

Simulink/Stateflow*

Paul Caspi

Pieter Mosterman

<http://www-verimag.imag.fr/> <http://www.mathworks.com/>

June 2008

1 Introduction

Probably, the early designers of Simulink in the late eighties would have been surprised if they had been told that their emerging tool would appear some 15 years later in a collection devoted to real-time languages. In effect, what they were aiming to was to build a graphical modelling and simulation tool on top of Matlab, a command line framework having the same purposes. What was then the evolution which has led us to view this surprise as a rather obvious fact now?

1.1 The Emergence of Model-Based Design

From the very beginning, control engineers have been led to reason in terms of models, first plant models and then controller models. In particular, controller models were natural as, at these times of analog technology, those controllers were composed of the same modelling artifacts as plant models: differential equations, transfer functions and the like. Then the implementation technology evolved toward discrete time components and computer programming but the modelling technology was able to follow this evolution by moving also to sampled-data models, difference equations, discrete transfer functions etc. In this way, control engineers were able to design and simulate discrete-time controllers in close loop with plant continuous-time models and, thus, they could analyse and predict what would be the expected behaviour of their designs.

*<http://www.mathworks.com/>

1.2 Automatic Code Generation

Then some people of different origins, academics, industrials, noticed that they could automatically generate the code out of these controller models.¹ This technique has become increasingly accepted² and from that time on, the distance between modelling and programming has become also shorter and shorter.

1.3 Hierarchical and Parallel State Machines

Another branch of modelling, more of a computer science origin, concerns automata and machines. Here, many improvements were brought starting from the eighties, Petri nets and process algebras notably, leading to practical tools like StateCharts [9] and Esterel [6]. In this field also, the distance between modelling and programming became always shorter. Now, modern control systems are complex ones, mixing both continuous control and discrete event control and this entailed a need for mixing both models. Thus mixed automatic code generation also became a useful feature.

2 Simulink/Stateflow: an Overview

2.1 Simulink

As said above, Simulink is a graphical modelling and simulation tool-box which is mainly devoted to continuous time (differential equations) and discrete time (difference equations) systems. Its graphics is essentially based on the block-diagram paradigm, *i.e.*, systems are built by interconnecting subsystems and blocks which are provided from libraries and are the elementary building blocks from which subsystems are composed.

2.2 Stateflow

Stateflow is also a graphical modelling and simulation tool which is based on the hierarchical and parallel state machine paradigm essentially borrowed from StateCharts. It has been fully integrated into Simulink so as to allow

¹One of the first instances of this discovery is due to the engineers of Aérospatiale in Toulouse, France, who designed the first computerised flight control of a commercial aircraft, the Airbus A320.

² There should be particular reasons for sticking to human coding after having performed a model-based design; one of these is performance but, as it has been the case with high level programming languages, efficiency is less and less an issue.

the simulation of mixed continuous time, discrete time and discrete event systems. It is worthy to notice that, to our knowledge, Simulink/Stateflow has been the first tool-box to achieve such an integration.

2.3 Real-Time Workshop

Finally an automatic code generator called Real-Time Workshop has been added to this tool chain. Real-Time Workshop is a very versatile tool which allows the generation of single thread code as well as multiple thread one. This allows the tool chain to fully cover the model-based design paradigm.

2.4 Third Parties Tools

An important aspect of this tool-chain is that, due to its popularity and versatility, many third party tools can be integrated into it. We can cite here some of them which also contribute to the real-time language issue:

Targetlink [2] and **ASCET** [4] are other code generators that start from Simulink diagrams and generate code, mainly in automotive contexts. It is noticeable that, as a matter of fact, these code generators have preceded the Simulink one.

SCADE [3] is a tool-box on its own which is specialised to generating code for safety critical control applications thanks to a code generator which is qualified according to the most severe safety standards of avionics, automotive and railway industries. Note also that this toolbox also achieves the integration of block diagram formalisms [8] and state machine formalism [6]. But, due to the popularity of Simulink and because it lacks the continuous-time modelling facility, it can be connected to Simulink as another automatic code generator specialised for safety-critical applications [7].

TTMatlink [5] and **Simtarget** [1] are code generators specialised for distributed systems based on the time-triggered paradigm. Thanks to these tools, not only multiple thread code but also fully distributed code can be generated.

3 Design and Programming examples

Should we add something there ? I have lots of simple examples we can put here.

4 Pros and Cons of Simulink/Stateflow

4.1 Pros

Some of the pros have already been mentioned: the popularity of the tool, its ability to encompass several modelling domains, and the huge number of third party tools associated with it makes it a must in the domain of real-time control. Other aspects are worthy to notice:

Its strong mathematical bases: Simulink is mostly based on strong mathematics which are basic to control engineering, like differential and difference equations, transfer functions, Laplace and Z transforms. To some extent, we have already noticed that Stateflow builds also on strong concepts of automata and machine theory.

Some appealing structuring concepts: These concepts are mostly inherited from the block diagram structuring paradigm. Besides making it a fully hierarchical formalism, it also makes it mostly a *functional* formalism which allows it to get rid of side effects. In this way, Simulink subsystems can also be seen as *components* which can be safely reused while keeping the same input/output behaviour.

4.2 Cons

Yet there are also several difficulties in using Simulink/Stateflow which are mostly related to historical reasons: as a matter of fact, the tool was not initially meant and designed for code generation but for modelling and simulation. It thus lacked many desirable features of a programming language which had to be added in the course of time when the interest of automatic code generation was discovered. But this addition of new features had to still preserve backward compatibility and this was clearly a very difficult task. The result is thus a very complex tool with many features, lots of possible tunings which require a real expertise to be mastered. A possible solution to these difficulties is subsetting and guideline definition and this is what is done in many industries that use it. Another solution consists of using different code generators as the ones mentioned in section 2.4: indeed, these code generators also play the role of filters, discarding those features which are not clean and could be error prone. For instance, using SCADE as a filter and using its qualified code generator is likely to provide a very safe modelling and code generation tool chain. This kind of approach is actually in use in the most demanding industries.

Among the main weaknesses we can mention:

The lack of formal semantics: As a matter of fact, since the first goal was simulation, the only semantics that was given was the one of the simulation engine³ Furthermore, since the simulation engine has many settings, it may be the case that the semantics depends on these settings and this makes the semantic problem even more difficult.

The initial lack of strong static checks: Similarly, simulation doesn't require strong static checks as, for instance, type checking. Indeed, initially, Simulink didn't care about data types because, for a control engineer, the only relevant data types were reals and complex numbers! Then, for the sake of computing and code generation, data types have been introduced but as an *a posteriori* improvement.

Difficulties in the semantics of parallel automata: This is a major drawback of Stateflow which triggers lots of warnings in the Stateflow documentation and originates many guidelines. Some of the difficulties are described in [10]. It can be seen also as a by-product of simulation: the semantics is what is provided by the simulator even if the simulation can go into infinite loops. But this makes it hardly usable in a safety-critical application.

5 Conclusion

The success of Simulink/Stateflow shows that this tool has fulfilled a real need and it is likely that its use will go increasing in time as a modelling, simulation, real-time programming and code generation tool chain. This is mainly due to the model-based design paradigm: the user is almost guaranteed that her design will behave as what she sees in simulation. Moreover, the designer in some sense doesn't need to be a computer engineer: she can forget computation and concentrate on what matters her, control, behaviours, architecture etc. Yet the tool needs still to improve so that users can get confident in it, above all in safety contexts. Meanwhile mixed usages like the ones suggested in 4.2 may be preferred.

³This has been a drawback shared by many tools based on simulators, like for instance VHDL: people had then to spend lots of efforts finding what was the semantics of VHDL.

References

- [1] <http://www.decomsys.com>.
- [2] <http://www.dspace.com>.
- [3] <http://www.esterel-technologies.com>.
- [4] <http://www.etas.com>.
- [5] <http://www.tttech.com>.
- [6] G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [7] P. Caspi, A. Curic, A. Maignan, Ch. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. *ACM Trans. Embedded Computing Systems*, 4(4):779 – 818, September 2005.
- [8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [9] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of NATO ASI Series, pages 477–498. Springer Verlag, 1985.
- [10] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo, editor, *4th International Conference on Embedded Software, EMSOFT04*. ACM, 2004.