From system functional definition to software code

David Lesens
Airbus Defence and Space

Keywords: Model Driven Engineering, SysML, Automatic Code Generation, COTS, DSL

Abstract: This paper addresses the classical problem of system to software engineering following a Model Driven Engineering (MDE) approach. Even if this approach is now widely used in the industry, some issues remain: Long term availability of the tools (for projects with duration of several decades), use of standards and Commercial Of The Shelf (COTS) tools versus Domain Specific Language (DSL), different modelling tools for the system and the software, quality and mastering of automatically generated code. This paper shows how it is possible to take simultaneous benefit of COTS (low price), DSL (adapted to specific needs) and in-house tools (which can be maintained for very long periods of time) to develop complex critical systems.

1. Introduction

Since several years, Model based System Engineering (MBSE) has been shown efficient to improve the capture of system requirements. Airbus Defence and Space – Space Systems has for instance successfully deployed SysML modelling to support the functional definition of the avionics sub-system of a launcher such as Ariane 5 Mid-life Evolution ([4]) or of a spacecraft such as the European Service Module (ESM) of the Multi-Purpose Crew Vehicle (MPCV, [11]). The functional part of the system definition is formalised by a SysML ([18]) model developed in co-engineering by the system experts and the modelling experts. Several documents are thus partially automatically generated from this single model: The system Definition Files, the digital Interface Control Documents (ICD) and the software Technical Specification. MBSE improves the communication between the teams and improves the generated documentation quality.

This paper shows how Airbus Defence and Space – Space Systems has defined a process and a set of tools to extend this approach to the software development (Model Driven Engineering or MDE). The objective of this work is to generate from a single model shared between the system and the software the three previously mentioned documents and a part of the software code.

After this introduction, the section 2 will quickly present the system functional modelling today deployed for the development of space launchers. The section 3 will summarize previous studies on software modelling and explain why these approaches have finally not been selected for the development of future space launchers. The section 4 will describe the specific multithreading design used for space launchers and how it has been decided to model it. Finally, the section 5 will show how an important part of the flight software can be automatically generated from these models. The section 6 will compare the proposed solution with the ones presented in section 3 and the section 7 will present the conclusion.

2. System functional modelling

The avionics subsystem of a space launcher is mainly responsible for (1) the flight control (navigation, guidance and control) and (2) the mission management (ground / board protocol, ignition and stop of engines, release of stages…). It is made up of a centralised on-board computer hosting the flight software (or potentially several computers working in a redundant mode to ensure the fault tolerance) and a set of avionics sensors (gyroscopes, accelerometers…) and actuators (valves, pyrotechnic commands, electro-mechanical actuators…) linked by a communication network.

The system is then cyclically executed: Measurement acquisition, flight control algorithm, execution of commands by actuators, and so on. The needs of the flight control toward the launcher system is translated into a set of blocks (hardware or algorithmic) communicating through data-flows. This functional architecture is modelled by SysML Internal Block Diagrams (IBD). The content of the algorithm blocks is textually specified or modelled in Matlab and then coded in Ada.

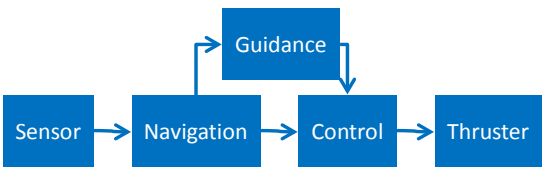

Figure 1: Functional data-flow architecture modelled in SysML IBD

Figure 2: Part of a launcher mission described by a finite state machine in SysML

The mission management describes the acyclic behaviour of the system. The phases of the launcher mission (ground phase, motor ignition and then lift-off or emergency stop in case of failure) are thus modelled by finite state machines in SysML and by a textual DSL (Domain Specific Language) adapted to the description of a space launcher mission.

The use of a COTS modelling tool (e.g. Rhapsody [14]) allows decreasing the costs: COTS tools are generally very mature and require only light customization. However, one of the main drawbacks of COTS is the difficulty of maintenance for a very long period of time. The use of a widely used standard ensures that editors will always be available in the future.

SysML has thus been selected because it is a standard widely used in the industry and in the academic world and because it provides the two kinds of diagrams needed to describe the functional architecture and the flight software of space launchers: Internal Block Diagrams and Finite State Machines.

The semantics of the modelling (SysML + DSL) is the one of Synchronous Languages such as Lustre ([8]) or Signal ([17]):

- The system is cyclically activated at a constant frequency.
- The inputs of the system are supposed to be available at the beginning of each cycle of execution.
- The outputs computed by the system are provided at the end of the cycle.

This approach is well known to ensure a full determinism of the system and allow simple compositions of elements with a predictable memory usage.

Here is an example of the DSL describing the execution in parallel of two commands and inspired by the notion of On Board Control Procedure (OBCP, [12], also called “functional sequences”):

```
sequence Lift-off is
        fork Cmd_Pyro;
        wait 5 ms;
        Valve_Cmd;
        wait end of Cmd_Pyro;
        Start_Control;
end;
```

Figure 3: DSL describing a functional sequence

This example is valid with respect to the synchronous semantics only if the duration “5 ms” is a multiple of the basic cycle of the system.

The communication network (related to the avionics system design) and the middleware and the threads (related to the software real-time design) are abstracted in this phase of development.

3. Previous studies on software design modelling

In parallel to the deployment of SysML modelling to capture the system functional definition, several modelling approaches have been assessed in order to capture the software design and to potentially generate automatically part of the flight software:

- SCADE Suite ([16]) was a promising track thanks to its certified code generator ensuring a high quality of the generated code. It has however several limitations
    - The SCADE modelling is partially redundant with the SysML modelling (especially for what concerns the finite state machine and the functional architecture). The development cost of these two models removes the benefits of the approach

- SCADE does not model today the multithreading architecture. Developing multithreading software with SCADE requires thus the independent modelling of each thread in SCADE and then the manual development of a real-time sequencer responsible for the activation of each thread.

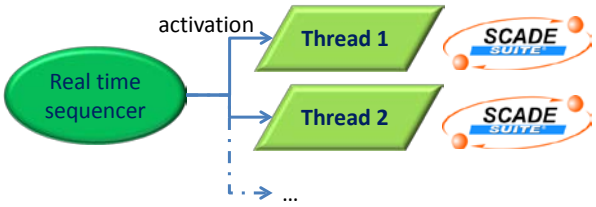

Figure 4: Multithreading modelling in SCADE

  The SCADE model becomes then a design model distinct from the functional model. This decreases greatly the advantages of the modelling
- The code generated from SCADE has not been designed to be manually maintained. This implies the need to maintain the SCADE modelling tool and the code generator during the complete life of the project (up to 30 years for a space launcher). The cost of this maintenance removes on the long term the saving of automatic code generation

- AADL ([1]) allows modelling the avionics and the software multithreading architecture. As for SCADE, this modelling is partially redundant with the SysML modelling selected for the system functional description. Moreover, AADL is able to capture any kind of software multithreading architecture, even the more complex. The flight software of a space launcher being critical, its multithreading architecture is very simple and based on the Rate Monotonic Scheduling (RMS [15], see section 4). AADL is thus too complex for the needs.
- MARTE ([9]) has the advantage of being close from SysML and UML. However, it has the same drawbacks than AADL.
- UML ([19]) is a modelling language especially well-suited for object oriented software design. As the flight software of future launchers will be partially developed in an object oriented manner (with the Ada 2012 [3] programming language), UML seems a good choice. However, the object oriented approaches of UML and Ada 2012 are not fully compatible (see [2]). For instance:
  - The UML notion of “class” is replaced in Ada by the notions of “package” (to define the naming space) and of “tagged type” (to define the inheritance)
  - Ada has a restricted set of “visibility” compared to UML
  - …

  Moreover, as for SCADE, the UML modelling is partially redundant with the SysML modelling. It has thus been decided that UML models will be developed to describe the principle of design, but not for the detailed design (except for some specific parts) and will not be used for automatic code generation.

As a conclusion of these studies, it appears that all the studied software modelling languages have two drawbacks: The redundancy with SysML (selected for the system functional description) and the cost to maintain the toolsets during a long period of time. Airbus Defence and Space has thus decided to rely mainly on SysML completed by a DSL (Domain Specific Language) to model the design of launcher’s flight software (the same DSL being used for the system functional behaviour (see section 2) and the software design).

4. Software design modelling

Defining a DSL requires an accurate definition of the objectives of the modelling. This section describes the flight software design used today on space launchers and which has been the basis to define the DSL.

The software design is composed of the static design (definition of the hierarchical architecture of the software, definition of classes and of objects, definition of the interfaces of components) and of the real-time design (definition of threads, of their scheduling, of the communication between the threads and of the communication between the software and the communication network). The main drivers of the flight software design are:

- The functional and real-time determinism ensuring the representativeness and the reproducibility of the qualification tests
- The decrease of the development and validation costs

The determinism of the multithreading design is ensured by using an extension of a Rate Monotonic Scheduling:

- The software is composed of cyclic threads (no acyclic threads)
- Each thread has a period multiple of the period of the thread just faster (harmonic threads)

- The scheduling is pre-emptive with constant priority (a thread with a shorter period has the priority over a thread with a longer period)
- The communications between the threads are performed during time triggered rendezvous

The Figure 5 provides the example of two communicating threads (a slow one in blue and a quick one in yellow).

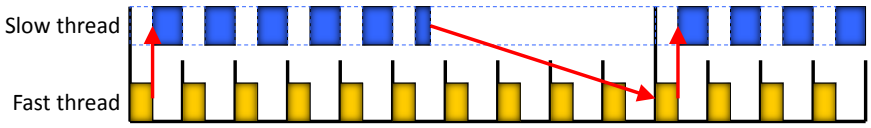

Figure 5: Time triggered communication between threads

The period of the slow thread is a multiple of the one of the quick thread. The quick thread has the highest priority. It can pre-empt the execution of the slow thread. The communications between the two threads are performed at the end of the period of the slow thread (in a protected section): even if the execution of the slow thread is quicker than expected, the communication will be performed at the same date. Thus, provided that each thread respects it worst case execution time, this design ensures the strict determinism of the software behaviour.

The decrease of the development costs is achieved by mapping the functional architecture and the software static design. This result is obtained by a co-engineering work between the system team (responsible for the functional architecture, see section 2) and the software team. The functional architecture shown Figure 1, which defines a set of functions and their communications, is thus directly used to generate the code (see section 5).

The work remaining at software design level is thus:

- To define the set of threads and their periods
- To map each function on a thread. Depending of the reactivity needs, a function at 10Hz can for instance be executed either at each cycle on a thread at 10Hz or at one cycle among two on a thread at 20Hz.

A simple DSL has been defined to describe this software design choices.

```
thread T1 is
        period (100 ms);
        functions (F1; F2);
end;
```

```
thread T2 is
        period (50 ms);
        functions (
                when 0 => (F3; F4);
                when 1 => (F3));
end;
```

Figure 6: Multithreading architecture described by a DSL

On the example of Figure 6:

- A major frame of 100 ms has been defined
- F1 and F2 are executed at 10 Hz (on the thread T1)
- F3 is executed at 20 Hz (on the thread T2)
- F4 is executed at 10 Hz (one cycle among two on the thread T2)

This real-time design is an extension of the synchronous language approach (on which the functional modelling described section 2 is based). The DSL describing the functional view (section 2) and the one describing the real-time design (section 5) rely thus on the same paradigm and are thus naturally compatible.

5. Automatic code generation

The multithreading design of the software (the definition of threads and the mapping of functions on the threads) remains today a manual activity which is formalised by a dedicated model using the DSL. All the remaining coding activities related to this multithreading design have been automated in an in-house tool generating Ada 2012 code from the SysML model and the DSL.

This automatic generation of code relies on a generic reusable library implementing:

- A scheduler of threads and of applicative software elementary blocks,

- An interpreter of a mission management (described by finite state machines and the DSL shown Figure 3).

Considering this generic reusable library, the following artefacts are automatically generated by the in-house tool:

- Configuration tables in Ada 2012 for the generic reusable library from the SysML finite state machine and the DSL describing the functional sequences (Figure 3) and for the definition of threads (Figure 6).
- Ada 2012 code for the skeleton of the functions (Figure 1) and the scheduling of these functions (Figure 6).

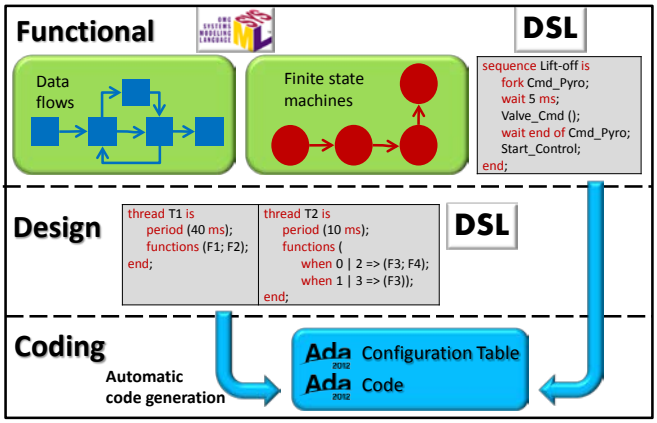

Figure 7: Code and configuration table automatic generation

There is no certification for space launchers similar to certification for civil aircraft. However, the quality of generated code remains a major issue: a failure of the flight software implies generally the loss of the launcher and of its payload (the satellites to be put in orbit) and in the worst possible case the destruction of the launch pad. The ECSS-E-ST-40C ([7]), the standard applicable to the development of space software in Europe, requires the classical reviews (of specification, of design, of code…) and test activities. The in-house code generator being not qualified according to this standard, reaching the same level of quality than for a manual code forbids suppressing any of these V&V activities. Reviews of code are for instance performed on the generated code.

The obligation of performing such reviews is often considered as a reason for not using automatic code generation because (1) the generated code is not enough readable to perform such review and (2) even a light modification in the source model may imply huge modifications in the generated code which makes mandatory a complete review of the generated code for each evolution. These two aspects were major requirements for the development of the in-house code generator: (1) the generated code is strictly equivalent of a manually written code and (2) the impact of a local model modification has a local modification on the generated code (allowing performing efficient comparisons between two versions of generated code and thus simplifying the code review).

The long term availability of the specific SysML modelling tool remains also an issue which has been solved in two ways: First, the notions of data-flows diagrams and of finite state machines have been also defined in the textual DSL; second, the code generator has been designed in order that the generated code is equivalent to a manual code, meaning that it can be easily manually maintained. During the development phase of a project, the SysML modelling tool will thus be used to decrease the development cost. After some years (5 to 10 years), i.e. during the maintenance phase, this tool will be potentially not any more maintained by the tool provider; the code will then be automatically generated from the textual DSL generated from the SysML model. If it is decided to not maintain any more the in-house code generator, it will still be possible to maintain manually the generated code.

The Figure 8 shows this three phases of maintenance:

1. The SysML model and the DSL are both maintained. The code is automatically generated. A DSL corresponding to the SysML model is generated for backup.
2. The SysML model is not maintained any more. The initial DSL and the DSL previously generated from SysML are maintained. The code is automatically generated.
3. The code generator is not maintained any more. The software code is manually maintained.

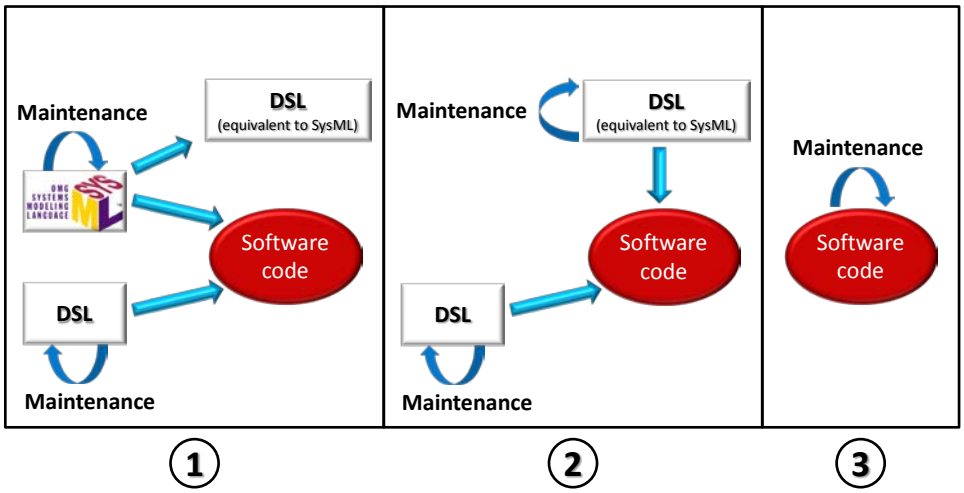

Figure 8: The three phases of maintenance

6. Synthesis

The following table summarizes the advantages and drawbacks of different modelling approach.

Approach	Functional view			Multithreading view	Safety	Long term maintenance
	Finite State Machine	Data flow	Functional sequences			
SCADE	Yes	Yes	At a low level of abstraction with finite state machine	No	Certified code generator	Rely on a proprietary tool
AADL	Yes but textual	More service oriented	No	Yes, but too complex for an extended RMS approach	No certified code generator	Standard
MARTE	See UML	See UML	See UML	Yes, but too complex for an extended RMS approach	No certified code generator	Standard
SysML (alone)	Yes but asynchronous	Yes (IBD)	Yes, but not formal enough	No	No certified code generator	Standard
UML	Yes, but asynchronous	No	Yes, but not formal enough	See MARTE	No certified code generator	Standard
SysML + DSL	Yes in SysML, with an adapted synchronous semantics	Yes in SysML (IBD) with an adapted synchronous and multithreading semantics	Yes in the DSL with the required synchronous semantics	Yes in the DSL, with an extended RMS approach	No certified code generator, but the automatically generated code is strictly equivalent to a manually written code	Standard SysML In-house tool The generated code can be manually maintained

7. Conclusion

Refining a system model to a software model and then to code remains an issue for critical system with a long duration: the tools are not maintained for the whole duration of the project and have generally a too large scope to generate code which can be manually maintained.

This paper has presented a solution taking benefit of the available modelling tools during the development, able to go smoothly from system functional definition to the code. This approach relies on the use of COTS (Commercial Of The Shelf) associated to DSL (Domain Specific Language) and in-house tools:

- The COTS tools are used to take benefit from mature and already widely deployed technologies
- DSL and in-house tools are used to take into account the specificities of the field

The quality and the long term maintenance issues are tackled by a specific effort on the code generator: the generated code is strictly equivalent to a manual code, can be reviewed and manually maintained.

The next step of improvement will be the harmonization of toolsets between the functional analysis (for instance with the use of the MEGA tool [10]), the system functional definition and the software (for instance with the Rhapsody tool [14]) and the physical architecture (for instance with Capella ([6]).

8. References

[1] AADL standard (www.aadl.info)

[2] “From Ada 83 to Ada 2012”, Philippe Gast and David Lesens (Invited presentation at Ada Europe 2015)

[3] Ada Reference Manual (ISO/IEC 8652:2012(E))

[4] Ariane 5 program (http://www.arianespace.com/launch-services-ariane5/ariane-5-intro.asp)

[5] Ariane 6 program (http://www.airbusafran-launchers.com/home/#ariane)

[6] Capella tool (https://www.polarsys.org/capella)

[7] European Cooperation for Space Standardization (ECSS, http://www.ecss.nl/)

[8] Lustre (http://www-verimag.imag.fr/Lustre-V6.html)

[9] MARTE standard (www.omg.org/spec/MARTE)

[10] MEGA tool (www.mega.com)

[11] MPCV program (http://www.nasa.gov/exploration/systems/mpcv/index.html)

[12] Standard ECSS-E-ST-70-01C (“Spacecraft on-board control procedures”)

[13] Ravenscar profile, see Ada Reference Manual [3]

[14] Rhapsody tool (http://www-03.ibm.com/software/products/en/ratirhap)

[15] Rate Monotonic Scheduling (Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment", Journal of the ACM 20)

[16] SCADE Suite tool (www.esterel-technologies.com/products/scade-suite)

[17] Signal (http://www.irisa.fr/espresso/home_html)

[18] SysML standard (www.omg.org/spec/SysML/1.3)

[19] UML standard (www.omg.org/spec/UML)