The Unified Model-Based Design: how not to choose between Scade and Simulink

Jean-Louis Dufour, Bertrand Corruble, Bertrand Tavernier

The Unified Model-Based Design: how not to choose between Scade[1] and Simulink[2].

Jean-Louis Dufour, Bertrand Corruble, Bertrand Tavernier.

Sagem Défense Sécurité, groupe SAFRAN.

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', interesting in two respects: on the academic side, it gives a simple semantics to our subset of Simulink, and on the industrial side, it permits at almost no cost the double-skill Scade/Simulink for software and system engineers. For the data-flow part, the unified subset is simply the **'clockless'** part of the Scade/Simulink intersection. For the control-flow part, the unified subset is much more restricted, but the fundamental aspect is that it strictly conforms to a well-known paradigm called **'mode-automatas'**.

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

1. The 'Unified Model-Based Design' concept

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 actor is Simulink, but for certified systems (avionics [our context is DO178 DAL A], railway, nuclear), it has to share the cake with Scade. The simplest (but fuzzy) definition of the subject of this paper, the 'Unified MBD', is the intersection of Scade and Simulink. It is an answer to two problems.

The first and principal problem is an industrial and human one. When the R&D team cannot choose between the two tools, this has unpleasant consequences. Here are the ones we want to address:

- It potentially doubles training costs, and developers have a 'cold start' at each language commutation,
- the choice of the 'right' language for in-house libraries can be Cornelian, because it dictates the language for the related products.

The second design driver is simply the main limitation of each tool (in the narrow context of software engineering):

- Simulink's DNA is development, certification was handled only recently: the Simulink subset we use has no formal semantics, contrary to Scade, and the tentative formal semantics are too sophisticated to be of any use in an industrial context ([HamonRushby2007]),
- Scade's DNA is certification, ease of development is perfectible: the Scade simulation environment is suitable for unit debugging, but integration and functional debugging is better done out of the tool, contrary to the seamless environment of Simulink.

This very last point is one of the two keys of Unified MBD: our wish list to Scade contains today mainly tooling points, because **the Scade language has reached a good expressiveness level, almost sufficient to deal with most critical applications**. To be completely clear, when we look at the C source of the application part of our critical avionics software currently under Model-Based development (the other parts are the Operating System and the Boot):

[1] ANSYS, Inc.
[2] The MathWorks, Inc.

- A few percent come from manual coding, because datatypes are low-level (bitfield manipulations) or algorithms are low-level (sin, sqrt) or sequential (sorting an array, Gram-Schmidt orthogonalization),
- a few percent come from automated coding of state-machines (because our applications contains few moding and are mainly algorithmic; on cockpit or in railway, it could be very different),
- and everything else comes from automated coding of dataflow diagrams.

Independently of this dataflow/state-machine ratio (which is application-dependent), we make a different use of the dataflow features and of the state-machine features:

- almost all dataflow constructions are used: they are very similar in Scade and Simulink, are very intuitive (well readable by system engineers, this is a key point for peer reviews). This will be illustrated in the next section;
- 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. This will be illustrated in the 3rd section.

Let's call 'Simple Scade' the part of Scade mentioned above (almost complete dataflow + almost nothing of state-machines). The second of the two keys of Unified MBD is that **Simple Scade is syntactically included into Simulink**. What does it mean? It means that without any semantic analysis, you can structurally translate (almost box to box and wire to wire) a Scade diagram into an 'equivalent' Simulink diagram. Equivalent is put in quotation marks because without a formal semantics for Simulink, it cannot be proved, but it can be verified by test (as in [Caspi&Coll2003]) or by expert judgement.

Let's call 'Simple Simulink' the image of Simple Scade under the syntactic translation. Contrary to Simple Scade, which is a syntactic subset of Scade, checking that a Simulink model belongs to the Simple subset requires a semantic analysis (simple: the transposition of Scade typechecking into Simulink). But once you are in the Simple Simulink subset, you can again syntactically translate towards Simple Scade.

Now we can give the **definition of the Unified MBD: it is the 'pack' formed of Simple Scade, Simple Simulink and their two syntactic translations**. The two translations have been prototyped, to check the robustness of the concept (the translators are not qualified, and we don't plan to reuse certification credits). This concept can be seen as a 'simple industrial version' of the academic works [Caspi&Coll2003] and [Scaife&Coll2004]. The knowledge of what is 'useful' was the key to keep things simple, and on this subset we claim to have the first 'truly operational' semantics of Simulink, usable in an engineering context.

To be completely clear, we must emphasize two points:

- The translators are really just proofs of concepts, and are far from being industrial tools. They have been quickly developed in MATLAB[3] to take advantage of its Simulink API for reading / writing / checking models (Scade models are stored in XML, for which MATLAB has also an API).
- The Unified MBD is not an Esperanto (i.e. a 3rd modeling language): it is more like a duo 'British Basic English[4]' / 'American Basic English', and the translators mentioned above simply change some 's' into 'z' or vice-versa. Projects still use Scade or Simulink, and the Unified MBD only appears as modeling rules. In this, it can be opposed for example to Synoptic

[3] The MathWorks, inc.
[4] https://en.wikipedia.org/wiki/Basic_English

[Cortier&Coll2010], which proposes a new modeling language, but it cannot be opposed to the P project [PothonBordin2013], because the Pivot language is not a language for modeling but for communicating models between tools.

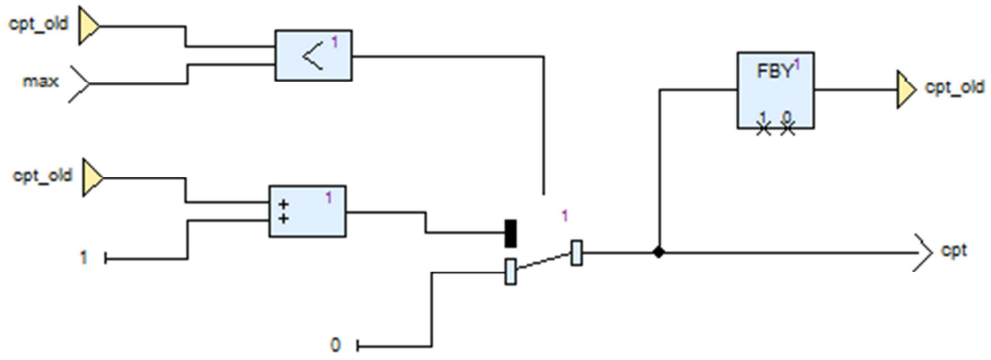
2. The dataflow part

Scade is not a pure data-flow language: it is a 'synchronous' data-flow language. It means that computation is not only driven by the values of the signals, but also (and firstly) by their 'clocks'. It permits compilation (pure data-flow is not really compilable, excepted in asynchronous hardware). In practice, clocks are not used, even for multi-tasking applications, and when you need to suspend an operator, you 'conditionally activate it on a value' and not on a clock: this is illustrated in the next sub-section. The reason is that it is conceptually simpler for engineers (the generated code is often the same), this is to be contrasted with the hundreds of academic papers on clock-calculi that have been published in the thirty past years. This is the main characteristic of the unified data-flow: to be **clockless** (technically speaking, we should say 'single clock', but clockless is more meaningful).

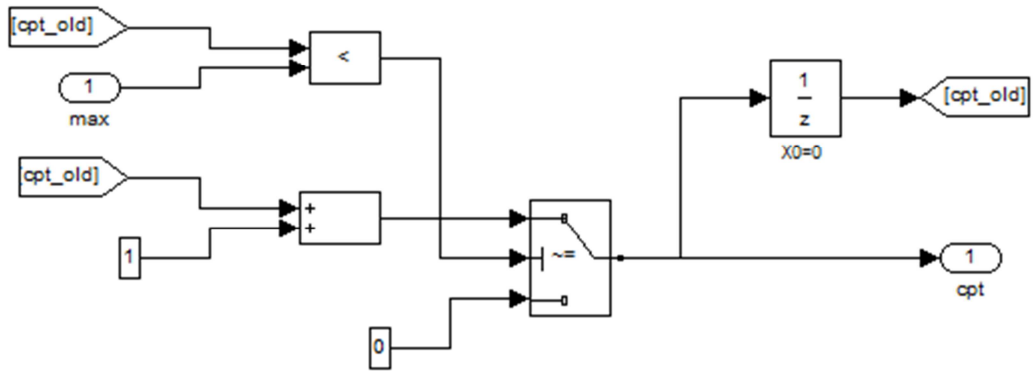
This section is structured according to the well-known equation 'Algorithms + Data Structures = Programs'.

2.1 Algorithms

The syntactic translations are in general completely obvious (this is the very reason why the Unified concept makes sense). Here is a standard cyclic counter in Scade

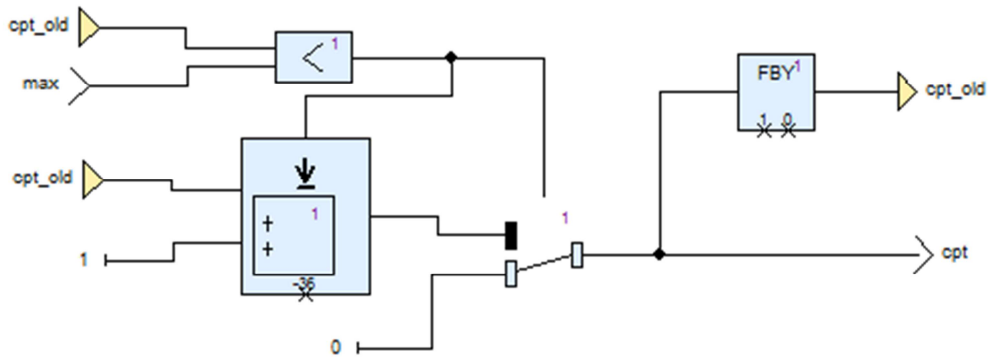


and its translation in Simulink:



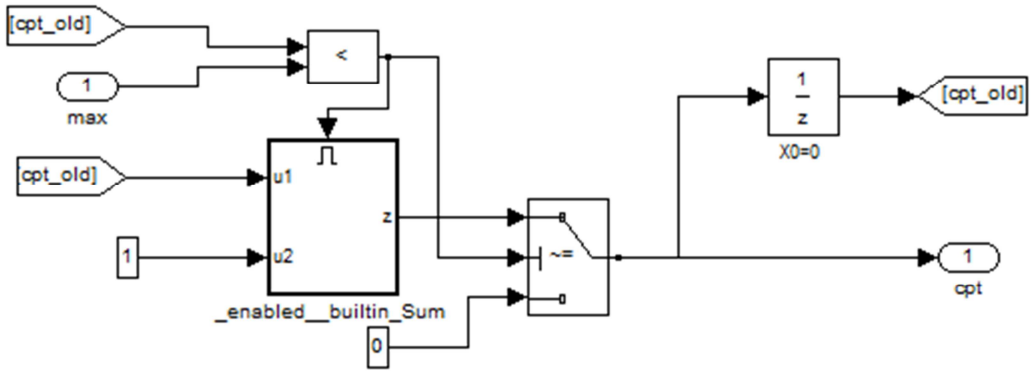
The not-completely-obvious aspects are with what Scade calls 'higher order patterns', typically the conditional activation or the map: in Scade they operate on a sub-block, whereas in Simulink they are in the sub-block (they 'qualify' it). Following the MBD motto (and long before, a famous French

statesman), a small drawing is better than a long speech: here is the same counter, but with a conditional addition (to explicitly avoid overflow).

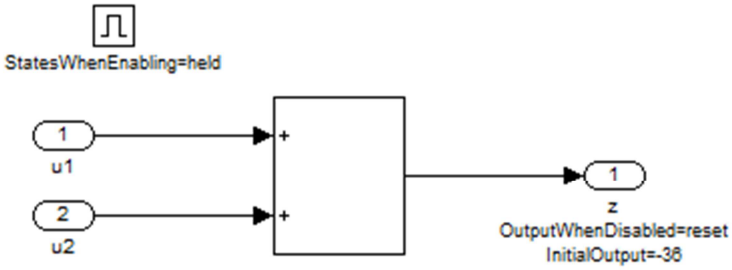


The strange block with a downwards arrow means ‘either activate the sub-block ‘+’, or return -36’. The default value -36 is part of the top-level block and not of the sub-block. The sub-block is the standard ‘+’, which doesn’t care about conditional activation or default result.

The Simulink translation looks identical, but in fact it is not:



The sub-block ‘_enabled__builtin_Sum’ is not the standard ‘+’: it contains an ‘Enable Port’ block and the default value -36 as a parameter of the output port ‘z’:



The translation can be improved by using the ‘block parameters’ mechanism of Simulink: this way, the ‘-36’ will appear in the top-level block, but there still will be a supplementary hierarchical level compared to Scade. The translation is syntactical (and reversible), but the structure is slightly changed.

2.2 Data structures

The only subtlety is the definition of vectors and arrays.

On the one hand, Scade, like C or Ada, is completely rigorous about that:

- a scalar is not a vector of dimension 1,
- a matrix is a vector of vectors,
- there is a unique 'vector constructor', which construct vector from scalars, matrices from vectors, etc… (same simplicity for 'vector projector'),
- the dimension of a vector is explicitly declared, and to permit generic algorithms (dot product for example, for any dimension), symbolic dimensional parameters 'M', 'N', etc. can be used, with a subset of arithmetic. For example, the concatenation of two vectors of size 'M' and 'N' is a vector of size 'M+N'.

On the other hand, Simulink, like Matlab, tries to be user-friendly, but to make things more 'interesting', it is not user-friendly in exactly the same way that Matlab is:

- a scalar is a vector of dimension 1 (in Matlab, it is a 1*1 matrix because vectors don't exist [row and column matrices do the job]), and is often equivalent to a 1*1 matrix,
- a matrix is not a vector of vectors,
- you cannot use exactly the same built-in blocks (or the same configuration of …) to construct/project vectors and matrices,
- the dimension of a vector or a matrix need not to be explicitly declared (even if it has to be statically known for code generation purposes).

The first consequence is that the Scade→Simulink translation is sometimes dimensionality-dependent, but hopefully this information is contained in the Scade diagrams: the translation is still syntactic.

The second consequence is more critical: for the moment, generic vector/matrix algorithms cannot be syntactically translated from Simulink to Scade, we can only syntactically translate each instance separately. Type-checking vector/matrix generic algorithms is a complex problem (see [JoishaBanerjee2006] for MATLAB), but hopefully, academic research has always kept this subject on its radar screen. Historically, it had purely academic motivations like APL evolutions or compiler efficiency motivations for FORTRAN, but today there is a new boost driven by parallelism/many-cores, and the concept of 'shape and dimension polymorphism' is increasingly studied. Concerning our Simulink→Scade translation problem, the type inference mechanism implemented in the 'Repa' package of Haskell ('Regular Parallel arrays') is not far from our needs [Keller&Coll2010].

3. The state-machine part

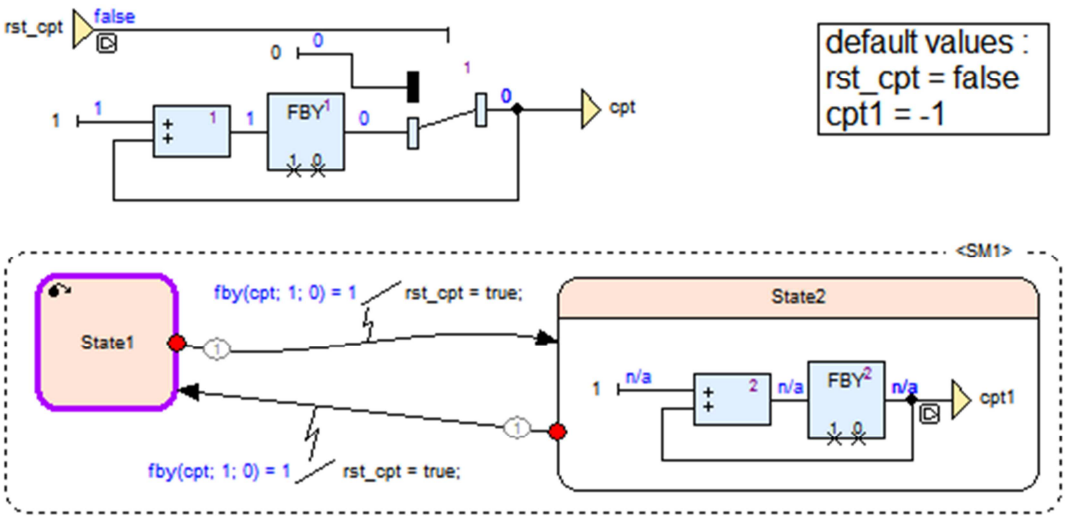
In term of 'Scade-Simulink intersection', state-machines are the exact opposite of the dataflow part: almost nothing can be unified (if you want to ensure the same look-and-feel and simple translations). In fact, as soon as you introduce hierarchy, the semantics differ. To try a simple explanation, in Simulink a sub-automaton is just a drawing artifact, whereas in Scade, a sub-automaton is a true state-machine.

Hopefully, the state-machines we use are ridiculously simple compared to the expressiveness of both state-machine formalisms, and the small intersection (non-hierarchical) makes sense. We have interesting use-cases of (one-level) hierarchy, so the subject is not closed, but for the moment the unified subsets are non-hierarchical.

3.1 The SCADE 'Unified Modeling Style'

Scade designers have pushed the dataflow/state-machine integration to the extreme, we will illustrate this on the following operator which generates two integers 'cpt' and 'cpt1' such that:

If cpt1>=0 then cpt = cpt1



The state-machine <SM1> is not an operator (the operator is the whole figure): it is just an equation (i.e. a definition of variables), at the same level than the dataflow definition of ‘cpt’. It defines the two variables ‘rst_cpt’ and ‘cpt1’, and what is interesting is that ‘cpt1’ is defined by a dataflow INSIDE ‘State2’. In fact, ‘State2’ is an operator, working ‘part time’ (in the figure it is at rest, hence the ‘n/a’s), which could contain itself another state-machine, etc.

The underlying paradigm is extremely clean and elegant: instead of ‘state-machine’ (a dataflow containing a FBY [i.e. an 1/z] is also a state-machine, after all) we should speak of ‘mode-machine’ (or ‘variant-machine’, ’schizophrenic-machine’, ‘Janus-machine’, etc.). The difference is that a state is just a value, whereas a mode is a behavior: a mode is simply an operator implementing one particular personality of many. The first data-flow embodiment of this paradigm were the ‘mode-automatas’ of [MaraninchiRémond1998].

The Scade implementation of this paradigm (see [Colaço&Coll2005] or the Scade Suite documentation) is rightly called the ‘Unified Modeling Style’ (not to be confused with our ‘Unified Model-Based Design’). And as many implementations, it is (substantially) more complex than its specification: to give an idea, let’s just say that none of the three authors has ever reached the end of the Scade Language ‘Tutorial’ [Esterel2011], which models mainly with state-machines a descendant of Gérard Berry’s stopwatch (published in [Berry1991], see also [Halbwachs1993] ; itself a descendant of Harel’s wristwatch [Harel84]).

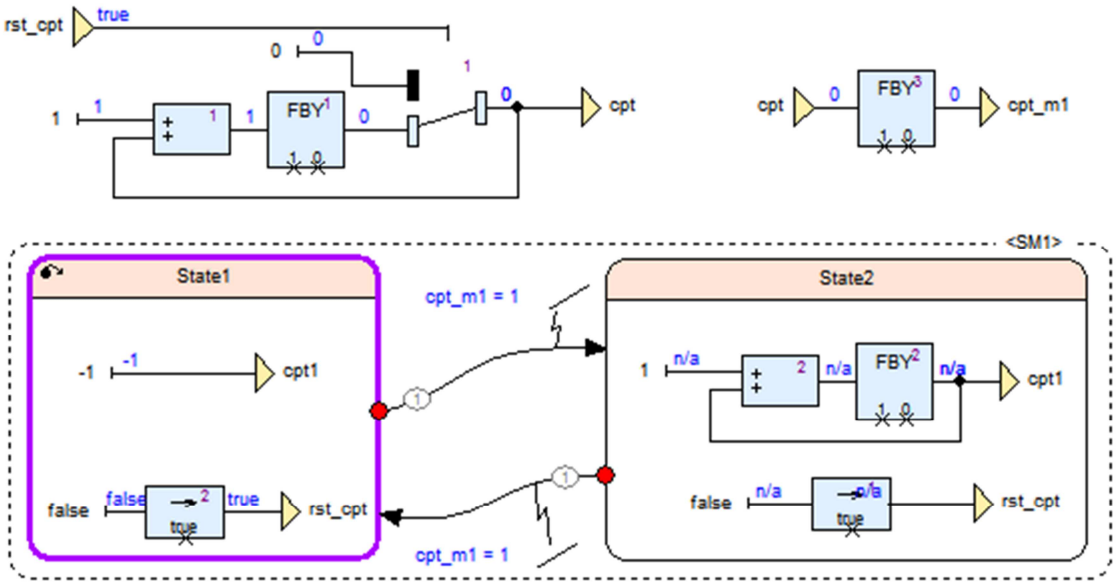
It would be too long to exhaustively enumerate all the limitations we impose on state-machines: weak/synchro/resume transitions, signals, ‘last’ operator, etc. In fact almost everything has been forbidden, and it is easier to explain what we kept. Let’s call a state-machine ‘simple’ when

- the states are independent operators (no ‘last’: no sharing between states), restarted (and not resumed: inner states are re-initialized) at each entering transition, and contain only (a subset of) built-in operators,
- transitions are ‘strong’, i.e. their conditions are the first thing computed, then the new state is determined and executed,
- transitions have no actions, no events and combinatorial conditions (no temporal operator).

So, this is a minimalist interpretation of the paradigm. To transpose the former operator into the unified subset, we have to:

- replace the default value mechanism on ‘cpt1’ and ‘rst_cpt’ by an explicit definition in every state,

- replace the two actions on transition by adequate initializations in the modes (transition actions are pushed inside the modes),
- replace the ‘fby’ in the conditions by the elaboration of a new variable ‘cpt_m1’ (the ‘fby’ is pushed outside the state-machine).



This last paragraph is a bit technical and can be skipped; it is just an illustration of the countless subtleties that occur with state-machines, which we consider to be inacceptable in a safety-critical design. If you observe only the outputs, the two previous diagrams are indistinguishable. But if you observe also the inner signal ‘rst_cpt’, there is a difference: it is false in the first diagram and true in the second (look at the upper-left corner of the diagrams). It occurs only during the first cycle (the initial start or ‘cold start’), and it is the reason why the first diagram is not accepted by our Scade→Simulink translator (the 2nd diagram is OK). More precisely, <SM1> in the first diagram is a state-machine, but is not a ‘mode-machine’, because ‘State1’ is not an operator: it behaves differently in the initial start (rst_cst = false) and in a warm start (a restart in the middle of a run, coming from State2; in this case rst_cst = true). In other words, State1 has an ‘extra-sensory’ knowledge (not given by its inputs and state) of the fact that a start is an initial one or a warm one. Again in other words, if we had to translate this in the unified subset, we would have to add a supplementary input to State1 to give him this information. We consider that it is no more a syntactical translation, but more fundamentally we consider that such a subtle semantic point is not acceptable in a safety-critical design.

3.2 Simulink: classicism

State-machines in Simulink are built with a toolbox called ‘Stateflow’. In terms of expression power and complexity, they are comparable to Scade, but they are ‘classic’ w.r.t. Harel’s ‘seminal synthesis’ [Harel1984]: the behavior of the states is sequential code.

The search for semantic simplicity and the intersection with Scade have led to a very reduced subset, for example from the most general form of transition

‘event_trigger[condition]{condition_action}/transition_action’

we keep only

‘[condition]/transition_action’

with global constraints on the transition actions in order to ensure the ‘mode-machine’ aspect.

3.3 The representation of the intersection

Concerning the dataflow part, the specification of the translations is very simple, from ‘drawing’ to ‘drawing’, and the implementation is just as easy, from ‘syntax tree’ to ‘syntax tree’.

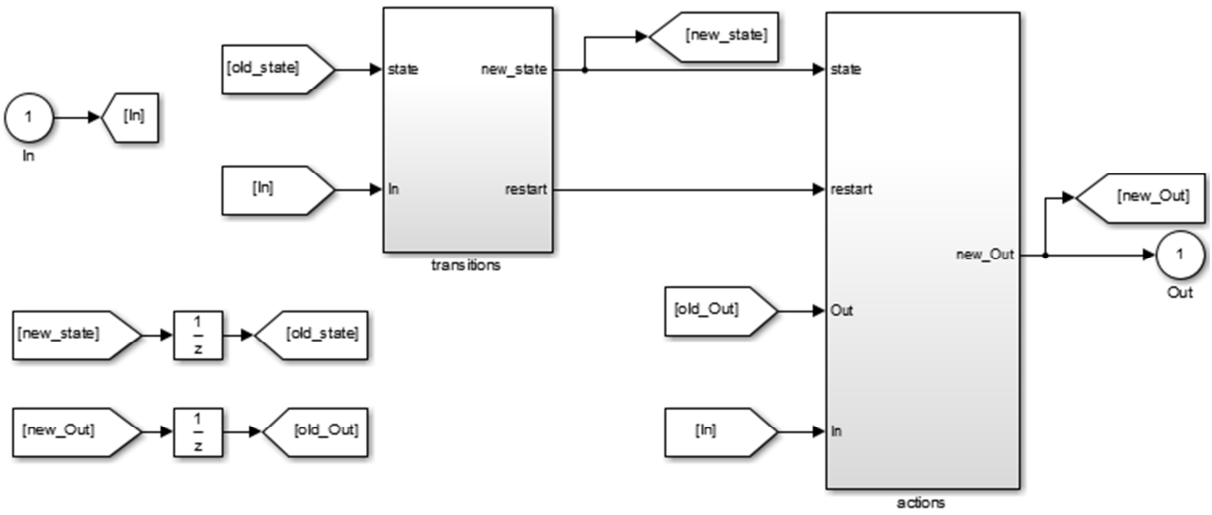
Concerning the state-machine part, things are different, and the need for an intermediate language was felt very quickly. The classical approach is to design a third language, and we could have used the ‘Pivot’ language of the ‘P’ research project of which we are partner ([PothonBordin2013]). But the aims of the P-project and of the Unified MBD are not the same, which explains why we have adopted another solution:

- the P-project needs a pivot language for ensuring semantic correctness of transformations of models, nobody will never directly read or write P models,
- the Unified MBD needs an operational, but also simple and readable way of specifying (for human end-users) subsets of Scade or Simulink.

Our solution for the specification (and translation) of unified state-machines is a dataflow design-pattern, consisting of (see also next figure):

- a set of states (left-bottom part of the diagram),
- a pure (stateless) function ‘transitions’ computing the next state and the restart order (middle of the diagram),
- a pure (stateless) function ‘actions’ using this next state and restart order to compute the output(s) (right of the diagram).

So, when we want to incorporate a new feature of, say, Stateflow, into the Unified MBD, we don’t have to check the simplicity of translation in Scade: we check the simplicity of translation in this design-pattern. This is not a third language; this is a pivot design pattern inside the Unified dataflow: formal semantics of state-machines comes ‘for free’.



4. The Standard Library, and conclusion

A common in-house ‘standard’ library has been defined, with basic operators like matrix computations and filters. The Scade and Simulink version share their specification and their test-cases. This way, we claim to have unified the look-and-feel of our models and to permit at almost no cost the double-skill Scade/Simulink.

The goal of this paper is to show that for critical software, the interesting subset of Scade/Simulink is not so big than that. But of course, the way forward is to continue to extend the two subsets, according to use-cases needed by development projects.

5. References

[Berry1991] Gérard Berry. Programming a digital watch in Esterel v3. TR 08/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, 1991.

[Caspi&Coll2003] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. International Conference on Embedded Software (EMSOFT), 2003.

[Colaço&Coll2005] J-L Colaço, B.Pagano and M. Pouzet. A conservative extension of synchronous data-flow with state machines.EMSOFT'05.

[Cortier&Coll2010] A. Cortier, L. Besnard, J.P. Bodeveix, J. Buisson, F. Dagnat, M. Filali, G. Garcia, J. Ouy, M. Pantel, A. Rugina, M. Strecker, and J.P. Talpin. Synoptic: A Domain-Specific Modeling Language for Space On-board Application Software. Chapter 3 of the book edited by S.K. Shukla and J.-P. Talpin 'Synthesis of Embedded Software', Springer, 2010.

[Esterel2011] Esterel Technologies, Scade Language Tutorial, 2011 (available only with Scade Suite).

[Halbwachs1993] N. Halbwachs. Synchronous Programming of Reactive Systems. Springer, 1993.

[HamonRushby2007] Grégoire Hamon and John Rushby. An Operational Semantics for Stateflow. International Journal on Software Tools for Technology Transfer (STTT), 2007.

[Harel1984] D. Harel. Statecharts, a visual approach to complex systems. Dept. of Applied Math. Weizmann Institute of Science, Rehovot, Israel, 1984. Revised edition: Statecharts: a visual formalism for complex systems. Science of Computer Programming, 1987.

[JoishaBanerjee2006] P. Joisha and P. Banerjee, An algebraic array shape inference system for MATLAB, ACM ToPLaS 28(5), 2006.

[Keller&Coll2010] G. Keller, M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, B. Lippmeier, Regular, Shape-polymorphic, Parallel Arrays in Haskell, ICFP, 2010, https://hackage.haskell.org/package/repa

[MaraninchiRémond1998] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. European Symposium on Programming, 1998.

[PothonBordin2013] F. Pothon and M. Bordin, Towards the qualification of open-source code generators: project P, int. conf. Certification Together, 2013.

[Scaife&Coll2004] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. International Conference on Embedded Software (EMSOFT), 2004.