Statecharts for Unified Model-Based Design – As simple as possible, as rich as needed

Jean-Louis Dufour

To cite this version:

Jean-Louis Dufour. Statecharts for Unified Model-Based Design – As simple as possible, as rich as needed. ERTS 2018, Jan 2018, Toulouse, France. hal-02156151
Statecharts for Unified Model-Based Design – As simple as possible, as rich as needed.


Summary: software projects using Simulink or Scade use in fact a subset of Simulink or Scade. The ‘alignment’ of these two subsets gives rise to a new concept, the ‘Unified MBD’, whose data-flow part keeps the expressiveness of both languages and has been introduced in a former paper. Here we present the control-flow part, which is much more restrictive, because the automata paradigms of Scade and Simulink differ fundamentally. But in fact, at least in our context, this loss of expressiveness is not a weakness, because our current design modelling practice is consistent with these restrictions, and because it opens the road for simple but faithful specification modelling practices.

Keywords: Model-Based Software Engineering, Languages and Compilers, Scade, Simulink.

1. Statecharts in Unified Model-Based Design

Model-Based Design (MBD) is today a major paradigm in the engineering of critical embedded software. Here we will focus on implementation models, from which code is automatically generated. The main actors are Simulink (its discrete part) and Scade, and what we call the ‘Unified MBD’ is simply their intersection. [DufourCorrubleTavernier 16] presented motivations and the data-flow part.

The control-flow part (we will say also ‘automata’, ‘state-machines’, ‘Statecharts’; ‘moding’) was just mentioned, with two key points:

1. Only a few percent of the code come from state-machines (because our applications contain few moding and are mainly algorithmic; on cockpit or in railway, it could be very different).
2. almost none of the state-machine constructions are used: their semantics is often subtle and slightly different between Scade and Simulink, so this is directly conflicting with the first goal of Model-Based Design which is communication between engineers.

The first point has now to be precised:

- the moding behavior is perhaps a minor part of the code (again, of our applications), but it is a major contributor of hard-to-find bugs. E.g. a recurrent touchy pattern is ‘exclusive-or’ redundant functionality, from purely software (2 Kalman filters) to equipment-level (2 boards): experience shows that guaranteeing exclusive-or behavior is beyond the industrial state-of-the-Art.
- these bugs occur at the equipment-level (let say, a single software), but also at the system level (several cooperating equipments in our perimeter, or worse, interaction with our context),
- the ‘moding behavior’ is (for the time being) a fuzzy concept which is not restricted to the state-machine design pattern: sometimes we see it hard-coded in data-flow, often because of bad cultural reasons or insufficient training. This ‘hidden moding’ phenomenon will not be developed further here, but it is an example of the few cases where the specification may not only dictate the ‘what’ but also the ‘how’.

The second point is well known to practitioners (at the design level), and has been noticed also by academics at the specification level ([HeimdahlLeveson95], [Gilinz02] who has not only inspired this paper but also its title).

Our context is given by fig. 1. Above the ‘software design’ level (i.e. MBD level), which is the subject of this paper, the ‘big picture’ is the following:
• at the ‘system design’ level of description (the input of software development), the good paradigm seems to be: the software ‘reacts’ to ‘events’ (time-based or functional) and requirements deal with sequences of events and reactions (like SysML’s sequence diagrams). Ultimately, when we will have a working concept of refinement between this level and the software level, this will be the right place to prove the correctness of the ‘moding behavior’. Unfortunately, this working concept is not yet available, …

• … so we must fall back on the ‘software specification’ level of description, where we react synchronously to what we call ‘pseudo-events’ (typically, the rising-edge of a Boolean flow). Here, a requirement can be single-cycle, like a state-machine, or it can be multiple-cycle, like a temporal logic formula, and refinement concepts begin to be operational ([SaidButlerSnook09], [PitermanPnueliSa’ar06]).

Figure 1 : context diagram of unified MBD

So, our (unattained) objective is to define a subset of Stateflow (Simulink statechart’s toolbox) and a subset of Scade’s automata, such that:

- There is a bijective and intuitive correspondence between both,
- These subsets are ‘as simple as possible, as rich as needed’ (‘rich’ refers to our context),
- Each of them support formal refinement from a yet-to-define specification formalism.

2. What is a state-machine ?
This question looks silly: everybody knows what a state-machine is! In fact, when you begin to study the similarities and the differences between Scade and Simulink, you realize that the subject is subtle, and that it is the key to the elaboration of a ‘unified’ specification and design formalism. The key concepts have slowly matured for 50 years, so to answer this question, History provides a natural guide. It begins in September 1955 with two famous papers of Edward Moore and George Mealy, both motivated by the appearance of the first electronic computers which were replacing the former relay computers. These two papers are among the first to explicit the distinction asynchronous vs. synchronous logic (input-change-triggered vs. clock-triggered computation) and to favor the latter.

2.1 Moore’s “Gedanken-experiments on sequential machines” and weak transitions
Moore is a mathematician, he is not familiar with technology. His ‘machines’ are black boxes, in the spirit of Turing machines, described by high-level properties:

- <<Time … come in discrete steps, so the machine … [is] … a synchronous device.>>
- <<Each machine will have a finite number n of states … >>
- <<The state … depends only on its state at the previous time and the … input symbol.>>
He justifies them by the fact that <<Digital computers ... are usually built in this synchronous fashion>>. He adds a last property, rather ad-hoc compared to the preceding general ones:

- <<The output symbol at a given time depends only on the current state ...>>

This strong restriction characterizes what we call today ‘Moore machines’: outputs don’t depend (directly) on inputs. It implies two apparently contradictory properties, speed and delayed reactivity:

- speed: unbeatable timing performance (in extreme cases, with a clever encoding of the state, outputs can be [a part of] the state), that’s why it has always been (and will always be) a standard design pattern in hardware;
- delayed reactivity: you cannot even describe a basic differential encoder ($y = x \oplus x^{-1}$), you have to delay it by one cycle.

Moore describes an example with a transition table (left) and an output table (right):

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Previous Input</th>
<th>Present State</th>
<th>Present Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>$q_4$</td>
<td>0</td>
</tr>
<tr>
<td>$q_2$</td>
<td>1</td>
<td>$q_1$</td>
<td>0</td>
</tr>
<tr>
<td>$q_3$</td>
<td>0</td>
<td>$q_4$</td>
<td>0</td>
</tr>
<tr>
<td>$q_4$</td>
<td>1</td>
<td>$q_2$</td>
<td>1</td>
</tr>
</tbody>
</table>

The interesting part comes now: <<An alternate way of representing the description of a machine can also be used, which may be somewhat more convenient to follow. ... called a transition diagram ...>>

This ‘experiment’ is crucial: we now have the exact behavior. The preceding output table was misleading, because Moore has fallen into the standard pitfall of state-machines: he has used the expressions ‘present state’ and ‘current state’, which are too much subjective hence diversely interpreted.

- The 1st column is related to the 1st cycle (Scade terminology; in Simulink this is time 0). It says that this cycle ‘starts’ in state $q_1$ (the ‘previous’ or ‘old’ state) so it has output 0 (independently of the input and of the next state).
- The 2nd column ‘starts’ in state $q_4$, which means that $q_4$ is the state at the ‘end’ of the 1st cycle: we will say that from the point of view of the 1st cycle, $q_4$ is the ‘next’ state or the ‘new’ state, which has been computed from the input (0), in parallel with the output.

The moral is that the left column of the output table has an erroneous title: it is not the ‘present state’ but the ‘previous state’. It is illustrated in the following Scade diagram which show the 1st cycle (Moore’s transition table is the left switch, the output table is the right switch, driven by ‘prev_state’):
The computation of the outputs is really done in parallel with the computation of the next state, but it is more intuitive to remember this in the following way:

The computation of the outputs is done before the transition to the new state.

Following Scade terminology (explained later), we will say that the transitions are **weak**: they don’t brutally take the control of the machine, they let the activities of the previous state occur a last time.

### 2.2 Mealy’s “... synthesizing sequential circuits” and ‘digital-completeness’

Mealy is an engineer, who knows the new switching technology. The expression ‘flip-flop’ has not yet emerged (‘1/z’ for Simulink, ‘fby’ and ‘pre’ for Scade), and instead he speaks of ‘unit of delay’, but apart from this, the article is completely modern. He describes his first example with a ‘circuit diagram’ and the equivalent ‘truth-table’:

Mealy is aware of the paper of Moore (published in the classic book ‘Automata Studies’ in April 1956, but in circulation at least 2 years before), and he goes on with the interesting part in a very similar way:
<<It is usually not clear from … the circuit diagram … what a sequential circuit does. The truth-table is more helpful and tells the whole story if we put it in a different form, called a state diagram.>>
In fact, he has taken the same example! It is visible only in the automaton view, not in the truth-table view, because in this latter view it is hidden by the encoding of the state (exercise: what is this encoding?). The important fact is that, contrary to the restricted model of Moore, we have a full model of computation. Technically, it is not a Turing machine, but it is a ‘digital-real-life’ machine: each circuit with s bits of state and i bits of input can be represented by a Mealy machine with \(2^s\) states, each having \(2^i\) outgoing transitions.

Usually, ‘Moore-style’ means ‘outputs defined in states’ and ‘Mealy-style’ means ‘outputs defined in transitions’, with the implicit suggestion that this is the origin of the better responsiveness of Mealy-style. But in fact, machines with outputs in states but depending on inputs can also represent any circuit. The key characteristic is whether the outputs depend on the inputs or not:

In any case, outputs are computed from the previous state, ‘in parallel with’ the next state, so transitions are said ‘weak’. This will become clear in the rest of the paper.

2.3 Harel’s Statecharts
These ‘flat’ automata address absolutely not the complexity problem of the design process. To illustrate it, there is no need to count the implementation states, just the ‘functional modes’. A typical engine controller (avionics or automotive) can contain 10 top-level functions, each having 10 modes: the number of functional modes is \(10^{10}\).

So, ten years later, in 1965, Petri nets appear, and again ten years later, in 1976, SDL appears. Both will meet great successes and are still in use today, but they will never be considered as a universal method of design. The first potential Grail will appear 30 years later, in [Harel84][Harel87]: the ‘Statecharts’. David Harel creates them in 1983 to enable an avionics engineering team to elucidate the system behavior of a new fighter. In the introduction of [Harel07], he presents them as the child of <<theoreticians venturing out into the trenches of the real world, “dirtying their hands” and working closely with the system’s engineers.>>. Academic variants are countless, industrial ones are at least 4 (MathWorks Stateflow is the subject of the next paragraph, and [CraneDingel05] surveys 3 other versions).

SDL had already introduced communicating parallel automata (this is the paradigm suggested in the complexity illustration of the first paragraph), Harel calls this ‘AND-decomposition’, but he generalizes this with fully-hierarchical automata: <<… the kernel of the approach is the extension of conventional state diagrams by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components.>>. He gives two motivations for the (exclusive) OR-decomposition:

- Clustering: several states having similar exiting transitions are a symptom of a hidden common behavior, which can be made explicit by grouping them into a ‘super-state’ from which a single “leave-any-state-inside” transition exits.
- Refinement: a state ‘engine running’ can be atomic from the point of view of a system engine, but composite from the point of view of a combustion engineer.

In a purely academic framework, Harel would probably have been satisfied with a basic AND/OR hierarchical extension, with ‘events’ for synchronization of parallel states. But the ‘real world’ needed
more. So he introduces the notions of ‘transition with [recursive] History’ (noted ‘\(H\)’ and ‘\(H^*\)’; also called shallow and deep), which permits to re-enter a composite state in the configuration of its last exit (there is even a special ‘clear-history’ action to reset to the initial configuration). A ‘pure’ Statechart is interfaced with its environment only via ‘external events’, and the conditions that guard transitions are rudimentary: he adds an ‘action’ language (provided by the Statemate environment) to add an algorithmic flavor. He partitions the actions of a state (Moore outputs) into ‘entry’, ‘throughout’ (‘during’) end ‘exit’ actions. In the following diagram on the left (fig. 37 of [Harel87]), Greek letters are events (e.g., \(\beta\) synchronizes the left and right AND-substates) and upper-case letters (from ‘S’) are actions (e.g., \(W\) is a Mealy action and \(S\) is a Moore entry action).

The diagram on the right (fig. 23 of [Harel87]) illustrates the multi-level transitions, which permits compact descriptions, but are a semantic nightmare. Harel recognizes: ‘Defining the formal semantics … is quite a delicate matter’. In fact, 30 years after this seminal paper, a consistent formal semantics is still a subject of academic research.

### 2.4 Mathworks Stateflow® and strong transitions

Stateflow appeared in 1997. The documentation doesn’t mention Harel or the Statecharts, but obviously the look-and-feel and the behavior are inspired by the Statecharts. In this paper we are interested only with ‘discrete’ Stateflow, but it is interesting to notice that fundamentally ‘continuous’ Stateflow and ‘discrete’ Stateflow are the same thing: the discrete use is the restriction of the ‘full’ Stateflow to a unique triggering event: the clock (more precisely, all the other events are synchronous with the clock).

Concerning the behavior, when we compare with Moore and Mealy charts, something fundamental is to notice: transitions are ‘strong’, i.e. they are the first computed thing, before the actions. Let’s illustrate this on the following boiler controller, captured at the cycle when ‘alarm’ becomes true:

The previous state is ‘Normal’, the next state is ‘Alarm’; and ‘led’ is (becomes) true: the action computed is the action of the next state.

From a user point of view this is completely natural. But when you compare with Moore and Mealy charts, where outputs were computed from the previous state (we said ‘in parallel’ with the next state), there is a subtle change:
We now introduce a new concept: the ‘active’ state of a cycle is the state which contains the actions to be computed in this cycle. In stateflow, ‘active’ = ‘next’. The interest will appear in the Scade part.

This choice has an unfortunate consequence: the previous state is no more accessible (in the previous model, we don’t know if we come from ‘Off’ or ‘On’), so from a fundamental point of view it is not ‘digital-complete’, and from an engineering point of view we cannot describe Moore automata. That’s why in addition to the standard Stateflows (called ‘classic’), MathWorks provides also a special kind of Stateflow with a Moore semantics.

2.5 SCADE 6, weak transitions and mode automata

SCADE® V6 appeared in 2008 and is the result of a deep (final?) synthesis on two questions: What is a transition? What is a mode? This synthesis was published in [ColaçoPaganoPouzet05].

2.5.1 Transitions: Weak or Strong?

The concept of active state is central to Stateflow’s semantics ([MathWorks17] ch. 3 or annex A), but strong or weak transitions are not mentioned (idem for Statecharts). In fact, this distinction is one of the two contributions of the academic synchronous domain (Esterel, Lustre, Signal) to the Statecharts semantics (the other one is the synchronous signal broadcast, much simpler than the ‘microstep’ semantics, but as signals are not in our unified subset, it is out of scope).

It begins in [Berry93] with a semantic clarification in the synchronous imperative language Esterel. (for our purpose, an Esterel program is simply a set of processes interacting via signals):

<< Consider a process p … Consider now a process q defined by the informal sentence “abort p when s” where s is a signal … Assume that s occurs for the first time at time i. … We want q to behave just as p up to step i-1 included. … there is a clear ambiguity at time i: should q perform … p …or should q terminate right away without letting p react? The experience of Esterel programming has shown that both interpretations have their use and that we indeed need two different kinds of abortion constructs.

- In the weak abortion construct s>p, process p is allowed to react at time i. … p can … perform its “last wills” …
- In the strong abortion construct s>>p, process p aborts at once and is not allowed to act at time i. >>

In [André95], it is transposed in the ‘SyncCharts’ automata language, which belongs to the Esterel ecosystem. It distinguishes 2 ways a transition can exit (abort) a state to enter another one:

‘OFF’ and ‘ON’ are not the names of the states, but their actions (the emitted signals). This is almost the modern notation of Scade (fig. 2a shows strong transitions, 2b weak ones), and the column ‘k’ of
Table 2 shows that weak transitions have not yet the Scade semantics (instead of OFF.ON it should be OFF; here the two states are active in cycle k). SyncCharts will be the state machine formalism of Scade V5 (around 2003), and the efforts to go from an Esterel semantics to a Lustre semantics will lead to the current Scade V6 version, described in [ColaçoPaganoPouzet05].

To understand the cohabitation of strong and weak transitions in Scade V6, the easiest way is to show the dataflow graphical semantics (right-side drawing):

This graphical view is illuminating compared to the formal explanations of [ColaçoPaganoPouzet05]. It explains in an obvious way why there is a unique active state (at each cycle), and why weak transitions can be conditioned by the output flows, but the strong transitions cannot (only the previous values are accessible; this is the same thing in Stateflow). The concept of 'active state' becomes interesting (at least academically; it reduced to 'next' in Stateflow); for example, ‘transient’ active states become possible (neither equal to previous nor to next state). This is the ‘Swiss knife’ of transition systems: without weak transitions, it gives Stateflow, and without strong transitions, it gives Moore/Mealy machines.

2.5.2 Modes are dataflow variants
In Stateflow, actions are written in C or Matlab. [MaraninchiRémond98] introduced the beautiful concept of 'mode-automaton' which says simply two strongly related things:

- The actions of the different states (at the same level) are variants of the same function; in particular they have the same outputs (of course they can have different local variables).
- These variants are dataflow (sub-)programs; these dataflow blocks can contain again mode-automata, so we have 'for free' the AND/OR hierarchy with the cleanest possible semantics.

Of course, this implies a lot of restrictions compared to the StateCharts (typically concerning the multi-level transitions) but they are not ad-hoc: they are naturally justified.

The basic example of [MaraninchiRémond98] is a ‘triangle waveform’ generator, between 0 and 10:

If you write it in Scade, you will not have a triangle. The problem is that X is shared between two blocs (the incrementer and the decrementer) and for technical reasons, Scade designers have decided to give a ‘special’ meaning to ‘pre X’ in this context: it gives X previous value computed by the very same block (a kind of ‘local’ previous value), and not the ‘global’ (shared) previous value. The shared previous value of X is given by a new construction, “last X “. Both constructions can cohabit, leading to subtle behaviors.
An unexpected (first noticed in [DufourCorrubleTavernier16]) contribution of mode automata is that they highlight a formal difference between the two informal visions of the OR-decomposition, ‘clustering’ (Stateflow) and ‘refinement’ (Scade). In the following Stateflow, S1x is a ‘superstate’ which, instead of containing actions, contains a ‘sub-stateflow’ (which itself contains another superstate S2x):

![Stateflow Diagram](image)

The state S2 is only reached at time 2 (cycle 3). To summarize, each cycle fires at most one transition.

Let’s look at a similar automaton in Scade (the screen has been captured at the first cycle):

![Scade Diagram](image)

We see that we reach ‘instantaneously’ State4. In fact, the property ‘each cycle fires at most one transition’ is still true, but separately inside each automaton. What happens is that State_2x is not a ‘superstate’: it is (more precisely it contains) an automaton <SM2>, which at entry fires a ‘true’ transition from State2 and enters State_3x, which itself is (contains) an automaton <SM3> etc. To summarize this, just remember that in Simulink a sub-automaton is just a drawing artifact (‘cluster’), whereas in Scade, a sub-automaton is a true (refined) state-machine.

3. Unified state-machines, model checking and way forward

Our Scade-Stateflow intersection has not fundamentally evolved compared to [DufourCorrubleTavernier16], except for one thing: convinced by the hardware use-case, we now accept Scade weak transitions, but without mix with the strong ones. Weak-only automatas correspond to ‘Moore’ Stateflows, and strong-only to ‘classic’ Stateflows. The accepted modelling features are still ridiculously limited compared to [Scaife&Coll04].

The fundamental divergence concerning the semantics of hierarchy is still the hard point. It goes beyond technique: even the basic question ‘what is the most natural semantics for a designer? Clustering or refinement?’ has not a clear answer.

However, there is a pragmatic way to manage the associated risk of misunderstanding. With our restrictions (in particular, no AND-decomposition), it is easy to ‘flatten’ a hierarchical automaton. We build a diagram which compares an automaton with its flattening, and we submit it to the Scade model checker (Design Verifier). If the computed objects of the automaton (in transition conditions and state actions) are only booleans and bounded integers (typically the elapsed time in a state), then Design Verifier will succeed and will exhibit differentiation scenarios (if any), permitting the designer to ask himself: ‘what did I mean?’. The following figure illustrates this on a simple model of a boiler controller.
The way forward is at the software/hardware specification level (High Level Requirements in DO178 terminology): what is the automaton specification language which supports formal refinement towards the unified design state-machines? Together with a methodological interdiction of ‘hard-coding’ (and ‘hard-specifying’) of automata, it will enable the formal guarantee of the ‘exclusive-or’ behaviors mentioned in the introduction, this at the adequate level (hardware/software specification or system design).

4. References


[Glinz02] M. Glinz. Statecharts for requirements specification – As simple as possible, as rich as needed. ICSE workshop on scenarios and state machines, Orlando, May 2002.


