Toward model-based engineering for space embedded systems and software

J-P Blanquart, A Rossignol, D. Thomas

To cite this version:

J-P Blanquart, A Rossignol, D. Thomas. Toward model-based engineering for space embedded systems and software. Conference ERTS'06, Jan 2006, Toulouse, France. hal-02270353

HAL Id: hal-02270353

https://hal.archives-ouvertes.fr/hal-02270353

Submitted on 24 Aug 2019

Toward model-based engineering for space embedded systems and software

J-P. Blanquart[1,*], A. Rossignol[1], D. Thomas[2]

1: EADS Astrium, Toulouse (France)
2: LESIA-INSA, Toulouse (France)

*: Contact author: jean-paul.blanquart@astrium.eads.net, EADS Astrium, 31 avenue des cosmonautes, F-31402 Toulouse Cedex 4, France

Abstract:
Embedded systems development suffers from difficulties to reach cost, delay and safety requirements. The continuous increase of system complexity requires a corresponding increase in the capability of design fault-free systems.
Model-based engineering aims to make complexity management easier with the construction of a virtual representation of systems enabling early prediction of behaviour and performance. In this context, Space industry has specific needs to deal with remote systems that can not be maintained on ground. In such systems, fault management includes complex detection, localisation and recovery automatic procedures that can not be performed without confidence on safety.
In this way, only simulation and formal proofs can support the validation of all the possible configurations. Thus, formal description of both functional and non-functional properties with temporal logic formulae is expected to analyse and to early predict system characteristics at execution.
This paper is based on various studies and experiences that are carried out in space domain on the support provided by model-based engineering in terms of:

- support to needs capture and requirements analysis,
- support to design,
- support to early verification and validation,
- down to automatic generation of code.

Keywords: systems engineering, software engineering, modelling, automatic code generation

1. Introduction

Future space system projects are characterised by stringent needs in safety, autonomy and in the capability to include more and more functionalities. In these computer-based systems, the embedded software, which constituted only a small part of the global system a few years ago, has become a key element to meet all the needs. Facing to the ensuing complexity, quality and safety requirements are getting difficult to manage and usually lead to cost overrun and delays in development.
Indeed, the new place of software at system level was not anticipated soon enough. System engineering aims at supporting the transformation of operational needs into a solution achieving a trade-off between requirements, technological possibilities (including reusable components) and cost and delay constraints. However it is still a strongly empirical activity:

- it consists of paper documents that cannot be checked with confidence,
- it does not guarantee a full traceability, for instance, it is hard to know if the design fulfils all the requirements with consistency,
- it includes an expensive phase of handmade coding, integration and tests, supporting only little reuse.

It is therefore not entirely surprising that operation failure reports reveal more and more design errors. In the intention of detecting this type of errors early in the development cycle, new means (e.g. languages, methods and tools) are considered, evaluated and carried out in several practical studies in co-operation with operational units.
The purpose of this paper is to make an overview of model-based contribution in support to embedded system engineering, based on various studies and experiences that are carried out in space domain:

- ASSERT (Automated proof-based System and Software Engineering for Real-Time applications) (EC IST FP6 Integrated Project)
- FAM (Frameworks, Architectures and Models) (EADS Astrium R&D internal study)
- SCAO GEN (automatic code generation from a reference architecture with building blocks in Matlab/Simulink®) (EADS Astrium R&D internal study)
- FDIR Autonomy (CNES R&D study)
- System and software co-engineering (CNES R&D study)
- R&D French Competitivity pole initiative for embedded systems (ISAURE Program with TOPCASED project)

This paper is organised as follows: Section 2 is first devoted to a general overview of selected principles, standards and tools. Then we split results and achievements upon two themes:

- Section 3 about control engineering and code generation from continuous models, from needs capture to automatic code generation on an AOCS functions subset using Matlab/Simulink® models;
- Section 4 about computer-based system engineering (hardware and software): requirements analysis and design and validation for FDIR, and provable system and software engineering.

Some perspectives and conclusion are finally discussed in section 5.

2. Principles, standards and tools

2.1 Embedded system design

An embedded system typically consists in a collection of hardware and software elements controlling a process. For instance, the avionics system of a satellite is in charge of flight, navigation and radio communications. It requires fine control application with real-time software bound to a specific execution platform. These three domains of knowledge (control, software, hardware) are characterized by severe constraints with efficiency and dependability requirements to be verified.

Control design: Control applications usually transform physical parameters. Formal models are based on continuous (transfer) functions using differential equations to model behaviours. In space systems, the overall survival and performances depend on control applications (e.g. AOCS) but validation tests are limited because space environmental conditions are difficult to recreate. Then, many efforts are made on studies and simulations that can be performed on tools such Matlab/Simulink® ([1]) or Scilab/Scicos®.

Software design: In the last decades, formal methods and languages were proposed to specify, document and whenever possible validate (real-time) software. At specification phase, B or Z approaches ([2], [3]), for instance, apply mathematical concepts (logic and assertions) to model and verify the system. A more practical way for specification is based on scenarios using Message Sequence Charts (MSC) included in SDL [4] and UML standards. Then behavioural description languages for reactive are mainly divided in two approaches.

- Imperative languages (e.g. ESTEREL), process algebras (e.g. MEISE, SCCS) or languages based on data flows (e.g. LUSTRE, SIGNAL) correspond to a synchronous approach.
- Communicating automata (e.g. ESTELLE, PROMELA), process algebras (e.g. CSP, LOTOS), Petri Nets and their specific timed extensions can be used for asynchronous systems.

The precision of formal methods can be exploited by tools for simulation, tests or verification in support to validation. However, in the industry, formal languages were dismissed in favour of semi-formal languages (mainly model-based) which are easier to integrate in current processes. As an example, UML is intensively used for software development. On the other hand, there are no specific UML language elements for expressing explicit parallelism and we observe a strong lack of semantics that can imply incoherency between diagrams. AADL ([5]) is then seriously considered by the embedded systems community. It can represent tasks, time of execution, deadlines, scheduling, and useful information for specific real-time analysis. Indeed, in addition to real-time constraints, embedded systems are characterized by further non functional requirements such as memory occupation, electric consumption and tight relation with hardware (e.g. specific executive platform).

Embedded hardware design: Programmable hardware (e.g. ASIC, FPGA) are more and more used to integrate functions on hardware (e.g. smart sensors or actuators). Languages for describing hardware, such as VHDL and Prolog, are called hardware description languages. They integrate specific means to describe concurrency and wired communications based on architecture which make them far from common software languages.

System design: The coupling of subsystems from different modelling domains implies the necessity to manage complexity at system level. In complex embedded systems, both hardware and software must be taken into account and tests and verification have to be done before implementation and integration. Then, new formalisms are required to support this multi-domain approach. For example, SystemC ([6]) enables design, simulation and verification of functions, and their implementation on hardware or software without re-engineering. In the same way, AADL and UML2.0 seem to have the ability to manage more than only software components as for Matlab/Simulink/Stateflow® which bridges discrete and continuous domains. Requirements analysis, modelling and traceability between several domains are also objectives of this kind of new languages. Our last consideration is for the incoming SysML ([7]) that should enable the specification, analysis, design, verification and validation of a broad range of systems including hardware, software or mechanical parts.

2.2 Selected technologies for space embedded systems engineering

Model-based: Satellites are complex remote systems that must be safe from the first (and only) flight. These systems can be hardly tested in real conditions (space environment) before operational launch. In this way, the needs essentially concerns fine simulation (verification), formal proof (validation) and complexity management from requirements analysis down to final integration.

The principle we are following in order to fulfil these requirements is Model Driven Engineering. Models are used to build a virtual representation (logical solution) of the system to enable early prediction of behaviour and performance before teams proceed to implement designs.

The Model Driven Architecture (MDA) standard [8] is originally an approach to use models in software development. It aims to implement a system by transformations from a Platform Independent Model (PIM) that concerns only domain aspects (logical solution) and enables a greater understanding of the user and system requirements. A transformation uses a Platform Dependent Model (PDM) that consists of chosen technical components (Physical solution) to implement the system. This transformation also requires a mapping that associates elements from PIM to elements from PDM. The result is called the Platform Specific Model (PSM). Separation of domain and technical aspects may improve overall clarity, portability, interoperability and reusability. Moreover, using different viewpoints, corresponding to different levels of abstraction for PDM, encourages iterative construction and iterative verification.

Our studies aim to adapt processes and languages to embedded system engineering so as to benefit from this kind of approach. MATLAB®, UML2, AADL and SystemC languages were first selected to support a new system process. SysML will surely act as a new candidate for high level analysis.

- Matlab/Simulink®: Despite other alternatives are proposed, Mathworks tools are likely becoming de facto standards for control application development. However, there are still problems to implement control algorithms because transition from continuous to discrete can alter the function. SCAOGEN study relates to automatic generation of code from MATLAB models (cf. section 3).

- UML2, AADL: As for Matlab, UML2 is a widely known standard with powerful associated tools. Several studies on critical real-time embedded systems also promote the use of AADL which has become an international standard since November 2004. Indeed, with embedded systems we must consider behaviour at execution, that is to say how software is implemented on execution platform and AADL allows these implementation choices to be described. Semi-formal and graphical methods were preferred because formal methods seem too specialised and cannot satisfy system level requirements. Selected languages give intuitive notation which makes communication between analysts and customers easier. Moreover, UML lack of semantics and real-time elements can now be avoided by the introduction of profiles (e.g. OMG/SPT [9]) and design patterns. Indeed, UML2 authorises customization with stereotypes and tagged-values to easily adapt specific domain aspects. For example, additional information from formal models can be associated to system model by annotations (tagged values). These elements should be used to generate (by transformation) implementation code or simulation and analysis models for specific tools. AADL presents the same concepts using annexes and user properties. ASSERT project and FDIR study (cf. section 4) are practical overviews of information processing systems engineering using AADL and UML2. Commercial I-Logix Rhapsody tool for UML and OpenSource OSATE Eclipse plug-in for AADL are used in our current studies.

- SystemC: HW/SW interfaces must be considered for errors avoidance. SystemC is based on ANSI-C and uses additional libraries to manage software /hardware programming and co-simulation. The HW/SW CODESIGN study aims to introduce a new system/software/hardware co-design process. Tools from Mathworks®, Magilem®, Celoxica® and Nepsis® are first considered.

3. From needs capture to automatic code generation on an AOCS functions subset using Matlab/Simulink® models

Up to now for each new satellite, Attitude and Orbit Control System (AOCS) software is mainly developed from scratch with few reuse at functional and software levels. Technologies and domain analysis are now ready to improve this situation by allowing efficient design and components reuse of AOCS software.

After defining and implementing an efficient co-engineering and co-validation process between AOCS studies team and AOCS software team, EADS Astrium intends to go further. We will present how EADS Astrium prepares implementation of such an enhanced development process based on the following approach and technologies:

- A common reference process both for AOCS and software engineering teams able to support needs capture, requirements definition, design analysis, code generation, functional simulation, functional and software verification.

- Reference AOCS software architecture definition: through development experiences on a large set of operational or on-going projects (Earth observation, telecommunication, scientific, deep space probes), we are now able to define a generic and robust architecture to implement AOCS needs for many new programs.
- Matlab/Simulink® models definition and use as a common description of AOCS functional blocks, AOCS software requirements, source for on-board software automatic code generation and support of a functional validation test bench.
- Based on previous projects, selection and definition of some first AOCS software building-blocks which are good candidates to be reused as a generic component for new projects.

This on-going preparation and the first following results are mainly supported by an EADS Astrium internal study (SCAO GEN) with software, AOCS and simulators teams on a case study application with CNES satellite Pleiades AOCS. This case study will be part of an ESA external study (ACG) with others European space partners aiming at assessment and introduction of Automatic Code Generation ins Space software development.

3.1 Stakes and objectives

The different stakes of introducing Matlab/Simulink® models and automatic code generation in AOCS software development can be summarised in:

➢ Automatic code generation (ACG) from models:

- To avoid multiple AOCS descriptions: for modelling (language Matlab/Simulink®), for requirements (language natural text), for software (language C or Ada),
- Common language to describe an AOCS software product in an environment enabling global simulation (executable specification),
- Reducing evolutions implementation life-cycle time,
- Less software coding effort,
- Less software verifications (Coding standard rules could be supported by tool),
- Reducing manual coding faults by automatisation,
- Potential reduction of software tests effort.

➢ Development and reuse of AOCS components:

- Plug and play : to reduce/avoid analysis and development of already existing building-blocks,
- To focus on function use context validation than on function itself.

To assess the interests and impacts of introducing this new technology in the existing process, the following main objectives of this study are:

- Demonstrate the feasibility and the interest of the code generation from Matlab/Simulink® models,
- Evaluate and determine the best tools for embedded software (RTW embedded coder®, Targetlink®),
- Identify to which kind of AOCS functions the code generation is applicable (algorithms, automata, FDIR, …),
- Analyse the consequences of code generation on modelling, simulations, documentation,
- Define a common process between AOCS studies and software teams, allowing code generation from functional simulation models
- Analyse the feasibility of defining reusable building blocks
- Warrant the feasibility and availability of a very representative functional simulator (HIFI simulator) with flight software in-the-loop and based on Matlab/Simulink® models.

3.2 Process

From prototype model to ACG model: ACG has a great impact on the AOCS studies development process. Right now modelling in the Matlab/Simulink® environment was considered has a prototyping activity performed during the preliminary A/B phases. So, no specific rule or standard was written. The main effort was brought on the C phase simulator which is a high fidelity simulator including the actual AOCS software. The AOCS software was written from AOCS specification, itself partially written from preliminary Matlab/Simulink® models. Once the HIFI simulator is available, the Matlab/Simulink® models are no more really maintained.

With ACG, the reference becomes the models themselves. So many constraints appear. The models must permit not only the auto-coding but they have to be readable and maintainable. These constraints make the modelling activity heavier, which is no more compatible of fast prototyping approach used during preliminary A/B phases. This leads to conclude that two kinds of modelling can be managed at AOCS studies level:

- Fast prototyping modelling approach similar to the current usage of modelling, maybe lightly improved by introducing general rules favouring the readability.

- ACG modelling approach that has to account strict rules allowing determinism of execution and code generation.

The prototyping model is not supposed to be maintained and used to initialise the final model.

The proposed new process is illustrated in the following Figure 1 and its main steps described further below the figure.

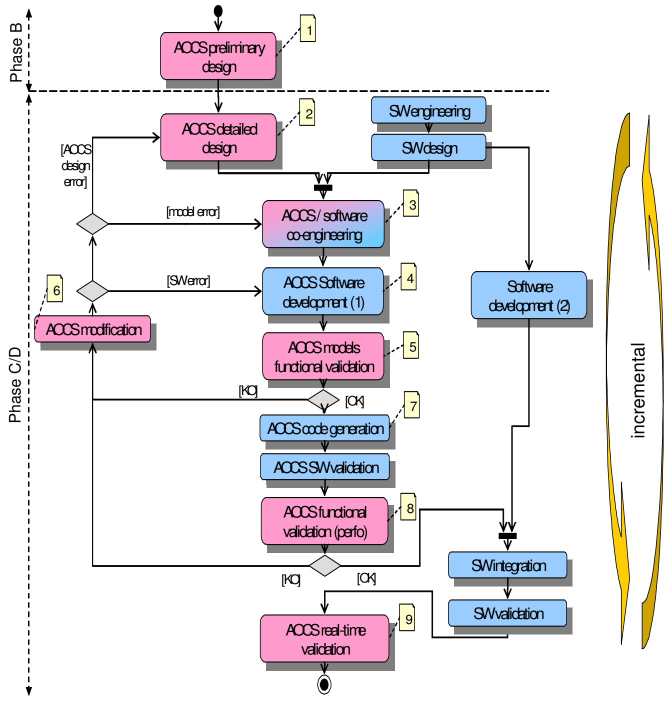

Figure 1: AOCS and software new process

Initial AOCS model building: the ACG model is first developed by the AOCS team form the AOCS reference model template and the preliminary prototype model. At this stage, the reference architecture shall be respected and as far as possible all modelling rules. During this phase of AOCS design, the software team can support the AOCS team to instantiate the AOCS reference model and especially on parts that are nut usually integrated in the AOCS models like the modes management and the FDIR aspects.

This first model is tested in closed loop in the Simulink® environment by the AOCS studies team.

AOCS model co-engineering: once the AOCS concepts are considered validated and documented, the co-engineering phase between the AOCS and the software teams starts to refine the model and add all the functionalities which were not mandatory to validated the AOCS concept. During this phase, the software team checks the whole models against the modelling rules and especially the auto-code-ability. Both teams work together on the same model.

SW development: this point concerns the part of AOCS software that could not be auto-coded. The process would then be similar to the current one. The both team agree first together the software specification then the code is developed and unit tested before generating "S-function" to be inserted in the Simulink® model.

AOCS model functional validation: this phase is realised by AOCS studies teams with support of software team if debug is required. This phase is not much impacted with respect to the current process. In this phase all tests are performed on models but not yet on the generated code in order to ease the tuning and the debug of the model.

Code generation: once the model is completely validated the code generation can be performed. An "S-function" of AOCS software is generated to be inserted in the Simulink® environment. A subset of test is run to check that code generation has not introduced error.

AOCS functional validation and performances: this step remains similar to what it is in the current process. The whole performance tests campaign is run on the AOCS flight software.

3.3 Architecture

The analyses performed commonly by AOCS studies and software teams shown that it was possible to define together an AOCS reference architecture that favours, on one hand, the reuse of AOCS functions by defining building blocks and on the other hand, the code generation because it accounts elements of software architecture standard.

This AOCS reference architecture imposes new constraints for AOCS teams but also provides advantages that were already experimented at software point of view. Once the architecture is understood, it's easier to move inside the model, because all modes, all projects follow the same standards. This makes the models more readable and eases the maintainability of the models.

It also eases the reuse of building blocks, because building blocks are thought to be inserted in this kind of architecture. So when a new project respects the reference architecture, it is then possible to reuse existing building blocks.

3.4 Modelling rules and code generation issues

Coders' performances: code generation on Simulink® models gives quite good results if accounting few constraints in the definition of the models.

Code generation from StateFlow® models is less efficient than for Simulink but remains acceptable.

Code generation from Matlab® models is still problematic: generated code is not easy to read and is quite inefficient especially in matrices handling. The analysis shall continue to define if it is possible according to the following criteria:

- Ratio of Matlab® algorithm with respect to the rest of AOCS;
- Capability to translate Matlab® models in Simulink® models;
- Acceptability of AOCS software performance degradation according to current margins;
- Capability of improvement of the Mathworks Matlab® code generator.

<u>Code generation applicability:</u> code generation is applicable to the whole AOCS model and it is possible to complete the AOCS model in order to generate the whole AOCS software from the model. This induces to add at model level, stubs of the interface of the AOCS software with the data handling software layers and components (Telecommand, Telemetry, monitoring and Input / Output).

<u>Modelling standards and rules:</u> quality of generated code depends directly on the quality of the models. So standards or rules previously applicable to software development become new standards or rules applicable to models development. They are of two kinds: standards and rules for models writing to ease the code generation configuration and ensure generated code readability; and standards and rules for models architecture which ensure models (and software) readability and maintainability.

In order to benefit of the software experience in terms of software architecture that accounts not only AOCS needs but also operability and maintainability needs, it is foreseen to apply to models development the same approach that the one currently applied on software development. That is, an AOCS reference models is defined that accounts all constraints either AOCS, or FDIR or operability.

Then, this reference models defines the framework for any AOCS models development. It defines the backbone of the model and the software architecture. It defines:

- Standard model breakdown
- Code generation configuration
- Standard interfaces
- Standard dynamic behaviour
- Standard activities scheduling profile

<u>Models sharing:</u> with the ACG approach the models become the reference from which the code is generated. That means that the models have to be complete, in terms of which functionalities will have to be in the final software. They also have to include all information necessary for code generation. This means that the models have to be shared by the AOCS studies team and the software team. The first one creates the models from the AOCS models framework. The second one completes them with code generation information, eventually updates them to permit auto-code-ability and add missing functionalities. The AOCS and the software teams shall then be able to share the same model to make it evolve.

3.5 Tools evaluation

Among tools available for code generation from Matlab/Simulink® models (Mathworks®, d-SPACE® and Esterel SCADE®), two tools were selected for evaluation:

- The Real Time Workshop Embedded Coder (RTW-EC®) from the Mathworks®
- The TargetLink® 2.0 from d-SPACE®

RTW/EC® seems yet less mature and intuitive than TargetLink®, but it accepts all Simulink® blocks and already integrates interesting features (like bus and embedded Matlab® function) that, even not yet completely operational, are very promising for targeted kind of model. It seems therefore providing a better capability of improvement in the short-term and will be used at least for the study. This choice will have to be confirmed at the end of the study.

4. Model-based engineering for embedded systems

The development of embedded systems, in particular for critical applications, calls for methods to master their complexity and the many interactions between their components and interrelations between hardware and software.

EADS Astrium has engaged a set of studies, pilot projects and experiments investigating the potential benefits of model-based approaches for the development and validation of embedded systems. Among them, two studies are highlighted in this section, addressing on the one hand model-based approaches for the development of embedded fault and error management functions, and on the other hand model-based approaches as a support to the provable development of embedded hardware-software systems.

4.1 Embedded fault management functions (FDIR)

The dependability of space systems is ensured (in addition to a set of fault prevention and robust design approaches) by specific on-board functions dedicated to the detection and diagnosis of faults and the selection of appropriate recovery or reconfiguration actions. These functions constitute globally what is called FDIR in space systems (for Fault Detection, Identification and Recovery).

The FDIR function must be able to maintain (or restore) the best achievable level of mission in case of adverse events such as faults or failures of on-board components. For advanced autonomous and complex space systems, the FDIR function must react to a very large number of potential events and combinations of events, taking into account the fact that not all events may be fully covered and that an appropriate balance must be found between the need to maintain or restore the mission (availability) and the need to maintain the ultimate survival capabilities of the spacecraft (safety).

Space systems FDIR is therefore designed as a complex architecture, generally highly hierarchical, of detection and reaction mechanisms whose many explicit or implicit (based e.g., on different reaction time) interactions make FDIR development and validation particularly difficult.

In a recent study [10] we investigated model-based approaches for the development and validation of FDIR for advanced autonomous space systems.

For this study, AADL appeared as particularly interesting, thanks to its capability to model the actions of the FDIR function on the very configuration of the physical architecture (through the definition of modes, each associated to a particular configuration of physical components and a particular mapping of functions and software components).

We also aimed at defining a modular progressive approach for the construction of complex FDIR functions, both to support complexity mastering and to provide enough flexibility to further adapt or update the FDIR functions during late development or even in-orbit maintenance.

We therefore elaborated a library of elementary "FDIR components" that can be progressively composed into a complex FDIR function.

This approach also exploits the capability of AADL to model both functional elements and components of the physical architecture and model their relationships and mapping. Similarly, we defined elementary components both for FDIR strategy (e.g., timeout, re-try, switch nominal and backup) and for FDIR architecture (e.g., hot duplex, cold duplex, majority voting).

As illustrated in Figure 2, this approach allows a progressive construction and validation of a complete FDIR to be performed by composition of architectural elementary components on the one hand, and of procedural elementary components on the other hand (defining the global strategy to exploit the various reconfiguration possibilities provided by the available architecture).

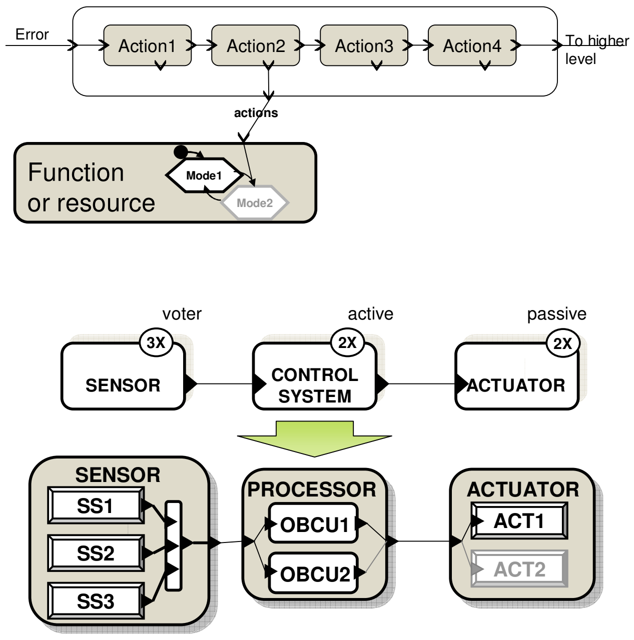

Figure 2: Progressive model-based FDIR construction approach

This Figure 2 above illustrates the composition of elementary FDIR procedures, mapped on functions or physical resources, into a strategy (topmost part), and the hierarchical composition of the dependability architecture with elementary building blocks (bottom).

However we also aimed at providing a powerful support to the validation of complex FDIR functions, including for instance the verification of the correctness of the various possible sequences of events and reconfiguration actions or the estimation of the reconfiguration duration.

Though there may not be "built-in" obstacles to perform such analyses on an AADL model, the current status of its definition and of its support tools does not provide the support we needed in particular terms of detailed behavioural description and analysis.

We then complemented the AADL models with UML models of the detailed behaviour, especially the transient behaviour, on event occurrence (fault), from one configuration to another. As illustrated in Figure 3, the models can be used to support various analyses or simulation (on automatically generated code).

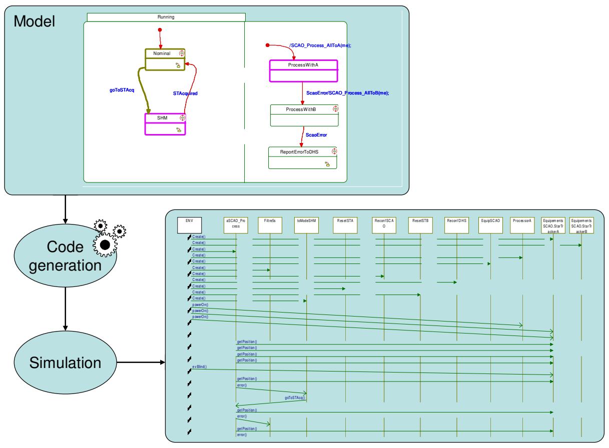

Figure 3: Model-based behavioural analysis and simulation

4.2 Provable engineering of embedded systems

Considering the complexity, criticality and also cost of embedded systems, there is a strong need of development and validation approaches capable to provide the desired level of confidence in a reasonable amount of cost and time. EADS Astrium has engaged studies towards this aim and in particular through its participation to the ASSERT project[1].

The ASSERT project aims at improving the system and software development processes for critical embedded real-time systems in the aerospace domain by:

- Identifying and developing proven critical system families' architecture, using a proof-based development process supported by formal notations, component models, processes and tools;
- Developing associated building blocks that can be composed, tailored and verified in open frameworks.

Among other activities, modelling tasks cover the elaboration of models of system families and models of their components to be instantiated and composed as building blocks of a given target system. Three modelling levels are considered: functional, software and physical architecture. This is illustrated in Figure 4 on a space system with a subset of applications (GNC, Guidance and Navigation Control, TMTC, Telemetry and Telecommand and Thermal control) mapped on an architecture provided for illustration, characterised by the existence of specific processors dedicated to highly critical functions (which could be achieved with appropriately managed criticality levels within a processor, as also investigated in the study).

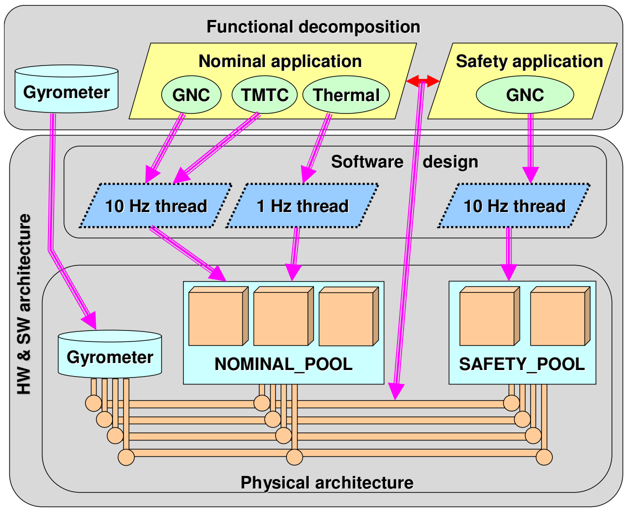

Figure 4: Three levels of modelling (example of architecture)

In the final process elaborated by the ASSERT project, the model of a given system will be defined as an adaptation of the system family model, by composition of properly instantiated or tailored building blocks, with provable composability properties.

These properties, and their proofs, address mainly real-time properties and assumptions (synchrony or asynchrony characteristics, deadlines, worst-case execution time etc.) and dependability properties (fault assumptions, failure modes, properties such as atomicity etc.).

Consequently the ASSERT modelling framework, based on AADL and UML, will include a set of extensions proposed to AADL, to cover the formal expression of all the properties defined in the ASSERT process, to enable their verification by support tools, and to support the generation of system glue code with predicted performance attributes. This extended AADL will be implemented through an UML profile ensuring continuity of the modelling, proofs and code generation chain, from system level down to software level.

[1] ASSERT: Automated proof based System and Software Engineering for Real-Time applications, Integrated Project (n°4033) of the 6th Framework Programme, Information Society Technology, European Commission. The ASSERT project has started in September 2004 for 3 years, with 29 academic and industrial partners under the coordination of the European Space Agency, ESA.

5. Conclusion

After very good results on using Matlab/Simulink® models and automatic code generation for rapid prototyping, the same technology is now explored for operational software provided that:

- It implements a well defined process;
- Modelling is compliant to a reference architecture and some design rules;
- The Code Generator is customisable and produces source code that is efficient, easy to integrate and with the right level of quality.

Concerning system and software modelling, methods and languages such as AADL and UML, present some important and interesting characteristics. Among them, we can mention the AADL capability to formally define the links between the software (or functional) elements and their execution support, and the capability to identify how a function is implemented and executed. It then supports the system development and validation, at least theoretically, in particular through detection (or prevention) of mapping errors and through analyses (e.g., data or control flow, real-time properties or partitioning properties). In addition, AADL provides a natural support to the modelling of functions which act on the configuration of the physical architecture, which is particularly well suited for FDIR functions modelling.

However AADL is still limited, at least in its current definition, in terms of detailed behavioural description, and moreover is still missing powerful support tools.

UML provides useful features in terms of e.g., high-level modelling, and behavioural modelling and verification. However the semantics does not fully support the consistency between the different diagrams (though this may be solved through additional profiles).

AADL and UML finally provide complementary solutions for the development and validation of complex embedded systems and software, which strongly calls for the definition and implementation of gateways between them.

These preliminary results show that all these new methods and languages support at least partially the system and software engineering needs and that, to a large extent they are very promising in terms of definition, verification and validation. However the selection and utilisation of these different formalisms, and languages still necessitate:

- To complete for each of them a detailed assessment;
- To integrate them into a well defined process;
- To be supported by a rigorous method usable within an operational industrial domain.

6. Acknowledgement

The authors acknowledge the contribution of their colleagues to this work and in particular Pierre Yves Farges and Michel Horblin (EADS Astrium).

The authors wish also to thank CNES and the European Commission, which funded all or part of the studies supporting the results and discussions presented in this paper.

7. References

[1] Matlab/Simulink, http://www.mathworks.com

[2] ISO: Information technology "*Z formal specification notation*" syntax, type system and semantics (2002) ISO/IEC 13568:2002, International Standard.

[3] J.M. Spivey, "*The Z Notation: A Reference Manual*", 2nd edition, Prentice Hall (1992). Available at http://spivey.oriel.ox.ac.uk/~mike/zrm/.

[4] SDL Forum Society, http://www.sdl-forum.org

[5] Society of Automotive Engineers (SAE) Aerospace Avionics Systems Division: "*Architecture Analysis & Design Language Standards Document*", version 1.0, Nov. 2004.

[6] SystemC Community, http://www.systemc.org

[7] SysML Forum Society, http://www.sysml.org

[8] Object Management Group (OMG): "*Model Driven Architecture Guide*", version 1.0.1, June 2003.

[9] Object Management Group (OMG): "*UML Profile for Schedulability, Performance, and Time Specification*" OMG document formal/05-01-02, Jan. 2005.

[10] JP. Blanquart, P. Grandjean, M. Horblin, P. Pleczon and D. Thomas: "*Rapport d'étude: Architecture FDIR pour satellite autonome*", Edition 1.0, September 20, 2005, reference EADS Astrium EF.NT.MH05.00131, CNES Study DCT/SA/AB n°04-2240. (In French)

[11] D. Lesens (coordinator): "*MA3S Reference Architecture Preliminary Definition Level 1*", ASSERT project (IST FP6 Integrated Project 4033), deliverable 004033_MA3S_EADS.DVRB.02, 2005.

8. Glossary

ACG	Automatic Code Generation
AOCS	Attitude and Orbit Control System
ASIC	Application Specific Integrated Circuit
CNES	Centre National d'Etudes Spatiales
FDIR	Fault Detection, Identification and Recovery
FPGA	Field Programmable Gates Arrays
HIFI Simulator:	High-Fidelity simulator with flight software in-the-loop
SoC	System on Chip