.3.30]. The RHAPSODY formalism mirrors that of UML in that a "step does not necessarily take zero time" [8]. Therefore, with respect to the zero-time assumption, it is theoretically possible that both the UML and RHAPSODY formalisms adhere to the synchrony hypothesis.

Synchrony and Simultaneous Events By the synchrony hypothesis, Classical statecharts must be able to react immediately to external events. They can do so, supported by the fact that different events may occur simultaneously, and be acted upon simultaneously, in Classical statecharts [15]. However, neither the UML nor RHAPSODY formalisms support the synchrony hypothesis in

---

[2] 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 [9].

this regard. Instead, both formalisms adhere to the concept of run-to-completion (RTC), which means that each event is handled completely before the next event is processed.[3]

It is thus impossible in a UML or RHAPSODY statechart for different events to be handled simultaneously.[4] For example, consider the statechart in Fig. 2. 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. However, in the other two formalisms, 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.
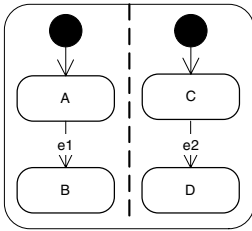
Fig. 2. Statechart with potentially simultaneous events
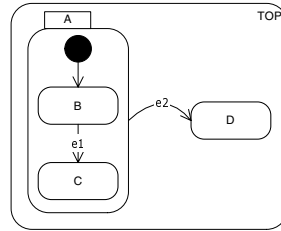
Fig. 3. Statechart with potentially conflicting transitions

3.2 Priorities of Conflicting Transitions

It is possible in all three formalisms 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 statechart in Fig. 3. Assume that the machine is currently in state B and that events e1 and e2 are generated. The two transitions enabled by these events are in conflict because their effects conflict. For instance, if the transition labelled e2 is taken, the state machine moves to state D, and the transition labelled e1 cannot be taken.

One of the most serious differences between the UML/RHAPSODY and Classical formalisms 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 [9, 15]. Priority is given to the transition with the highest scope. In the case of the statechart in Fig. 3, the scope of the transition labelled e1

---

[3] In UML, "event occurrences are detected, dispatched, and then processed...one at a time" [18, Sect. 15.3.12]. In RHAPSODY, events are handled "one by one, in order" [7].

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

is state A, while the scope of the transition labelled e2 is the state TOP. Since priority is given to the transition with the highest scope, event e2 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" [18, Sect. 15.3.12]. In RHAPSODY, lower level states also get priority [7]. In this case, the transition labelled e1 originates from state B, which is a substate of state A, the origin of the transition labelled e2. Since priority is given to the substates, event e1 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 [8].

3.3   Order of Execution of Actions

In all three formalisms, is it possible to list multiple actions (or behaviours) on a transition between two states, as shown in Fig. 4. 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 [9]. 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" [18, Sect. 15.3.14]. Similarly, in RHAPSODY, "actions are guaranteed to be performed in sequential order" [8]. For both UML and RHAPSODY therefore, at state B, $x = 1$ and $y = 5$.
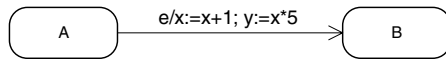

Fig. 4. Transition with a list of actions [9]

3.4   Fork and Join

Fork and join constructs are common to all three formalisms, although the notation is slightly different in Classical/RHAPSODY than in UML. Published work on the Classical and RHAPSODY formalisms show forks and joins as simply arrows with either multiple sources or multiple targets. The UML specification, as well as the RHAPSODY 6.0 tool itself [13], show separate fork and join constructs, which break the transitions into incoming and outgoing transitions.

In addition to the notational differences between the formalisms, 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 statechart in Fig. 5 would be ill-formed in Rhapsody, even with the alternate notation taken into account.
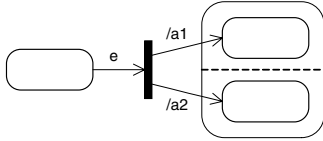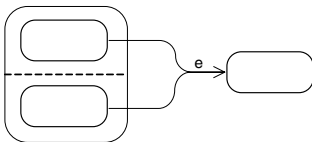

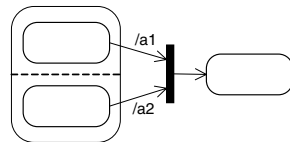**Fig. 5.** This UML fork would be ill-formed in Rhapsody
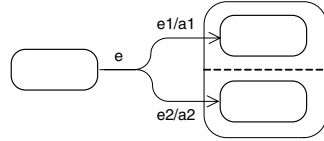
Fig. 6. This Classical fork would be ill-formed in both UML and Rhapsody

As another example, the Classical statechart in Fig. 6 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 formalisms. In the Classical formalism, 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 formalism 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.

The Classical statechart in Fig. 7 would be ill-formed in UML because UML does not allow for event triggers after the join pseudo-state. 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 pseudo-states. In fact, joins are not explicitly triggered in UML; they are only used with completion events [21], i.e., leaving the last state in each region of an orthogonal state. Finally, the UML statechart in Fig. 8 would be ill-formed in Rhapsody, since Rhapsody does not allow for any labels on transitions coming into a join.

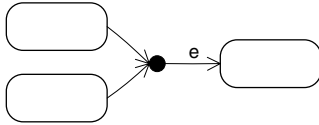**Fig. 7.** This Classical join would be ill-formed in UML

**Fig. 8.** This UML join would be ill-formed in Rhapsody

3.5   Junction

Junction constructs are common to all three formalisms, although there are some well-formedness differences. For example, the Classical statechart in Fig. 9 is ill-formed in UML. However, it is possible to make the statechart compatible by simply moving the event trigger to the transitions coming into the junction. In fact, each incoming transition may even have a different event trigger.

**Fig. 9.** This Classical junction can be made compatible to UML
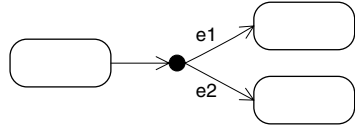
**Fig. 10.** This Classical junction would be ill-formed in UML and Rhapsody

In addition, the RTC assumption also affects the compatibility of the junction construct. For example, the Classical statechart in Fig. 10 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 pseudo-state. Finally, Rhapsody does not allow for more than one outgoing transition from a junction.

3.6   Conditional

Classical and Rhapsody statecharts support a specific conditional construct, such as that shown in Fig. 11. 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,[5] but its semantics can be mimicked with the standard junction pseudo-state, as shown in Fig. 12.
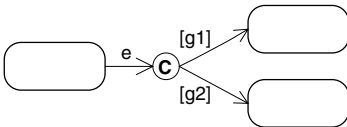
**Fig. 11.** Conditional construct supported by Classical and Rhapsody formalisms
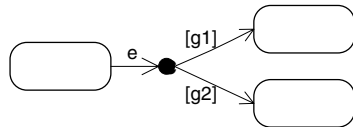
Fig. 12. UML supports the same static choice by using the junction pseudo-state

---

[5] The conditional construct was removed from UML 1.3, since it is equivalent to a junction [3, Sect. 3.4.3].

3.7   Choice

UML does allow for a *dynamic* choice pseudo-state, which is not equivalent to the Classical/RHAPSODY conditional construct. Consider the UML statechart in Fig. 13. 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.

Although neither the Classical nor RHAPSODY formalisms support this dynamic choice construct, it is possible to simulate it at least in RHAPSODY. Consider the RHAPSODY statechart in Fig. 14. In this case, the fact that RHAPSODY makes use of *microsteps* [8] 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.
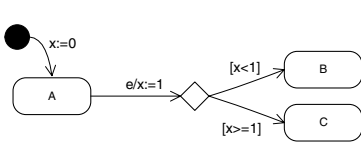

Fig. 13. UML supports dynamic choice
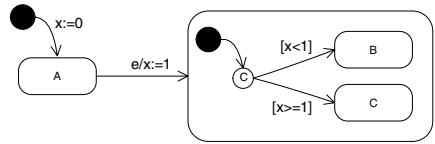
Fig. 14. Dynamic choice can be simulated in RHAPSODY

It is very important to note that even if the conditional in Fig. 14 were replaced by UML's static choice construct (junction), the state machine would not behave identically in UML. UML does not make use of microsteps, and the action along the transition will not be considered when the guards are evaluated [21]. If this state machine were to be evaluated in UML, it would move to state B.

3.8   More on Compound Transitions

In Classical statecharts, any composition of pseudo-states, simple transitions, guards and labels is permitted, but these transition compositions are constrained for practical purposes in UML state machines [16]. Therefore, there are some Classical statecharts which cannot be easily converted to UML. Consider, for example, the statecharts below. Fig. 15(a) shows a compound transition between two states. Both transitions are labelled with an event trigger and an action. In the Classical formalism, the transition coming into the junction cannot be executed without also executing the transition coming out of the junction [9]. Therefore, this compound transition is equivalent to the single transition in Fig. 15(b), which is labelled with the conjunction of two events, and a pair of resultant actions.

(a) Compound Transition   (b) Conjunction of Events
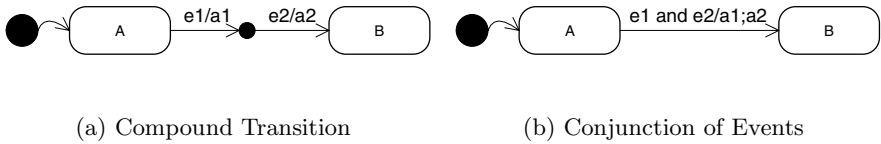
Fig. 15. Sample compound transition from Classical statecharts and its equivalent single transition (with conjunction of events) [9]

Neither of these equivalent state machines would be well-formed in UML. On the one hand, the state machine in Fig. 15(b) cannot be interpreted because UML does not allow for the conjunction of events. On the other hand, the state machine in Fig. 15(a) also cannot be interpreted because UML does not allow for triggers on transitions leaving a pseudo-state [18, Sect. 15.3.14].

4   Comparison Summary

Table 1 summarizes the findings of the previous 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 model in one formalism can be easily ported to the other formalisms with a simple notation translation.
- Differences in the well-formedness category are more serious. Sometimes it will be possible to modify a model to make it compatible to another formalism, e.g., the UML statechart in Fig. 12 represents the Classical/RHAPSODY statechart in Fig. 11. Unfortunately, not all models can be made compatible, e.g., the Classical statechart in Fig. 7 cannot be translated into an equivalent UML statechart.
- Finally, differences in execution behaviour are the most serious of all. This is not because they imply a model cannot be ported to another formalism, but because a model designed with constructs/concepts from this category can be well-formed in more than one formalism and yet behave differently in each. The statecharts in Fig. 1 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 statechart with deferred events[6] would be ill-formed in the other two formalisms. However, if the deferred

---

[6] 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 the state machine to recognize certain events (which do not trigger transitions) and postpone responding to them.

Table 1. Summary of differences between Classical, UML and Rhapsody statechart formalisms. 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	●	⊗	⊗			✓	
Pseudo-states							
initial	●	●	●	3			
final	●	●	●	4	✓		
fork	●	⊙	⊙	5	✓	✓	
join	●	⊙	⊙	5	✓	✓	
shallow history	●	⊙	⊗	6		✓	
deep history	●	⊙	⊙	6		✓	
junction (static)	●	●	⊙	7		✓	✓
conditional (static)	⊗	+	+	8	✓		✓
choice (dynamic)	●	⊗	⊗			✓	
Transitions							
event trigger	●	⊙	⊙	9		✓	
action (behaviour)	●	⊙	●	1			✓
completion	●	⊘	⊘	10			
Semantics							
simultaneous events	⊗	+	⊗	11		✓	✓
simultaneous actions	⊗	+	⊗	12			✓
priority	●	⊙	●	13			✓

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 [11, Sect. 6.1.1].

3 Called 'default' [9].

4 Called 'termination connector'; symbol is a circled 'T' [11, 7].

5 Notation 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 [3, Sect. 3.4.3]. See Section 3.6.

9 Classical allows conjunction and negation of triggers [23], as well as disjunction. UML does not permit conjunction or negation [23, 16]. Rhapsody does not support conjunction [7] or disjunction [12], or presumably, negation.

10 Completion events and transitions are not mentioned in Classical or Rhapsody statecharts; although null transitions are permitted.

11 See Section 3.1.

12 See Section 3.3.

13 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)

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 model is well-formed and behaves as expected. Instead, the behavioural problems indicated in Table 1 are *in addition* to any notational 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 formalisms, but it also brings to light several facts, such as:

- RHAPSODY is much syntactically and semantically closer to UML than to its Classical ancestor, especially with respect to behavioural semantics. This means that models can be more easily ported between UML and RHAPSODY than between either of these formalisms and Classical statecharts.
- UML is the only formalism that allows for dynamic choice.
- Many of the well-formedness and execution behaviour differences 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 formalisms is inverted, it does not cause any notation or well-formedness problems with the syntactic constructs. In other words, the fact that a model would behave differently due to the opposite priority schemes would not be found by a syntax or well-formedness checker.

5   Related Work

The UML 2.0 Semantics Project 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 statecharts could not be considered equivalent, even though at first glance, they appear almost identical.

Unfortunately, although there is much research relating to these formalisms, there is no definitive comparison between them. The most detailed comparison is a bulleted list in an older UML specification [16, Sect. 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 and UML statecharts, without going into great detail. The bulk of the research presented in this paper is thus a result of detailed inspection of the UML specification [18], as well as key documents relating to the Classical [5, 6, 9, 11] and RHAPSODY [7, 8] formalisms.

It should be noted that there are several other statechart-like formalisms linked to specific tools, such as RoseRT, AnyStates, LabVIEW, SmartState, etc. We have not considered these latter formalisms for two reasons: 1) many of these tool-specific formalisms claim to support UML and thus could be considered a subset of the UML formalism; and 2) these tools are not very well represented in the research literature.

6   Conclusion

There are currently three popular formalisms for modelling state machines: UML statechart diagrams, Classical statecharts and RHAPSODY statecharts. Modellers may adhere to MDD without being restricted to one particular formalism. In general, the similarities between Classical statecharts, UML statechart diagrams, and the statecharts implemented by the RHAPSODY tool are enough to imply to the non-expert that a state machine modelled in one formalism can be interpreted in the other formalisms. Unfortunately, this is not necessarily the case; there are enough syntactic and semantic differences between the formalisms 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 models which behave differently in different formalisms.

The results of this research are of interest to modellers, tool developers, and end users of statecharts and statechart diagrams for the following reasons:

- Modellers should be aware of how their models will be interpreted in different formalisms. This is especially important with respect to execution behaviour issues, where a modeller might be expecting a different behaviour than that exhibited by a model. In the same vein, statecharts can be used as a communication medium between modellers and their customers, or end users. Users may interpret these models differently, based on an alternate formalism 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, models might be shared between modellers, or ported from one modelling environment to another. If the participants are not aware of the potential problems of notation, well-formedness and execution behaviour, these models 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 formalisms. Tool developers may also offer import/export capabilities; our work indicates the parts of a model that must be translated or otherwise modified. In addition,

the development of syntax and well-formedness checkers can benefit from knowledge of these differences.

Future work on this particular topic includes adding in the formalisms supported by tools such as RoseRT, AnyStates, LabVIEW, SmartState, etc. Another possible avenue is to investigate the possibility of creating automatic or guided translations between the different formalisms.

Acknowledgements

References

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

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

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

[4] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In *Proc. Workshop on Precise Semantics for Modelling Techniques*, pages 55–72. Technische Universität München, TUM-I9803, 1998.

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

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

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

[8] D. Harel and H. Kugler. The RHAPSODY semantics of statecharts (on, on the executable core of the UML) (preliminary version). In *SoftSpez Final Report*, LNCS 3147, pages 325–354. Springer, 2004.

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

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

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

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

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

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

[15] E. Mikk. *Semantics and Verification of Statecharts.* PhD thesis, Christian-Albrechts University of Kiel, 2000. Bericht Nr. 2011.

[16] OMG. OMG Unified Modeling Language specification. Adopted Formal Specification formal/03-03-01, Object Management Group, 2003. Version 1.5.

[17] OMG. UML 2.0 infrastructure specification. Technical Report ptc/03-09-15, Object Management Group, 2004.

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

[19] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proc. Int'l Conf. on Theoretical Aspects of Computer Software*, LNCS 526, pages 244–264. Springer, 1991.

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

[21] Bran Selic. Personal Communication, March 2005.

[22] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 863, pages 128–148. Springer, 1994.

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