The Understandability of Models for Behaviour*

Vladimir Estivill-Castro[1 [0000−0001−7775−0780]] and René Hexel[1 [0000−0002−9668−849X]]

Griffith University, School of ICT, Nathan 4111, Australia
{v.estivill-castro,r.hexel}@griffith.edu.au

Abstract. Models are used mainly to communicate among humans the most relevant aspects of the item being modelled. Moreover, for achieving impact in modern complex applications, modelling languages and tools must support some level of composition. Furthermore, executable models are the foundations of model-driven development; therefore, it is crucial that we study the understandability of executable models of behaviour, especially from the perspective of modular composition. We consider the match between the delicate semantics of executable models for applications such as reactive-systems and real-time systems and the usually simple understanding of developers. By performing a series of experiments with UML statecharts and logic-labelled finite-state machines (LLFSMs) we explore understandability of event-driven versus logic-labelled state machines as well as the architectural options for modular composition. We find that expertise in manipulation of the models is essential, and that clarification of the semantics for logic-labelled finite state machines is necessary for LLFSMs to remain formally verifiable and suitable for robotic and embedded systems.

Keywords: Model understandability · state diagram · logic-labelled finite-state machines.

1 Introduction

"Models are often, although not always, designed to be viewed by humans. In such cases, the models must be clear and easy to understand. One way to ensure this is to use a modular approach in constructing the model" [47]. The use of models to conceptualise, construct, deploy, maintain, and improve software systems would be useless if such models were not understandable by humans, or at least expert software engineers. Model-Driven Software Development (MDSD) suggests models realise a higher level of description and abstraction, as well as a more human-like approach when specifying the behaviour of software systems. Higher abstraction, away from assembly language, has been the progression of programming languages [6], and it excels with the use of models.

Modelling is essential to communicate the representation of a system, module, or function from a particular perspective, with the precise intention of enabling

* Supported by Griffith University and Universiat Pompeu Fabra.

more comprehensive and productive analysis. But this is only true if models are understandable. Models abstract information; that is, models represent the same information as written specifications but in a more compact and compressed way. Once again, readers of such models would benefit if indeed there was no loss in translation. If the models are indeed understood, they enable focussing on relevant aspects and facilitate productive analysis, design, and deployment.

Modelling formalises requirements: *"So-called 'natural language' is wonderful for the purposes it was created for, such as to be rude in, to tell jokes in, to cheat, or to make love in (and Theorists of Literary Criticism can even be content-free in it), but it is hopelessly inadequate when we have to deal unambiguously with situations of great intricacy, situations which unavoidably arise in such activities as legislation, arbitration, mathematics, or programming."* [11].

In this paper, we review the mechanisms most commonly used to represent and model behaviour; namely, UML's statecharts [41]. We expand on a series of experiments [16], and re-iterate a series of experiments to evaluate the apparent symmetry of the `entry` and `exit` actions of the notations for a state. In contrast to earlier work [16], we cross international boundaries, cultural boundaries and language boundaries contrasting results across Australia and Spain. We confirm that humans familiar with the UML notation generalise rapidly and overlook the intricacies of the differences between these two constructs. Because MDSD uses well-established notations such as the UML, we believe it is essential to understand the profound implications for the semantics of UML forms for representing behaviour. The understandability of UML diagrams is crucial for correctness, validation, and formal verification of executable models. Software developers have highly ranked the understandability of representations among their criteria for the adoption of UML [38]. These practitioners argue against unnecessary model complexity and lack of formal semantics [38].

Our first quote highlights modularity in formal verification and model checking. Languages and tools for modelling behaviour must offer a mechanism to compose simpler behaviour into more sophisticated behaviours in order to describe the complex interactions and responses expected of sophisticated modern software systems in today's applications. Complex systems would not be achievable if it was not for some form of composition [48]. Therefore, in this paper, we explore the implications that nesting diagrams has as the mechanism to compose models of complex behaviours from models of simpler behaviours. Not surprisingly, this approach also runs into issues of understandability and, perhaps more seriously, scalability. Specially if one is to combine this with the mechanism of the subsumption architecture [5], such as the ability of one behaviour to suspend, and then restart or resume another behaviour.

Our observations are complementary to observations regarding the readability of formal notations: *"the familiarity with notation and structure that comes natural to [champions of formal notations] takes time, training and practice to acquire"* [20]. However, we explore further the implications for statecharts and behaviour modelling. For this, we take advantage of the theoretical and experimental validation of metrics on UML statecharts [23]. In particular, we measure

NEntryA (number of entry actions), NExitA (number of exit actions), and NCS (number of composite states). Although previous experiments [23] suggest these metrics are not to be correlated with the understandability of UML diagrams, our results indicate that NEntryA (number of entry actions) and NExitA (number of exit actions) are indeed relevant for understandability of a state diagram.

Those earlier experiments suggested an inconclusive correlation between the understandability of UML state diagrams and NA (number of activities), NSS (number of simple states), NT (number of transitions), and NG (number of guards) [23]. Follow-up experimentation [8, 7] reached the same conclusions.

We are of the opinion that besides the relevance of those metrics to the understandability of UML state diagrams there are other issues. We will stress the *asymmetry* of the `entry` and the `exit` actions. But we add to the list the event-driven nature of UML. In fact if we look at logic expressions to label transitions (as opposed to events), earlier research using such logic expressions in state machines represented in tabular form found that subjects handled the task with very high accuracy [51]. The other issue, as we already alluded to, is that, although abstraction and understandability had been heralded for nesting states, this was not so evident when used in experimental settings [8, 7, 51]. We claim here that the issues of nesting, and the asymmetry of `exit` versus `entry` are subtle, but crucial to understandability and have, thus far, not received sufficient detailed analysis.

We argue that symmetric rules for handling the sequencing of `entry` and `exit` actions, while simple and straightforward at first glance, represent a series of fallacies. Our results show that these rules are hard to comprehend and to apply by software developers, especially when timing issues and composition are involved. Also, defining a semantics that results in executable models for applications such as reactive systems and real-time systems is very delicate, especially if suspend/resume/restart signals cannot be ruled out between behaviours. Third, the apparently simple semantics actually results in a combinatorial explosion of states that hinders verification. We reach these conclusions from reviewing the results of a series of experiments with software developers and dissect how issues of understandability of state diagrams relate to nesting as well as being event-driven vs logic-labelled. We contrast this with the deterministic execution of logic-labelled finite-state machines (LLFSMs), which achieve model composition through a subsumption architecture using suspend/restart/resume. As a result, we propose a specific alternative semantics for LLFSMs that is suitable for robotic and embedded systems.

2 Background

Although not a primary reference, the English version of Wikipedia is a major source of influence and receives more than 20 billion queries a month. While the ultimate authority on the UML is its reference documentation [37], we postulate here that it would not be uncommon for software developers to use textbooks or sources such as Wikipedia for reviewing the intended meaning of the `exit` and `entry` actions of UML's states in statecharts. *"Every state in a UML statechart*

can have optional entry actions, which are executed upon entry to a state, as well as optional exit actions, which are executed upon exit from a state. Entry and exit actions are associated with states, not transitions. Regardless of how a state is entered or exited, all its entry and exit actions will be executed" [46]. The most natural assumption to make is that these actions are symmetric. However, it is not hard to discover that this is not the case. One could apply the technology of model-to-model transformations [32] to reduce UML statecharts with `entry` and `exit` actions into UML statecharts that do not have such actions. This model-to-model transformation is justified by the indication that the `entry` and `exit` actions of a state are abbreviations for what otherwise would be the inefficient use of multiple states. These actions are presented symmetrically as set-up and tear-down phases: *"The value of entry and exit actions is that they provide means for guaranteed initialisation and cleanup, very much like class constructors and destructors in Object-oriented programming"* [46]. However, the transformation makes explicit the asymmetry of the `entry` action with respect to the `exit` action. In the latter, we need to remember the target state of the transition in an intermediate state.

Unfortunately, there is not much of an improvement with the *Foundational UML* (`fUML`) — an executable subset of standard UML. `fUML` offers precise execution semantics. However, `fUML` uses Clause 15 of the UML Superstructure to define the execution semantics for statecharts. `fUML` description for the `entry` action and the `exit` action of a state is completely symmetric as it describes Alf [36, Page 328].

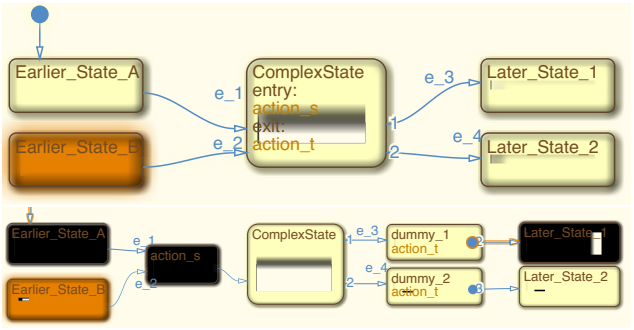


Fig. 1: Schema reflecting the model-to-model transformation that synchronises the `entry` and `exit` actions and illustrates their semantics.

The fundamental tool to handle complexity is to divide into coherent modules and compose back the global solution from the functionality of the components. The dominant mechanism to model complex behaviour in software systems and provide logical modularisation are hierarchies. The dominant form in state-based specifications is *nesting* [51]. In particular, nesting sub-states (so-called OR-decomposition) [45, Chapter 2]. Some consider nested states a *"great diagrammatic simplifications when a set of events applies to several sub-states"* [12].

Others [45, Page 69] regard *hierarchically nested states* [25], as the most important invention. However, Mellor [31] highlighted several difficulties and complex semantic issues. Nested states are a mechanism to produce common facilities and simplification to event-handling policies (similarly to the *Ultimate Hook Pattern*). A notation that implies inheritance is a very powerful abstraction for sharing common features (including behaviour) and perhaps substantive of object-oriented models and the UML in particular. This abstraction capacity mostly follows Liskov's Substitution Principle [30] and implies that a sub-state of a composite state has behavioural inheritance. However, in the case of states, the *is a* relationship of inheritance is replaced by *is in (is-in-a-state)* relationship [45, Page 72]. For example, the model in Figure 2[1] shows that when the

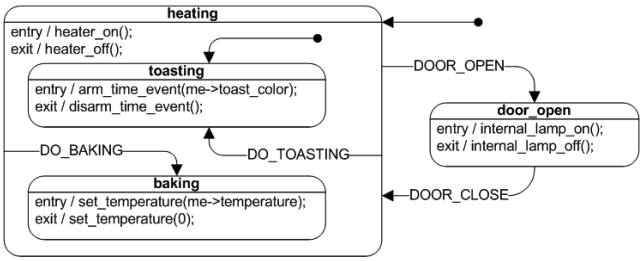


Fig. 2: A sample UML statechart.

system is in the **baking** state, it is *is-in* the **heating** state. The semantics of hierarchically nesting of states in UML is commonly specified with an interlingua approach which again is a model-to-model transformation that flattens the model. For hierarchically nesting, *"the Cartesian product machine is used as the interlingua semantics of statecharts"* [13, Page 63].

Another composition mechanism of the UML is orthogonal regions [45, Chapter 2] (so-called AND-decomposition, which also implies unconstrained concurrency). Along with class diagrams, UML statecharts are one of the top used artefacts [41]. However, there are also alternatives such as the already-mentioned subsumption architecture that enables layers of timed, logic-labelled finite-sate machines to structure more sophisticated behaviours on top of simpler behaviours [5].

3 State-based diagrams

The UML is predominantly graphical, and in experimental evaluations of different presentations of statecharts it has been found that, for obtaining a high-level understanding of the system, graphical notations are preferred [51]. Subjects agree that hierarchical models are easier to read than flat models and that hierarchies are absolutely necessary for modelling complex system [51]. However,

[1] Figure 2 [45, Figure 2.7] appears in Wikipedia's page on UML state machines and is distributed as commons material; we also used it on our experiments.

subjects make the most errors when working with hierarchical models [51] and errors they would not make if the flat model was used [51]. UML statecharts are now heavily used for embedded systems, even as executable models, synthesising VHDL [49]. Nevertheless, UML statecharts remain the subject of strong criticism [44, 4, 24, 40]. On one hand, the criticism is accurate regarding the ambiguous semantics, but on the other hand, UML is best used at the conceptual level. Therefore, we explored the issue of understandability, when the apparent syntactic sugar[2] is preserving meaning but somehow those producing the statecharts or those reading it fail to understand such shorthand notation.

Despite issues of visual syntax, visual and textual standard notations enable communicating software designs to stake holders [34]. In particular, the UML can be cost-effective [14], especially when using UML with a degree of formality that is realistic for being a reflection of the executable code. Monitoring 20 senior developers (ten with UML experience) on five realistic maintenance tasks that required between one to two weeks necessary conditions for UML to be effective were identified [14]. Despite the fact that the subjects were experienced developers, in order to level their background, a one day UML refresher was delivered [14]. However, that research only considered class diagrams and sequence diagrams. The conclusion matches earlier observations [51] and shows that expertise with UML and usability of associated tools are strong influencing factors in the cost-effectiveness of using UML [14].

Class diagrams (structural descriptions), sequence diagrams, and stereotypes have been the focus of UML understandability [35]. The understandability of UML's use-case templates [35] was studied because of their relevance as the main communication vehicle between all stake holders, including developers. Once again, the understandability is linked to simplicity; UML artefacts must be intuitive to understand to be successful [35]. UML diagram understandability is related to cognitive load; and thus, inexperienced users struggle with diagrams that require heavy, intrinsic cognitive load correlating with diagram details [35]. *"The use case model is understandable if it allows users to recognise problem domain information and extend their understanding in problem solving."* [35]. This definition of understandable has been used with the strong recommendation [35] that for evaluating understandability, besides question accuracy, experimenters shall evaluate the time required to complete tasks: "understanding *is a cognitive process, [and] it is difficult to directly observe it, and tests to measure participants' performance were conducted to assess the level of understanding cognitively developed by each participant."* [35].

To the best of our knowledge, previous work on the understandability of UML's statecharts is narrow [7, 8, 16, 51]. The first study focused on the form of expressing transition [51]. This study was followed by theoretical and experimental studies of the features that raise interpretation difficulty [23]. Later, the focus was that composite states add comprehensibility when users have prior familiarity with these features [8]. The counterintuitive outcome was that the experi-

[2] Again, we use the model-to-model transformation of Figure 1 to emphasise that `entry` and `exit` actions are a syntacticly convenient notation.

ments could not establish a direct link between composite state complexity and UML statechart understandability. The seemingly intuitive hypothesis would be that the use of composite states provides simplification, and thus enhances understandability; but this hypothesis is not true for inexperienced users [8].

We hypothesise that developers find UML statecharts hard to understand because their nesting usually implies uncontrolled concurrency and thus hides unexpected complexity that is intrinsically hard to reason about. That is, nesting of states, although simply described (see earlier Wikipedia quote), implies complex rules to resolve the sequence of execution. As such, we believe that logic-labelled finite-state machines (LLFSMs) are more understandable. The base of our proposal is that LLFSMs execute under a deterministic schedule. Previous research on formal verification and model checking with LLFSMs [18] demonstrates that LLFSMs avoid the exponential explosion of associated Kripke structures used by model checkers. By contrast, UML statecharts are event-driven, requiring a complex event handling process of at least five sub-steps.

Event generation and channelling: All generated events must be propagated to those states in all statecharts who are the source of a transitions waiting for the event.

Event conveyance in zero time: Events are transported to current objects and states, theoretically with no delay and without changing the event while perfectly preserving the order of events, even in a dense-time environment.

Event reception: Events are placed on queues, typically one queue per statechart [45].

Event dispatch: De-queue the event activating concurrently all responders (the listeners to the event) with *Run-Until-Completion* semantics [45].

Event consumption: Indicates that the event has been handled; in some cases, removal from the queue is just part of this step [45].

This mechanism implies the existence of call-backs associated with the corresponding events. The Hollywood principle is often viewed favourably and used in many software patterns to seemingly minimise coupling. However, as a consequence, the call-back order of execution becomes unpredictable, requiring a model checker to evaluate all paths of execution. Formal verification must consider all possible orders in which events may be queued and de-queued. This is a fundamental source of combinatorial explosion for model checkers and cognitive load for developers. To this cognitive load we further need to add the semantics of hierarchically nesting of states in UML (or combinatorial explosion because of the Cartesian product [13, Page 63]). Moreover, UML users' cognitive load rises because they must keep in mind all aspects of the event-driven *Run-Until-Completion* semantics since *"an event can trigger a transition in all active threads, in some action threads, or in none"* [13, Page 63].

By comparison, LLFSMs offer three fundamental approaches for composition.

Control/Status Message Passing: Orthogonal behaviours with different responsibilities can synchronise through a shared memory reader/writer architecture that avoids race conditions.

Using mechanisms to `suspend/resume/restart`: Enabling all sorts of rich machine hierarchies; in particular, allowing subsumption architectures.

Use of a subsumption switch: Wrap the actuators/effectors of a robotic/embedded system with a module that filters commands in accordance with priorities of behavioural layers.

4 Experimental context

We invited students from two universities (Griffith University in Queensland, Australia, and Universitat Pompeu Frabra, in Barcelona, Spain) to participate in several controlled experiments and their replication. The Australian participants had completed at least one third-year, or master's software engineering course. The Spanish participants had completed a second-year software engineering course and a elective robotics course that uses LLFSMs to create robotic behaviours (some of the students in Spain were in their fourth year while some were in their third year). The experiments consist of either

Same treatment of all subjects: Everyone solves the same problem, but we analyse metrics to confirm/reject the hypothesis. Here, two or more aspects of a participant's performance are measured. Different attributes of the participants may correlate with high/low performance. For example, all subjects are required to describe everything that is communicated in a UML statechart and we measure the accuracy of describing `entry` versus `exit` behaviour per unit of time.

Different treatment of subjects: All subjects answer the same questions but about a different, randomly assigned diagram, whereupon we perform an ANOVA (or t=test, if two classes). We randomly divide the participants into two or three groups for tasks on equivalent, but different diagrams. We assign the groups randomly for an equivalent task; but each group proceeds with diagrams with a specific feature with the control from a diagram without the feature (for example state nesting versus a diagram with no nesting).

The method we follow starts by formulating a hypothesis, for instance *"use of composite states improves understandability of UML"*. The second step defines a a measure that accounts for how quickly subjects solve a task and how accurately they solve the task. Typically we use the same measure of *"understandability efficiency"* [8] as the accuracy (the number of correct answers) divided by the time taken. Then, we define a hypothesis testing scenario; for example:

H_0**:** the use of composite state diagrams does *not* improve the understandability efficiency.

H_1**:** the use of composite state diagrams does improve the understandability efficiency.

The experiments in Australia took place during July/August 2018 while those in Spain between May/June 2019. Our experimental framework has the following aspects in common with other UML understandability studies [8, 10, 23].

Participant population: The Australian students were from two different campuses in four different degree programs (one master and three undergraduate programs). The students at Universitat Pompeu Fabra were from a single campus and a single degree program (a 4-year undergraduate program).

Motivation and persuasion: Subjects were motivated to voluntarily participate using similar incentives such as explaining that the tasks would be illustrative of the final exam [8, 10, 23].

Anonymity and voluntary participation: Participation is voluntary and responses were anonymous; thus, students were not evaluated on their individual performance.

Simplicity: The tasks in the experiment did not require a high level of industrial experience. We selected relative easy to comprehend data models [10]. We emphasised the premise that a simple data model was preferred over a more complex one (as the focus was not the application domain nor the accuracy with which the model reflects complex situations).

Concealed information and performance: We did not reveal our scoring approaches or metrics of interest. Participants were allowed plenty of time to complete the task.

Expertise: Students were in their final year, completing a course in software engineering, or they were masters students who had already completed a prior IT degree.

Long-term Preparation: Subjects received significant instruction on the main constructs of the UML. Model-Driven development was illustrated and exercised in laboratories using ARGO-UML [43] (students developed UML class diagrams and generated code in C++, Java and SQL, analysing multiple aspects of the mappings). Moreover, statecharts were used in laboratories using MDSD and executable models through the QM^{TM} tool. Students were required to review "A Crash Course in UML State Machines" [1] with overlapping content [45, Chapter 2] and distributed by Quantum®LeaPs.

Pre-task preparation: Prior to attempting the tasks, subjects were given the opportunity to review material on UML statecharts, e.g. the earlier cited Wikipedia page [46], plus two others [19, 2].

Researchers from experimental software engineering expect only minor differences between professionals and students when participants perform relatively small tasks [3, 26]. Therefore, to support the same assumption that students as subjects are appropriate [8, 10, 23] all tasks consisted of the interpretation of UML diagrams. We argue our tasks are simple because the level of nesting was capped at 2: at most one machine and one sub-machine. To minimise the impact of particular visual notations [34], we used materials from others that clearly use the same visual notation or we used a graph-layout software with the same layout parameters (in particular, we represented statechart models as conforming to a meta-model and used an ATL transformation [27] from the meta-model to `dot` [22]). The focus of our research is the semantics of the notation and its representation on diagrams [10] (and we insist, not across visual notations [34]).

For behaviour models that produce short output, subjects were asked to anticipate the output generated. For behaviours that generated continuous output,

subjects were required to identify the main traits of the behaviour, or alternatively subjects we asked whether a particular sequence of output statements occurred in that precise order. For example, with reference to Figure 2, a question asked if `internal_lamp_on()` always happened before `internal_lamp_off()`. We also emphasise that for understanding tasks (and the understandability of UML artefacts) it is common to request subjects to provide as much information as possible and to define the expected response prior to issuing the task.

5 Experimental tasks and results

5.1 Calibration

Our first experimental task was an experiment with different treatment of subjects. We reproduced verbatim Cruz *et al.*'s original Figures 5 (F5) and 6 (F6) [8] and questionnaire [8, Appendix A]. Subjects answer the same questionnaire but are randomly partitioned them between the two figures. Each figure is supposed to have an equivalent UML diagram that models the same behaviour of a phone call: F5 uses nesting states while F6 has no nested states. However, F6 is a simplified version of F5 (recall the interlingua semantics of nested states). That is, F6 draws much fewer transitions than those implied by F5. This difference demonstrates that indeed, nesting suppresses many transitions that, if drawn, would clutter the diagram (perhaps unnecessarily so). Although F5 and F6 are not semantically equivalent, the questions in the questionnaire [8, Appendix A] did not explore their semantic differences. This task could be considered particularly simple, and unfortunately [8, Figure 5] or [8, Figure 6] may be translations from Spanish to English (the figures have a spelling error).

Calibration Results The replication of the questionnaire [8, Appendix A] shows no significant evidence of a difference between nested and plain diagrams. Our experimental results are equivalent to their outcomes [8]. In Australia we collected responses from two campuses: 18 and 20 subjects respectively, each equally divided into the two groups (nested versus flat). In Spain, we collected responses from 21 subjects, with 11 nested and 10 flat. We measured *understandability efficiency*. While in the Australian results there seems to be no significant improvement (or difference) in understandability/efficiency by using nested states, the Spanish results seem to suggest even a slightly worse performance with the nested version. We offer here a new explanation derived from our earlier observations and still congruent with the original conclusions [8]. Nesting states incur in *construct redundancy* [34]. They also imply identifying a transition that is not directly leaving the source state; thus, nested states are an advanced concept. Visual notations are formal notations [34] and "uniquely human-oriented representation". *"One should not underestimate the difficulty of reading a formal specification written in mathematical notation"* [21]. In the most Piagetian constructivist style, nested statecharts require as pre-requisite knowledge flat statecharts in a concept-map. One needs to understand the notion of state before one can capture *sub-states* (let alone the notion of *chain state* [42]).

The discovery of *sub-states* revises the conceptual framework of statechart. It can only be operated efficiently if one masters the potential combinations implied by nested states. Thus, users reach command of nested statecharts when experience and regular usage assimilates the implicit semantics that the interlingua semantics implies.

5.2 Simple, Nested Model

Our second experimental task is an experiment with same treatment of all subjects. Participants must predict the output of a simple model, in particular, to comprehensively describe the information provided by the UML statechart in Figure 2. The anticipated answer was to obtain paragraphs equivalent to those in Figure 3. Also a questionnaire (refer to Figure 4) that has 8 questions testing subjects on whether they could correctly identify behaviour (sequencing) on `exit` actions, `entry` action, `entry` and `exit` actions, Run-Until-Completion, State Nesting, ordering of events, ordering of nesting (priority on exiting a hierarchy of nested states), and re-entering a hierarchy of nested states.

(2 points: statechart and composite states/sub-states) This diagram models the behaviour of some device that has, fundamentally, two states: `heating` and `door_open`. The `heating` state has sub-states `toasting` and `baking`. Because of the solid-dot pseudo-states, this device starts in the `heating` state, and in the `toasting` sub-state. Separate states are exclusive, so the system is either in the `heating` state or the `door_open` state. Similarly, the system is either `toasting` or `baking`. However, sub-states occur within their parent state; for example `baking` happens always while `heating`.

(1 point: transitions labelled by events) An event `CLOSE_DOOR` will transition the system from the state `door_open` to `heating`.

(1 point: when On-Entry and On-Exit happens) An event `DOOR_OPEN` will cause the system to move from `heating` to `door_open` no matter what sub-state in `heating`. When we leave `heating` in this case, the action `heater_off` will be performed as an *exit* activity of the state `heating` followed by the action `internal_lamp_on` which is the *entry* activity of the state `door_open`. Here we see two actions where one happens before the other.

(1 point: nesting is described) The transitions between `heating` and `door_open` are both external transitions, but the transition of the events `DO_BAKING` and `DO_TOASTING` are internal transitions. So when `DO_BAKING`, no matter the sub-state in `heating`, we will come to the sub-state `baking`, but we will not execute the *exit* of `heating`, we will execute the *exit* of `toasting` (if we were in `toasting`, that is the action `disarm_time_event` is performed but `heater_off` is not. However, the action `set_temperature` of `baking` is performed after as the *entry* to `baking`. Every time the system goes out of `toasting` the `disarm_time_event` is executed. Similarly, departing form `baking` always executed the action `set_temperature`.

Fig. 3: Grading scale to assess the translation to English when interpreting the model of Figure 2 (based on [16, Figure 3]).

Simple, Nested Model Results Our first remarkable result is the score difference observed for subjects answering questions regarding `exit` actions vs `entry` actions on the same diagram. Our questionnaires had 8 questions: a correct answer provided one point, an incorrect resulted in a negative point. For each

subject, we subtracted their score for the `exit` answer from the score for the `entry` answers. The null hypothesis was that the mean of these differences is 0. For our first campus experiment, with $N = 51$ respondents, the mean of the difference scores was 3.00 with a standard deviation of 3.85. The standard error of the mean was 0.54. A t-test with 50 degrees of freedom rejects the null hypothesis (p-value less than 0.00001). The replication at the second campus had only $N = 26$ respondents; nevertheless, the mean of the score differences was 2.11, with a standard deviation of 3.97. This results in an estimate of the standard error for the mean of 0.41 and the t-test with 25 degrees of freedom also rejects the null hypothesis (p-value less than 0.00001). In Spain we only had $N = 10$ participants; nevertheless, the mean of the difference scores was 2.4 with a standard deviation of 3.33 (the standard error of the mean was 0.76). The corresponding t-test has 9 degrees of freedom and also rejects the null hypothesis at 95% significance level with p-value less than 0.049. Thus, our experimentation reveals that subjects have different capacity to answer symmetrical questions regarding `entry` sections of statecharts as opposed to `exit` sections. The mean accuracy is higher for the `entry` questions than the `exit` questions.

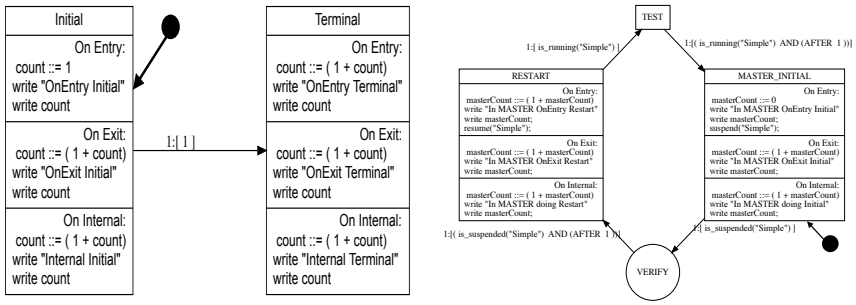
The diagram models the behaviour of a toaster oven. Assume that no events have been issued prior to each of the questions below, and that the two behaviours were launched concurrently in the order `Outer` followed by `Inner`. Answer only in terms of the actions: `heater_on()`, `heater_off()`, `arm_time_event(me->toast_color)`, `disarm_time_event()`, `set_temperature(me->Temperature)`, `set_temperature(0)`, `internal_lamp_on()`, and `internal_lamp_off()`.

1. If the event to `DO_BAKING` is received, what is/are a/the sequence of actions produced by all behaviours involved?
2. If the event to `DO_BAKING` is received; and later, after a few seconds, the event to `DOOR_OPEN` happens, what is/are a/the sequence of actions produced by all behaviours involved?
3. If the event to `DO_BAKING` is received, and while the action `disarm_time_event()` is being performed, the `DOOR_OPEN` happens, what is/are a/the sequence of actions produced by all behaviours involved?
4. If the event to `DO_BAKING` is received, and while the action `set_temperature(me->Temperature)` is running the `DOOR_OPEN` happens, what is/are a/the sequence of actions produced by all behaviours involved?
5. If the event to `DO_BAKING` is received, and after a few seconds the event to `DOOR_OPEN` happens, and while the action `set_temperature(0)` is being performed, the `DOOR_CLOSE` happens, what is/are a/the sequence of actions produced by all behaviours involved?
6. If the event to `DO_BAKING` is received, and after a few seconds the event to `DOOR_OPEN` happens, and while the action `heater_off()` is executing, the `DOOR_CLOSE` happens, what is/are a/the sequence of actions produced by all behaviours involved?
7. Write down the minimum sequence of events and conditions needed, to go from the state **toasting** to the state **baking**, and back to **toasting**, but this going back is not caused by an event to `DO_TOASTING`:
8. Write down the minimum sequence of events and conditions needed, to go from the state **toasting** to the state **baking**, and back to **toasting**.

Fig. 4: Questionnaire (in the style of earlier questionnaire [8, Appendix A]) to evaluate understandability of Figure 2 and equivalent diagrams with LLFSMs (based on [16, Figure 6]).

5.3 Non-nested LLFSM

Our third experimental task is an experiment with same treatment of all subjects. Subjects were required to predict the output of the logic-labelled finite-state machine in Figure 5a. We used the downloadable version of the `clfsm` scheduler [17] for logic-label finite-state machines and the `MiEdit` editor [15]. We used current versions of ROS under Ubuntu for the experiments (the then-current ROS-Kinetic under Ubuntu 16.04 LTS in Australia, and the updated Ubuntu 18.04 LTS and and ROS-Melodic in Spain). Subjects were provided practice in executing LLFSMs with the `clfsm` scheduler. Communication between LLFSMs was using the mechanisms of the ROS' middleware.



(a) A simple logic-labelled finite-state machine.

(b) A master logic-labelled finite-state machine.

Fig. 5: An arrangement of LLFSMs where the master suspends and resumes the simple LLFSM.

This apparently simple model has implications for understanding the notion of guards, when in a ringlet is a transition evaluated, and whether the `exit` is executed in a terminal state.

Non-nested LLFSM Results The notion of logic-labelled finite-state machines (LLFSMs) could be seen as UML models with no events and only guards: LLFSMs with no events are also called *procedural state machines* [13], in that case, the model is not at the mercy of the arrival of events: *"because [the automaton] can access the input symbols at any time, it can visit states as fast as we wish"* [13, Page 15].

The discussion of the notion of *guard* is typically linked with the illustration that UML statecharts are extended state machines [45, Chapter 2]. Since there are no events in LLFSMs, their precise semantics specifies exactly when the Boolean condition is evaluated (a snapshot of all external variables is taken before commencing of a ringlet, and all guards of all transitions are evaluated in this context). But this issue is somewhat ambiguous for the UML, the expressions are meant to be evaluated upon the arrival of the event. However, events are

queued in executable models and guards are evaluated during the dispatch of the event [45, Chapter 2] (recall the sub-steps to handle an event in Section 3).

Therefore, understandability of LLFSMs (although completely sequential), seems also to require a certain level of maturity and familiarity with UML (as we mentioned in earlier sections, most experimental evaluations of artefacts and cost-effectiveness of the UML suggest expertise and significant familiarity are required). Our results are consistent with this. We evaluated the understandability/efficiency of the subjects as the accuracy of questions about the LLFSM terminating (or running in a continuous loop), whether the execution leaves the state named **INITIAL** without executing the `do` (Internal) section, and whether the `exit` of the **TERMINAL** state is executed because no transition fires. Therefore, a fourth element is that, when in state **TERMINAL** the `do` does run. We had 21 respondents on our first campus, 10 graduate students and 11 undergraduate students. The accuracy divided by the time taken is used as understandability/efficiency and the values satisfy a normal distribution assumption with a Q-Q plot (for each group). The graduate students' mean understandability/efficiency is superior to that of the undergraduate students (statistically significant at a $\gamma = 95\%$ confidence level). Upon replication on the other campus, we had 6 undergraduate volunteers and 12 graduate volunteers. Despite the lower numbers, we also saw a significant result (at $\gamma = 95\%$), showing a superior understandability/efficiency for graduate students over the mean for undergraduate students. In Spain we only had 4 respondents, and all were third year students. Despite their earlier practice with LLFSMs they all made the same mistake of including the `do` action in the **INITIAL** although the transition fires immediately and they all include the `exit` action of the **TERMINAL** state despite no transition fires. All took more than 5 minutes to complete the task. We believe this result confirms that even the simplest UML artefacts hide very delicate issues.

5.4 Nested LLFSMs

Our the fourth task also used an executable arrangement of LLFSMs. We tested understanding of the the `clfsm` scheduler again under the same ROS middleware. In preparation for this task, the `clfsm` scheduler capabilities to `suspend`/`resume`/`restart` one LLFSM from another LLFSM had been practiced in laboratories in the students' courses (in both Australia and Spain). Although the executable model produces continuous output, the task consisted of formulating a qualitative prediction of the execution of the arrangement in Figure 5. This is the concurrent execution of the LLFSMs in Figures 5a and 5b.

Nested LLFSM Results We recorded the participants' accuracy relative to the time used to measure the understandability efficiency. The accuracy was regarding the correct prediction of the behaviour of the concurrent execution of two LLFSMs. The precise execution varies slightly when `clfsm` is invoked with the LLFSMs in Figures 5a and 5b in different order. This swapping of the arrangement order slightly modifies the output. Moreover, in this case, the execution

continues endlessly. Our results indicate a similar pattern as previously. The first campus had 14 undergraduate and 10 graduate students, the second campus had 10 undergraduate and 11 graduate respondents. Performance was significantly superior for graduate students at $\gamma = 95\%$. The undergraduate students seem to follow each LLFSM separately. But these subjects could not master the notion of ringlet (and of round-robin schedule of the concurrent execution of the two LLFSMs) with the same understandability/efficiency of the graduate students. In Spain we only had 6 volunteers from fourth year. Their performance was superior to the undergraduate students from Australia at a significance level of $\gamma = 95\%$, but we cannot place them above or below the Australian graduate students. This is consistent with the level of expertise. The Australian undergraduate students were in 3-year programs, while our more highly experienced participants were fourth year (Spain) or Master's students (Australia).

5.5 Subsumption and Delegation Results

This task requires significant preparation. In corresponding laboratories participants had been working with examples of message passing using the ROS publisher/subscriber software pattern (`rostopic`s) and the client/server software pattern (`rosservice`s). They had been shown that the sequential execution of LLFSMs implies an LLFSM cannot take the role of a ROS-subscriber (LLFSMs cannot not use callbacks). The examples illustrated that the *wrapper* software pattern is applicable here. A wrapper ROS-Node that plays both the role of a subscriber and a service is place between the publisher of a signal and the LLFSMs interested in the topic. The wrapping of signals (events) in a `rostopic` into a `rosservice` enables the LLFSM to act as a ROS-client and query the status. One of the examples practised ahead of time by all our students is the third example of the the downloadable ROS LLFSMs examples [15]. Here, the elementary turtle icon of ROS is driven to walk about its environment staying away from the boundary.

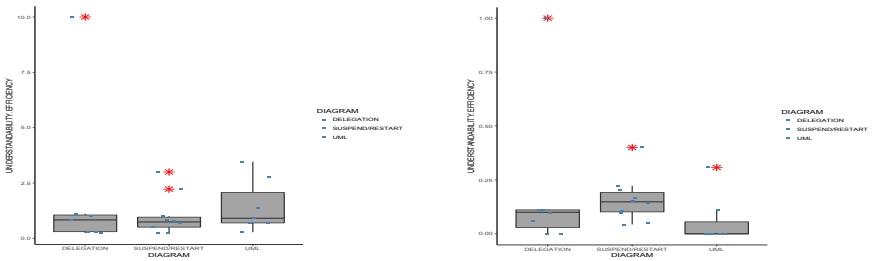
The actual task consisted of implementing the model presented in Figure 2 using LLFSMs. We provided the executable code (as a ROS-package) for a service wrappers for the signals `DO_BAKE` and `DO_TOASTING`; and the signals to `DOOR_OPEN` and `DOOR_CLOSE`. The instructions of the tasks requested to emulate the nesting hierarchy of Figure 2. The `Inner` behaviour responds to `DO_BAKE` and `DO_TOASTING` and as a result of that switches from the state **toasting** to **baking**. The second behaviour corresponds to the `Outer` behaviour that responds to `DOOR_OPEN` and `DOOR_CLOSE`. Subsequently, the behaviours are integrated. Subjects were required to commit to one of the two strategies by which LLFSMs represent state nesting: that is, subjects were asked to chose between `suspend`/`resume`/`restart` or to use a delegation (forwarding) of messages.

Subsumption and Delegation Results A remarkable aspect of these tasks is that all groups of student volunteers, including the participants from Spain, selected the incorrect implementation pattern. None of the subjects obtained a correct implementation with LLFSMs of the model in Figure 2.

5.6 Randomised Diagrams (Australia)

For the sixth experiment, subjects were randomly partitioned into three groups and provided the same earlier Questionnaire (Figure 4). However, each group was provided with a different diagram. The first group was provided the diagram in Figure 2, the second group was provided a model solution to its implementation using `suspend/resume/restart` with LLFSMs, and the third group was provided a model solution using delegation/forwarding with LLFSMs.

Randomised Diagrams (Australia) Results Here we used first a 3-factor ANOVA (between subjects / one-way) analysis, as we identify the three types of diagrams. If we measure the accuracy on the 8 questions in Figure 4 divided by time, we find no evidence that the means are different. The box-plot in Figure 6a (generated with R's `ggplot` [39]) shows not much difference, except for one outlier where one subject did extremely well for the LLFSM diagram using the delegation pattern. However, if we break the questionnaire into the four middle questions (3, 4, 5, and 6, which deal with Run-Until-Completion semantics), we can see that the results are significantly better for understandability/efficiency for diagrams with suspend/resume (refer to Figure 6b). Conversely, on Ques-



(a) Box plot of the 3-ways understandability/efficiency per model type of the Figure 4 questions.

(b) Box plot of the 3-ways understandability/efficiency per model type on Q's 3 to 6 for Figure 4.

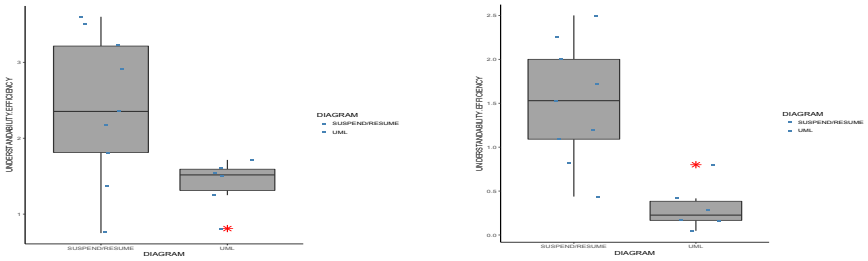
Fig. 6: Box plots for Australian participants (the units on the right plot are half to the left plot since half the questions are used).

tions 1, 2, 7, and 8, the UML diagram performs much better. Again, the 3-factor ANOVA results show no statistically significant difference. However an unpaired (two sample) t-test of the understandability/efficiency on the UML diagram versus the suspend/resume diagram does indicate the rejection of the null hypothesis at 95%. That suggests UML diagrams are understandable as long as we set up scenarios with well-spaced events, where users can follow all the consequences of one event before the arrival of another. LLFSMs seems to be the other way around. While, at a first glance, the run-until-completion semantics appears obvious and straightforward, in our experimental task denoted in Figures 2 and 4, almost all subjects had substantial trouble with Questions 4 to 8.

5.7 Randomised Diagrams (Spain)

This experiment is similar to the Randomised Diagrams experiment performed in Australia we just described (see Subsection 5.6). However, we had only $N = 15$ participants, so we randomly divided them into only two groups, those working with the UML from Figure 2 versus those working with a suspend/resume solution of LLFSMs. Again, the questions are those of Questionnaire (Figure 4). Our partition results in 6 students working with the UML diagram from Figure 2 with the other (9 participants) working with the LLFSM diagram.

Randomised Diagrams (Spain) Results The plots in Figure 7 show that the participants were (statistically significant) more proficient with the LLFSM diagram than with the UML diagram. Figure 7a is the total score for the questionnaire, while Figure 7b. In this experiment, participants performed particularly poorly with those question that deal with the *Run-Until-Completion* semantics of the UML. Recall that the *Run-Until-Completion* semantics requires that the UML users keeps in mind the queue of events while resolving the current event. Most users seem to *interrupt* the handling of the current event and perform the actions of new arriving event. Note that the y-axis in the two figures in



(a) Box plot of the 2-ways understandability/efficiency per model type of the Figure 4 questions.

(b) Box plot of the 2-ways understandability/efficiency per model type on Q's 3 to 6 for Figure 4.

Fig. 7: Box plots for Spanish participants (the units on the right plot are half to the left plot since half the questions are used).

Figure 7 is not the same scale, nevertheless, the gap between those with the UML diagram and the others widens, highlighting how challenging is to grasp the *Run-Until-Completion* semantics of the UML.

6 Analysis

6.1 Lesson Learned

LLFSMs are apparently simpler because, as we mention, they could be considered UML statecharts without events. But LLFSMs offer a precise and unambiguous semantics that provides complete detail for execution and verification.

Our results suggest that LLFSMs require significantly more maturity from participants. However, when issues of timing and order of execution become more critical, or when interpreting and understanding the effect of event showers, or the handling of events while another event is still being processed, LLFSMs are much clearer and more transparent. This is particularly supported in the Spanish replication of the experiments.

It may not seem unexpected that these experiments demonstrate that enduring experience with UML is required for high understandability/efficiency. We nevertheless found some remarkable surprises. For example, we discovered that among the population of subjects there is a strong belief that UML statecharts imply strong restrictions on the ordering of events. In particular, 32% of the first-campus group (52 respondents) indicated in one particular question that the diagram in Figure 2 implies that `DOOR_OPEN` must always be followed by `DOOR_CLOSED` (the group on the second campus had 26 respondents, but a percentage as high as 38% also expected such an ordering of events). These responses occurred despite earlier lab demonstrations to students (prior to the questionnaire) that showed an implementation of Figure 2 with QM^{TM}. In those demonstrations, we explicitly showed that all sequences of events of the form (`DOOR_CLOSED`|`DOOR_OPEN`)* were valid for the `Outer` behaviour. Our participants were a subset of the students instructed in the laboratories who experienced the execution of the implementation which displayed a behaviour that would toggle between the state **door_open** and **heating** at the right time. That is, duplication of the event `DOOR_OPEN` once in the state **door_open** is possible and has no effect. Moreover, 68% of these participants could not commit either way about whether the diagram implied something regarding the order of events. Only 10% could confirm that the UML diagram in Figure 2 is at the mercy of the sequence of events coming from the environment, and its implementation (or executable model) behaves correctly only when it makes no assumptions about a benevolent environment.

6.2 Threats to Validity

Clearly, students are not professionals in the field of practice, and although, like others [28], we justified their participation, those with experience in several projects may be a different group subjects [14]. Especially since our results suggest that expertise developed with experience is a contributing factor to understandability. Since the tasks are simple or at least not very sophisticated (for example, Figure 2 and Figure 4), it is possible that the results could be different in other settings. For example, industrial scenarios usually involve complex behaviours of many inter-dependent statecharts [38]. Also, understandability interacts with other factors (for instance, different development tools) in more complex ways than in our controlled experiment. In particular, to remove other factors, we conducted the experiments separately, allowing at least one week in between two of them to minimise the effect of fatigue. The groups were small so we could eliminate plagiarism because we could ensure no individual received any coaching, advice, or communication with others.

Nevertheless, for evaluating understandability, the students' lack of experience is potentially a catalyst for the difficulty in grasping the models [35]. If models were highly understandable, novice users would not exhibit the difficulties we observe. Moreover, using simple tasks for UML diagrams is a suitable approach when dealing with subjects that are students, as long as we keep in mind that the experience of users could have a more profound effect on more complex tasks [35, 28].

The UML refresher material may be another issue to consider for external validity. We found the results from Spanish students somewhat surprising, regarding their inability to handle the *Run-Till-Completion* semantics of the UML, but their high performance with LLFSMs. It is possible that more practice is required with a tool such as the already mentioned QM^{TM} that implements the *Run-Till-Completion* semantics. Results may vary if participants are exposed and have more exposure to one model of semantics than the other. One issue could be that Australian students effectively had little prior exposure to the notion of state machines. Neither of five programs feeding into the pool of Australian subjects had a course on automata and formal languages, while he Spanish students do have exposure to automata theory and some had exposure to a course in compilers with some content on lexical analysis. We should point out that if we compare the performance of Spanish students against Australian students under the same diagram, the Spanish students' understandability/efficiency is superior, with statistical significance, to the Australians. We can see that in Figure 7a the mean understandability/efficiency for Spanish participants is way above 1 for all diagrams, which is where Australian participants are almost for all diagrams (refer to Figure 6a). The same is true when we focus on the *Run-Till-Completion* semantics. The understandability/efficiency for Spanish participants is above 0.25 for all diagrams (Figure 7b) which is above the understandability/efficiency for Australian participants in the corresponding figure (Figure 6b). Similarly, results may vary if subjects had exposure to state diagrams from other areas.

We face the same challenges as all other studies with respect to construct validity (the suitability of the instrument to measure understandability). *Understanding* is a cognitive process, we can only measure performance elements that we believe reflect the level of understanding. We are using the common hypothesis that failure to achieve a task (such as translation into another language or into the output behaviour sequence) is linked to a lack of understanding. However, UML notations may be simply hard to learn (which may be associated with understandability). We also may not have been able to record the understanding failures accurately.

Challenges could be derived from violations to the statistical assumptions that enable a particular analysis (that is, statistical validly), such as low statistical power or low effect size. When testing between two groups, we used the t-test. Where we employ ANOVA, we assume homogeneity of variance as there does not seem to be any other factor that would invalidate this assumption. We used a Q-Q-plot to validate the assumption of normally distributed random

variables. When partitioning, each value was sampled independently from any other variable to ascertain between-subject factors. However, we acknowledge that our sample sizes were smaller than those in other, similar studies. Nevertheless, we discussed results only where we could report statistical significance. Since participation was voluntary, the class sizes (where lectures and laboratories were delivered to participants) were larger than the samples reported. This self-selection of the subjects implies a potential bias. It is possible that diligent students seek more practice.

7 Asymmetric Semantics

Our results in the earlier section as well as others [9] suggest that hierarchical nesting of UML statecharts does not scale well. This suggestion that nesting level inversely correlates with understandability could be justified because when taken as executable models, hierarchies of UML statecharts would have high McCabe Cyclomatic Complexity (the number of linearly independent paths through the execution is high because nested statecharts are an abbreviation for one large statechart with as many states as the Cartesian product of those in the hierarchy). We recall that McCabe Cyclomatic Complexity is the number of linearly independent execution paths through the model.

However, the issue complicates itself when discussing `entry` and `exit` actions. The following quote reflects a brief explanation for the roles of sections in UML statecharts. *"Regardless of how a state is entered or exited, all its entry and exit actions will be executed"* [45, Page 76]). It reflects symmetric semantics for `entry` actions and `exit` actions. However, we will build on our earlier arguments to illustrate further inherent asymmetry that is inherently present in LLFSMs when we consider the suspension of a member of the arrangement. The suspension is a meta-action (from the perspective of the machine being suspended) that is performed by the scheduler (when triggered, for example, by a higher-level machine in the subsumption architecture). In particular, it is quite inappropriate that a machine that has been suspended (and thus no longer operating) were to execute any actions. Such suspended machines should not execute any code, not even their exit actions (of if it does, as per philosophy of the subsumption architecture, these actions are blocked from causing any effect). Figure 2 illustrates that humans interpreting the model would expect the oven would be immediately turned off when sensors report the door is open. The results of our study show a significant difference in the number of participants that preferred to treat the implementation in Figure 2 by the `suspend/restart` mechanism when the `DOOR_OPEN/DOOR_CLOSED` (respectively) signals are detected.

Suggesting that `exit` actions are not performed in a suspended state-machine may seem to contradict the event-driven nature of UML with its associated run-until-completion semantics — where all `exit` actions are *always* performed. Considering what happens when a machine is suspended raises the cognitive load on designers and developers when constructing or interpreting a set of statecharts. In particular, designers must consider that the suspender machine needs to account for further activity still performed by someone who is suspended. This

possibility that a suspended statechart may perform some actions violates the principle of the least surprise. Moreover, the delicacies of this issue can have severe consequences in safety-critical systems (such as the radiation magnetron of a microwave), where two opposing concerns (regular operation and immediate shutdown) suddenly have to be catered for in the same (`exit`) action. We now address the critical issue of defining the semantics of a machine that receives a `suspend` with respect to the `exit` section of its current state.

We observe that the inherent asymmetry between execution context and sphere of control [29] (subsystem vs meta-action) results in an asymmetry of the `entry` and `exit` actions. Composition of larger models of LLFSMs is achieved by including further behaviours in the pre-scheduled sequential execution of an arrangement and explicitly invoking their execution (or suspension). The semantics of an arrangement of LLFSMs is that all machines in the arrangement are executing concurrently, but only one at a time is effectively running. When the holder of the execution token runs the actions associated with its current state, it executes one ringlet in the current state, and then, the scheduler passes the token of execution is passed to the next machine. The semantics for running a ringlet is defined as follows.

1. The `entry` action is executed if (and only if) the previous state was different to the current one.
2. The predefined sequence of transitions is evaluated, if none of them is true, the `do` section is executed and the ringlet finishes.
3. Alternatively, the `exit` is executed, when a transition T evaluates to true, and the target state of T becomes the current state. This also completes the ringlet.

Thus, when arriving from another state, the `entry` section of a state is executed once and only once, without exception (this interpretation of transitions with identical source and target state caused some issues in SCXML [50]).

Note that, at the time a machine is suspended the suspender (another higher-level, controller machine) holds the token of execution. That is, the `suspend` happens outside the sphere of control [29] of the machine being suspended. Therefore, for robotic and embedded systems, which often implement the subsumption architecture [5] and `suspend`/`resume`/`restart` signal, a suspended machine should not run any actions after the suspension signal. For example, consider a behaviour design where the `entry` section a state S prepares a motion-related action for a robot, later to be actually set in motion by a control signal in the `exit` section and by a transition labelled with some condition (such as, an object become visible). Suppose that a super-imposing behaviour ensures some safety constraints on the motion, for instance, that the posture is safe for the motion. If the robot were to change to an unsafe posture (a fall), the super-machine would issue a suspend but the controlled behaviour would execute the `exit` and drive the robot to perform a motion in an unsafe posture. In fact, if designers follow the semantics suggested here, the trigger of a motion shall not be when leaving a state, but on arrival to a state (the design is vastly improved by breaking

the preparation and launching of the motion into two states, and the launching being in the `entry`).

As another illustration, we review the higher-level machine switching behaviour between **toasting** and **door_open** in Figure 2. The submachine that switches between **toasting** or **baking** is the inner behaviour, and it would only receive the token of execution after it has been suspended, i.e. after the higher-level machine has performed the `suspend`. By the time the inner machine receives back the token of execution, it already is in its suspended state. Thus, it would be quite surprising if it were to resume its prior state to catch up and run some associated `exit` action.

To consider `entry` and `exit` actions as symmetric actions has some potential mnemonic elegance. However, the argument here is that this symmetry only applies within the sphere of control [29] of a single machine. The earlier examples serve to stress our argument. Actions in the `exit` are executed when one transition fires (the Boolean expression labelling it evaluates to true). Running the actions of such `exit` section while on suspension results in the actions of `exit` being executed in a completely unpredictable context. This context is unpredictable because is subject to all the actions of all other behaviours in the composition; that is in the arrangement. Such a context can clearly be radically different and inconsistent with the conditions that are stated in the Boolean expressions that label the transitions leading away from the state in question. This is terribly unsafe (and thus, our argument why the example above should be redesign, and the semantics favour the redesign).

Our discussion explains one further point of asymmetry now focused on the `entry` section. Namely, the `entry` section is executed when the operation of a machine is resumed. This is completely analogous, consistent and correspond to having no exception to when a machine is first started and the `entry` of its initial state is executed. Thus, the corresponding `entry` is executed when a LLFSM that was previously suspended is resumed or restarted. The rationale is simple, the corresponding machine is in control and able to perform its specified actions.

The earlier suggested re-design of example robot preparing and signaling a motion corresponds to classical software engineering notions. Executing the actions of an `exit` section precisely, and only when one of the transitions has its guard evaluated to true, acts as a precondition (in the sense of programming by contract [33]). Thus, this semantics enforce a stronger, The enforcing of this semantics enables designers and developers to practice first-principles of software development with statechart models. Among these principles are code re-use, separation of concerns, decoupling and locality of effects since in a layered architecture, lower layers shall be completely unaware of higher layers.

The semantics of LLFSMs does not prevent the implementation with LLFSMs of a design using UML's composability with nested states. For example, in Figure 2, the `DOOR_OPEN` signal acts as a trigger to several `exit` actions (in the outer and in the inner machines). Because this is a UML design, the `exit` actions are aimed at reversing the corresponding `entry` actions. For instance,

suspending the inner machine and not executing its `exit` would leave the toaster on. The problem is that the UML design hides that the super-machine requires cooperation from its sub-machine. This assistance ought to be made explicit through notification (to the sub-machine) of a condition. This is clearly a delegation pattern (and forwarding of a corresponding signal). When the system is in the state **baking**, it is also in the state **toasting** and thus it shall listen to `DOOR_OPEN` as well. Moreover, the order in which all the nested states execute their corresponding `exit` actions becomes explicit by using delegation. Note, however, that the original description of nested-state semantics by Harel prioritised first the super-state over the sub-state, but UML has revised this interpretation and no an inverse prioritisation is used. This, once more, emphasises the significance of clear, accessible semantics to the designer. Thus, our the discussion here emphasises the importance of characterising the scenarios where the subsumption architectural pattern of independent components is applicable, versus those situations where other patterns, such as delegation and communication, are applicable. Making these explicit to software engineers may alleviate the confusion that exists, as our experiments have revealed.

8 Conclusions

Our experiments have shown that the simplistic model of symmetry between `entry` and `exit` actions does not even hold true when tested against simple, nested models. Perhaps unsurprisingly, more experienced participants (masters students) showed a superior capability in their levels of understanding compared to less experienced participants (undergraduate students).

The wide-spread use of state diagrams for model behaviour for the ever growing number of embedded devices (just consider the Internet of Things) makes it imperative that executable models, such as LLFSMs, delineate the precise semantics derived from the `entry` and `exit` asymmetry. We have argued for a semantics where `exit` is executed upon leaving the state in the sphere of control of the current machine. The experiment show that the interaction of nesting states with UML's run-till-completion semantics is particularly hard to grasp.

Overall, our study shows that, while complex, nested models are hard to interpret for humans, a precise semantics (as is necessary for verifiable, executable models) needs to be intuitive for human understandability. Importantly, compared to the literature, where seemingly simple, symmetric semantics have led to often counter-intuitive or difficult-to-comprehend system behaviours, we have demonstrated an asymmetry in participants' understanding of `entry` vs `exit` behaviour that supports our hypothesis of an intrinsically asymmetric execution semantics leading to a more intuitive system behaviour.

References

1. A crash course in UML state machines. https://www.state-machine.com/doc/AN_Crash_Course_in_UML_State_Machines.pdf (2015), [Online; accessed 20-06-2019]

2. State machine diagram tutorial. https://www.lucidchart.com/pages/uml-state-machine-diagram (2018), [Online; accessed 20-06-2019]
3. Basili, V.R., Shull, F., Lanubile, F.: Building knowledge through families of experiments. IEEE Trans, on Software Engineering **25**(4), 456–473 (1999)
4. Börger, E., Cavarra, A., Riccobene, E.: Modeling the dynamics of UML state machines. In: Abstract State Machines-Theory and Applications. pp. 223–241. Springer (2000)
5. Brooks, R.: A robust layered control system for a mobile robot. IEEE J. Robotics and Automation **2**(1), 14–23 (1986)
6. Colburn, T., Shute, G.: Abstraction in computer science. Minds and Machines **17**(2), 169–184 (2007)
7. Cruz-Lemus, J.A., Genero, M., Manso, M.E., Morasca, S., Piattini, M.: Assessing the understandability of UML statechart diagrams with composite states—a family of empirical studies. Empirical Software Engineering **14**(6), 685–719 (2009)
8. Cruz-Lemus, J.A., Genero, M., Manso, M.E., Piattini, M.: Evaluating the effect of composite states on the understandability of UML statechart diagrams. In: Model Driven Engineering Languages and Systems. pp. 113–125. Springer (2005)
9. Cruz-Lemus, J.A., Maes, A., Genero, M., Poels, G., Piattini, M.: The impact of structural complexity on the understandability of UML statechart diagrams. Information Sciences **180**(11), 2209 – 2220 (2010)
10. De Lucia, A., Gravino, C., Oliveto, R., Tortora, G.: An experimental comparison of ER and UML class diagrams for data modelling. Empirical Software Engineering **15**(5), 455–492 (2010)
11. Dijkstra, E.W.: Foreword. In: Hinchey, M.G., N., D.C. (eds.) Teaching and Learning Formal Methods, pp. vii–viii. Elsevier (1996)
12. Douglass, B.P.: Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley, Boston, MA, USA (1999)
13. Drusinsky, D.: Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking. Newnes (2006)
14. Dzidek, W.J., Arisholm, E., Briand, L.C.: A realistic empirical evaluation of the costs and benefits of UML in software maintenance. IEEE Trans. Softw. Eng. **34**(3), 407–432 (May 2008)
15. Estivill-Castro, V., Hexel, R.: Downloads. http://mipal.net.au/downloads.php (2016), [Online; accessed 20-06-2019]
16. Estivill-Castro, V., Hexel, R.: Resolving the asymmetry of on-exit versus on-entry in executable models of behaviour. In: 7th Int. Conf. on Model-Driven Engineering and Software Development, MODELSWARD. pp. 51–63 (2019)
17. Estivill-Castro, V., Hexel, R., Lusty, C.: High performance relaying of C++11 objects across processes and logic-labeled finite-state machines. In: Simulation, Modeling, and Programming for Autonomous Robots - 4th Int. Conf., SIMPAR. Lecture Notes in Computer Science, vol. 8810, pp. 182–194. Springer (2014)
18. Estivill-Castro, V., Hexel, R., Rosenblueth, D.A.: Efficient model checking and FMEA analysis with deterministic scheduling of transition-labeled finite-state machines. In: 3rd World Congress on Software Engineering (WCSE 2012). pp. 65–72. IEEE Comp. Soc. (CPS), Wuhan, China (2012)
19. Fakhroutdinov, K.: State Machine Diagrams. https://www.uml-diagrams.org/state-machine-diagrams.html (2009), [Online; accessed 20-06-2019]
20. Finney, K.: Mathematical notation in formal specification: too difficult for the masses? IEEE Trans. on Software Engineering **22**(2), 158–159 (1996)

21. Finney, K., Fedorec, A.M.: An empirical study of specification readability. In: Hinchey, M.G., N., D.C. (eds.) Teaching and Learning Formal Methods, pp. 117–129. Elsevier (1996)
22. Gansner, E.R., Koutsofios, E., North, S.: Drawing graphs with *dot* (2015), https://www.graphviz.org/pdf/dotguide.pdf, [Online; accessed 20-06-2019]
23. Genero, M., Miranda, D., Piattini, M.: Defining metrics for UML statechart diagrams in a methodological way. In: Conceptual Modeling for Novel Application Domains. pp. 118–128. Springer (2003)
24. Glinz, M.: Problems and deficiencies of UML as a requirements specification language. In: 10th Int. Workshop on Software Specification and Design. p. 11. IEEE Comp. Soc. (2000)
25. Harel, D., Politi, M.: Modeling Reactive Systems with Statecharts: The STATEMATE Approach. McGraw-Hill, New York, NY (1998)
26. Höst, M., Regnell, B., Wohlin, C.: Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. Empirical Software Engineering **5**(3), 201–214 (2000)
27. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1), 31 – 39 (2008). https://doi.org/https://doi.org/10.1016/j.scico.2007.08.002, special Issue on Second issue of experimental software and toolkits (EST)
28. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Trans. on Software Engineering **28**(8), 721–734 (2002)
29. Kopetz, H.: Real-Time Systems - Design Principles for Distributed Embedded Applications. Real-Time Systems Series, Springer, Berlin, second edn. (2011)
30. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (Nov 1994)
31. Mellor, S.J.: UML point/counterpoint: Modeling complex behavior simply. Embedded Systems Programming (2000)
32. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (Mar 2006)
33. Mitchell, R., McKim, J., Meyer, B.: Design by Contract, by Example. Addison-Wesley, Reading, MA (2002)
34. Moody, D.: The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. on Software Engineering **35**(6), 756–779 (2009). https://doi.org/10.1109/TSE.2009.67
35. Mustafa, B.A.: An experimental comparison of use case models understanding by novice and high knowledge users. In: New Trends in Software Methodologies, Tools and Techniques - 9th SoMeT_10. Frontiers in Artificial Intelligence and Applications, vol. 217, pp. 182–199. IOS Press (2010)
36. Object Management Group: Action language for foundational UML (alf) — concrete syntax for a uml action language. Version 1.1. Tech. Rep. formal/2017-07-04, An OMG Action Language for Foundational UML Publication (2017), normative reference: http://www.omg.org/spec/ALF/1.1
37. Object Management Group: Omg unified modeling language bersion 2.5.1. Tech. Rep. formal/2017-12-05, OMG Object Management Group Publication (2017), normative reference: http://www.omg.org/spec/UML
38. Petre, M.: UML in practice. In: Int. Conf. on Software Engineering. pp. 722–731. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013)

39. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2016), https://www.R-project.org/
40. Reggio, G., Astesiano, E., Choppy, C., Hussmann, H.: Analysing UML active classes and associated state machines-a lightweight formal approach. In: Fundamental Approaches to Software Engineering, pp. 127–146. Springer (2000)
41. Reggio, G., Leotta, M., Ricca, F., Clerissi, D.: What are the used UML diagrams? a preliminary survey. In: 3rd Int. Workshop on Experiences and Empirical Studies in Software Modeling (EESSMod 2013 co-located with MODELS 2013). vol. 1078, pp. 3–12. CEUR (2013)
42. Richardson, M.: Guideline: Statechart diagram. http://www.michael-richardson.com/processes/rup_for_sqa/core.base_rup/guidances/guidelines/statechart_diagram_640B5D0B.html (2015), [Online; accessed 20-06-2019]
43. Robbins, J.e.: Cognitive Support Features for Software Development Tools. Ph.D. thesis, Department of Information and Computer Science, University of California, Irvine (1999), advisor: Prof. D. F. Redmiles
44. Rumpe, R.: Executable modeling with UML – a vision or a nightmare? –. In: Issues and Trends of Information Technology Management in Contemporary Associations Volume 1. pp. 697–701. Idea Group Publishing (2002)
45. Samek, M.: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems. Newnes, Newton, MA, USA (2008)
46. Samek, M.: UML state machine. https://en.wikipedia.org/wiki/UML_state_machine (2009), [Online; accessed 20-06-2019]
47. Seshia, S.A., Sharygina, N., Tripakis, S.: Modeling for verification. In: M., C.E., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking., pp. 75–105. Springer (2018)
48. Shaw, M.: Toward higher-level abstractions for software systems. Data & Knowledge Engineering **5**(2), 119 – 128 (1990)
49. Wood, S.K., Akehurst, D.H., Uzenkov, O., Howells, W.G.J., McDonald-Maier, K.D.: A model-driven development approach to mapping UML state diagrams to synthesizable VHDL. IEEE Trans. on Computers **57**(10), 1357–1371 (2008)
50. World Wide Web Consortium: State chart XML (SCXML): State machine notation for control abstraction (Sempember 1st 2005)
51. Zimmerman, M.K., Lundqvist, K., Leveson, N.: Investigating the readability of state-based formal requirements specification languages. In: 24th Int. Conf. on Software Engineering. ICSE. pp. 33–43 (2002)