SPACE ON-BOARD SOFTWARE REFERENCE ARCHITECTURE

SAVOIR-FAIRE working group (Jean-Loup Terraillon (*ESA*), Andreas Jung (*ESA*), Paul Arberet (*CNES*), Sergio Montenegro (*DLR*), Alain Rossignol (*Astrium*), Gérald Garcia (*TAS*), Jianning Li (*SSC*), Ana Isabel Rodriguez (*GMV*), Silvia Mazzini (*Intecs*), Poul Hougaard (*Terma*), Stuart Fowell (*SciSys*), Massimo Ferraguto (*SSF*)) and Marco Panunzio (*University of Padua*)

Abstract: The paper aims at describing the motivation and the outcome of the definition of a reference architecture for software on-board the spacecraft platforms. It is based on the work of an industrial working group named Savoir-Faire, supported by industrial activities, which have defined the architectural principles and the process to define further the architecture. The architecture is using a component based approach executed by an execution platform adapted to space. The paper will present the user needs, the architecture requirements, the architectural principles, and the way forward.

Keywords: COrDeT, Savoir-Faire, architecture, space on-board software, component

1. INTRODUCTION

1.1 Background

Space industry has recognized already for quite some time the need to raise the level of standardisation in the avionics system in order to increase efficiency and reduce cost and schedule in the development. As an example, Eurospace made the recommendations, at the end of 2006, in the frame of the European Harmonisation process, to develop modular avionics architectures and to standardize interface in order to define building blocks (BBs). This was confirmed by a Panel Session on Avionics Reference Architectures (ARA) took place on 4th October 2007 in the frame of ESA's workshop on Avionics Data, Control and Software Systems (ADCSS '07).

The implementation of such a vision is expected to provide benefits for all the stake-holders in the space community:

− *Customer Agencies:* reduction of the project development lifecycle and reduced technical risk for software development and operations;

− *System Integrators:* increased competitiveness in the world market through lower price and shorter time-to market; multi-supplier option;

− *Supplier industry:* diversified customer basis; supplied building-blocks compatible with prime architectures across the board.

Furthermore, the adoption of standards is a known facilitator to focused and global innovation.

Space can benefit from the example set by other industrial sectors where similar initiatives have been successfully conducted for some time already, e.g. AUTOSAR for the automotive industry. Although it might be argued that the business model is different in the automotive and space sectors, AUTOSAR demonstrates that the need for standardisation is born from the drive of industry to become more competitive, irrespectively of the sector.

There are a number of ongoing initiatives towards this vision, both on the side of industry and in the frame of ESA's R&D programmes. Space primes and on-board software companies have made significant progress and have implemented and/or are implementing reuse on the basis of company internal reference architectures and building blocks. However, in order for this standardisation to provide the maximum benefits, it has to be tackled (at least) at European level rather than at company level.

ESA has recently achieved parallel activities aimed at preparing the grounds for improving the onboard software reuse in a systematic manner (COrDeT and Domeng; see acknowledgement). The studies confirmed that interface standardization allows to efficiently composing the software on the basis of existing and mature building blocks. Much attention is being devoted today to this standardization, but the pace needs to be increased in the development, validation and adoption of the standards.

1.2 Towards reference architectures: Savoir and Savoir-Faire

The success of this strategy relies on harmonisation actions (within space industry) to define and agree on the new generic architectures, interface, building blocks and process. Reference architectures must be defined jointly with industry and the results must be shared in order to be beneficial to all spacecraft developments, since it is demonstrated by concrete industrial experience that there are significant commonalities between avionics systems across spacecraft service domains (Science, Earth Observation, Telecommunication and Navigation).

In April 2008 an initiative between European Space Agencies and Space industry at prime and supplier level was started with the objective to take stock of the ongoing initiatives in both agencies and industry towards the vision of an avionics development that maximises reuse and standardisation, to identify the gaps and to help concentrate all the efforts from industry, national agencies and ESA towards the shared objectives. To this effect it was seen as convenient to refer to all ongoing initiatives, related to this vision by the name “Space Avionics Open Interface Architecture”, in order to focus the attention on the overall goals and to facilitate the synergy between such initiatives.

To enable progress on the objectives of SAVOIR it was needed and required to set-up a platform where technical discussions can take place and where key recommendations (e.g. what standards to develop) can be made by the stake holders. To this end, an Advisory Group was established to steer the work plan of the technical discussions, referred to as SAVOIR Advisory Group.

The SAVOIR Advisory Group decided to spawn a specific subgroup on on-board software reference architectures called “SAVOIR Fair[1] Architecture and Interface Reference Elaboration working group. This subgroup has to achieve the definition of software reference *architectures*, based on open *interface* standards, for the purpose of specifying *building blocks* that can be developed, qualified and composed into compliant avionics software systems with a minimum of re-engineering effort, providing a maximum of reliability and performance and simple to use and to implement. It acts as design authority for current R&D activities, provides recommendations to future R&D activities and provides the SAVOIR Advisory Group with a synthesis of the results.

The paper gives an update on the current status of the working group activities and related R&D activities [5].

2. WHY REFERENCE ARCHITECTURE?

2.1 Motivation

The space projects schedules are always decreasing. Teams need to increase efficiency and cost-effectiveness in the development process of onboard avionics. But there is a trend towards more functionality implemented by the onboard software and more complexity for the overall space mission. Therefore, the overall objective of space industry is now to standardise the avionics systems for space programmes, and therefore the on-board software.

The proposed way to achieve this objective is by adopting a building block approach, i.e. an approach that permits to implement the on-board software from a set of pre-developed and fully compatible building blocks, plus specific adaptations and "missionisation" according to specific mission requirements. The target missions are the core ESA missions, i.e. high reliability & availability spacecraft driven systems (e.g. operational missions, science missions).

The next issue is to be able to develop the “right” building blocks, which can be produced and delivered by suppliers to any system integrator. To achieve this, reference architectures must be defined.

For software, the key to a generic/reusable architecture is to separate the application aspects from the general-purpose data processing aspects. The lower level layers of the architecture should handle the computer issues (implementation of communication, real time, dependability). The higher level layers should deal only with application aspects. However, the application building blocks (ABB) should be fully characterized with their needs in terms of communication, real-time, dependability and should rely on the platform building blocks (PBB) for the complete implementation. Automatic configuration of PBB from ABB attributes is a target.

The reference architectures are then used as a basis to develop standards for interface specifications. This enables the development by industry of building blocks, allowing the implementation of the therefore famous AUTOSAR concept: “Cooperate on standards, compete on implementation”.

2.2 Faster, Later, Softer

The OBSW life cycle must be organised in consistency with the system life cycle that features the definition of functional increments. It must in particular:

- allow for a *faster* software development in the context of a reduced schedule, i.e. the ability to release in a short time a new version of the OBSW product and ultimately the complete and validated product,
- be compatible to a *late* definition or changes of some of its requirements, e.g. typically for mission specificities like the system FDIR, system mission management or adaptation to hardware specificities or hardware changes,
- cope with the various system integration strategies, i.e. be *flexible* enough e.g. to allow for early release to integration or cope with late central data management unit availability.

From the above given programmatic stakes, the slogan *“Faster, Later, Softer”* has been derived in the COrDeT studies and represents a summary of three stakes for the on-board software life cycle. Those stakes are included and defined as user needs.

[1] The architecture must be fair in being impartial for all the stakeholders, even-handed for the interface definition, reasonable technically, non discriminatory for technologies and adequate to the purpose.

2.3 User needs

In order to guide the work some user needs have been collected:

Shorter software development time: Future projects will require faster software development in the context of a shorter schedule. The overall schedule is reduced because: (i) the definition of software requirements is finalized later; (ii) the final version of the software is expected to be released earlier. The cost of the SW itself is a minor fraction of the cost of the whole system, but the impact of delays in availability of the SW may have big impact on the overall schedule and consequently also on cost of the project.

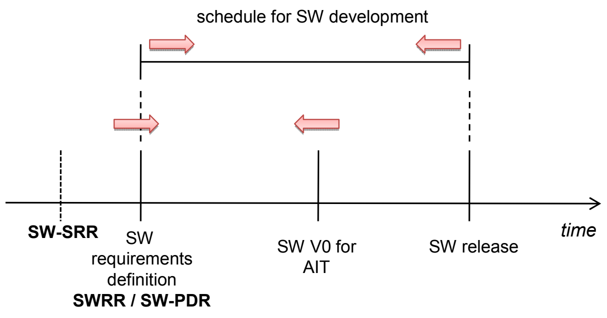

Figure 1: Software schedule issue

Reduce recurring costs: Identification and reduction of recurring costs arguably helps to either deliver project resources to focus on the value added of the product (the functional contents) or to reduce the cost of development while providing the same set of functions. Recurring costs in this context are meant to be those parts of the software which do not directly provide an added value and are not mission specific, e.g. device drivers, real-time operating system, providing communication services, archetypal parts in the application software, etc.

Quality of the product: The level of quality of the software must be at least the same as the one of OBSW developed with current approaches (timing predictability, dependability, etc...).

Increase cost-efficiency: Cost-efficiency relates in this context to the "value" of the software product that is developed with a certain amount of budget. An increased cost-efficiency is achieved by: i) developing the same set of functions for less budget; ii) developing the same set of functions with more stringent requirements (for example, more performing and robust control laws) for the same budget; iii) increasing the number of realized functions for the same budget.

New development approaches may be required to fulfil this user need. The budget available for software development is not expected to grow and it may instead be subject to reduction. On the contrary, the performance of core functionalities is expected to grow (e.g. accuracy of the AOCS controls) and new complex functionalities are expected to be developed (i.e. formation flying, advanced autonomy, etc...)

Reduce Verification and Validation effort: The cost of V&V activities in the space domain is the main contributor to the cost of software development and may range between 50% to 70% of the overall cost; it is also one of the most time-consuming activities. New development approaches shall foster the reduction of effort for Verification and Validation. This result is achieved with the adoption of a rigorous design methodology and a suitable overall process. A possible strategy consists in the adoption of a design process inspired to the principle of *Correctness by Construction* (C-by-C), analysis at early design stage and provisions for reusability of (functional) tests are the key ingredients to the attainment of the expected reduction of effort; a reduced effort for V&V activities concurs to attaining a shorter development time and reduced cost.

Mitigate the impact of late requirement definition or change: The definition of new requirements or their change may occur during the whole SW lifecycle. The causes of those situations are typically traced to late refinement of system design, evolution of the operational level, to mission-specific concerns like a late finalization of the system FDIR or of the mission management, or to software modification required to compensate for HW problems found during system integration.

Support for various system integration strategies: Preliminary software releases are important to allow early system integration. Software development as well may be managed with different strategies. The new approach is required to facilitate those different strategies and ease the final integration of the increments or elements.

Simplification and harmonization of FDIR: For future missions, a simplification and hopefully harmonization of the Fault Detection, Isolation and Recovery (FDIR) approach is advocated. This need has to be attacked both at system and software level. For the former, system engineers have to rationalize the definition of the FDIR strategy. For the latter, a simplification and rationalization of the software provisions for FDIR is advocated. A valuable strategy would consist in providing a set of functionalities and design patterns that cover the essential mechanisms for the software realization of a FDIR strategy.

Optimize flight maintenance: Flight maintenance may be required to change the OBSW. Facilitation of the required operations, as well as a harmonization of the strategy to perform it will decrease the time and cost of maintenance. It would be desirable to also minimise the risk of in-flight maintenance by updating parts of the software (possibly an entire component) without having to reboot the CDMU.

Industrial policy support: In general, the development process should enable multi-team software development. Industrial policy is very specific to the ESA environment. It requires some flexibility in the allocation of software elements to industry, according to criteria such as prime/non-prime, or geographical return. Multi-team software development facilitates the subcontracting to non-prime while keeping integration controllable, and facilitates the application of the geographical return policy.

Role of software suppliers: A new harmonized approach to software development should promote the increase of competence of supplier. The approach inherently foster supplier competition: different suppliers may develop the same component and compete on quality, extension features, performance, cost and, to a lesser extent, schedule. Interestingly, for what concerns suppliers, the approach will enable to provide software to every software prime, without the need to adapt all the software to the specific development policies of each single prime.

Dissemination activities: The benefits that are earned with the adoption of an agreed development approach may be increased with the collaboration of system engineers. The definition of future systems (by system engineers) can be improved by exposing them to the core principles of the approach. If they specify out of the domain of reuse, the cost will certainly increase.

Future needs: Several needs can be foreseen for future missions. In particular the growing software complexity is one of the origins of those needs. These needs should be taken into account and should be evaluated on their impact on the software reference architecture. Examples of future needs are the integration of functions of different criticality level, of different security level, use of Time and Space Partitioning (TSP), support to the multi-core processors, contextual verification of safety properties.

2.4 High level requirements

The user needs have been then translated into a set of high level requirements, in particular about software reuse, separation of concerns, reuse of V&V tests, HW/SW independence, component-based approach, software observability, software analyzability, property preservation, integration of software building blocks, support for variability factors, late incorporation of modification in the software, provisions of mechanisms for FDIR, and software update at run time.

The requirements have been prioritized in order to guide the planning of the work. They have been traced to the user needs.

2.5 The software reference architecture

The software reference architecture is made of two main parts:

- An *software architectural concept* addressing the pure software architectural related issues,
- The *functional aspects* represented by a set of building blocks and the corresponding interface definitions, expressing the functions derived from the analysis of the functional chains of the core on-board software domain.

Reference architecture	=	**Software architectural concepts**	+	**Building blocks & Interfaces**

3. THE SOFTWARE ARCHITECTURAL CONCEPT

3.1 Component model

The *software architectural concept* is based on a component based software engineering (CBSE) approach [1]. The approach defines a *component model* that features three software entities, the *component* (which is the design entity), the *container* and the *connector* (two entities used in the implementation which do not appear explicitly in the design space). The approach permits the creation of software as a set of interconnected components. The set of components represents the functional architecture of the software. An underlying execution platform provides services to components, containers and connectors. Finally, all the software is deployed on a physical architecture (computational units, equipments, and network interconnections between them).

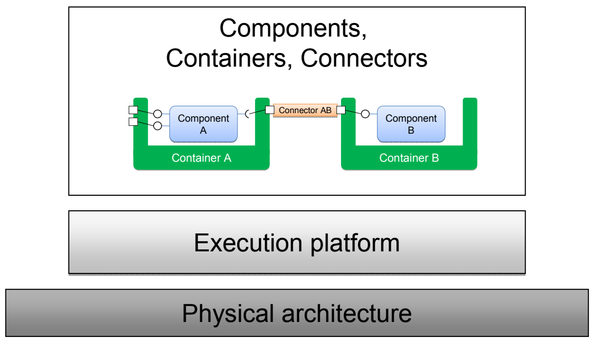

Figure 2: Software architectural concept

The success of a development methodology aimed to software reuse in the space domain (considering the specific constraints of embedded systems) lies on: (i) the adoption of a rigorous separation of concerns in particular between the functional and non-functional dimensions; (ii) the verification of properties of components and of component assemblies.

Separation of concerns is enforced in a CBSE by a clear and careful allocation of concerns to the three distinct software entities of the approach (the component, the container and the connector). Additionally the design space should be equipped with a multiple view that enforce separation of concerns and permit to the various design actors to concentrate only on their concern of pertinence.

Components are decorated with various properties. The second key factor of CBSE is the verification of properties, in particular (a) composability, which is ensured when a property stipulated on individual components hold upon component composition; and (b) compositionality, which is the ability to determine a property of a component assembly or in general of the whole software by the only use of properties of their individual components.

The component model defines formally all those functional concepts as well as the rules governing their usage. It may be noticed that ASSERT (http://www.assert-project.net/), even if focusing on other aspects, features a kind of component model.

3.2 Computational model

Likewise, the computational model defines formally computational entities as well as the rules governing their usage. Using a computational model is required by the Space software engineering standard (ECSS-E-ST-40C). A dynamic software architecture is described according to an analysable computational model, i.e. from the description a schedulability analysis can be conducted.

3.3 Execution platform

The execution platform is the part of the software architecture providing all necessary means for the implementation of a component and computational model. The services can be subdivided into four different parts: (i) services for containers (to enforce or monitor non-functional properties), (ii) services for connectors (e.g. for the implementation of communication means), (iii) services for components (typically technical services like access to the on-board time, or storage services) and (iv) services to implement so-called "abstract components" (e.g. PUS monitoring, OBCPs, HW representation).

Note: Reference architecture and Time and Space Partitioning (TSP).

The notion of reference architecture includes (i) a computational model and (ii) a conformant execution platform that provides services for containers, in particular scheduling services. It is therefore possible to select Time and Space Partitioning as a scheduling service, with a cyclic or time triggered computational model.

This choice only impacts on the services for connectors, as the communication services must also include an inter partition communication channel.

4. THE FUNCTIONAL ASPECTS

4.1 Introduction

The building blocks and interfaces, which are the *functional aspects* of the reference architecture, are the result of the mapping of functional chains of the spacecraft onto the software architectural concept, through the allocation of functions to components or execution platform services.

Building blocks & Interfaces	= mapping of	**Functional chains & Variability factors**	onto	**Software architectural concepts**

4.2 Functional chains

Functional chains are a consistent and well enclosed set of functions that are needed to use and operate the spacecraft and typically consist of a set of hardware and software that together perform an end-to-end system level function. Functional chains should be integrated and validated (as far as possible) in separation.

An example for a functional chain is the thermal control which uses the following entities: thermistors (sensors), I/O interface, control algorithm, heaters (actuators), Telecomand reception, Telemetry delivery. Other functional chains are the AOCS; Data management and control (TM/TC); Mission Management, Payload management, etc.

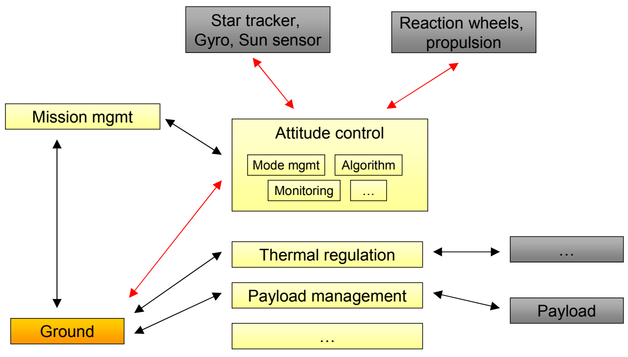

Figure 3: Example of functional chains

4.3 Domain engineering

The reference architecture is made to be reusable. The architectural concept gives the mechanisms to implement reuse. However, the functional aspects must also be reusable. To achieve this goal, the domain of reuse of the functional chains must be defined. This is usually done using the concept of domain engineering. This process consists of two main steps:

- Domain analysis that identifies the scope of the domain (the functional chains to be covered), its commonalities and its variability factors.
- Domain design that maps the functions identified in the previous steps on to the software architectural concept by allocation of functions to components or execution platform services.

4.4 Variability factors

For the development of a reference architecture and the identification of the elements which can be re-used in the frame of this architecture, an analysis of the commonalities and variability factors within the domain is essential. The domain analysis gathers all this information synthesized in the so-called feature model. Various aspects of the domain can be taken into account, e.g. functional chains (top-down approach), physical world (bottom-up approach), past, current and future architectures (software as well as hardware), experiences from other domains (e.g. automotive).

The COrDeT study identified six variability factors:

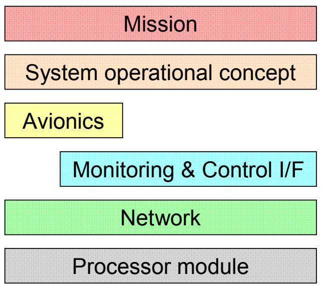

Figure 4: Variability factors

The variability factors are dependent on each others but solely in one direction (from mission to processor module). Indeed the assets implementing the mission variability will rely on assets implementing the system operational concept, avionics, network and processor module variability factors.

Setting the variability within the domain of reuse obviously impair the definition of any system out of the domain of reuse. It must be understood that a proper definition of the domain is essential, as it will trade the freedom of specification of system engineers, for the efficiency of the system implementation. The definition of the functional chains for the reference architecture needs therefore strong support of system, operation, avionics and control experts.

The mapping of the functional chains on the software architectural concept allocates functions to components and execution platform services. It must be noted that those components or execution platform services must not cross the boundaries of the identified variability factors in order to improve re-use. The result is subsequently expressed as a set of building blocks (either application components or execution platform elements) and their interfaces.

4.5 Interfaces and their definition

General and reusable interfaces must be carefully designed and group a limited number of operations which have a strong functional cohesion, according to the Interface Segregation Principle:

"Building blocks shall not be forced to depend on operations that they do not use."

This principle deals with the disadvantages of defining large interfaces. Large interfaces are not cohesive. Cohesion is a measure of how strongly related and focused the responsibilities are. In other words the interfaces of building blocks should be broken into fine grained groups of functions that have highly related responsibilities for specific clients or service. This is also in relation with the cohesion of building blocks, which must be highly cohesive. A building block with highly related responsibilities that does a little amount of work is considered highly cohesive.

5. THE REFERENCE ON-BOARD SOFTWARE ARCHITECTURE FOR SPACECRAFT

A mapping of the functional chains onto the elements of the software architectural concepts results in the identification of building blocks as follows:

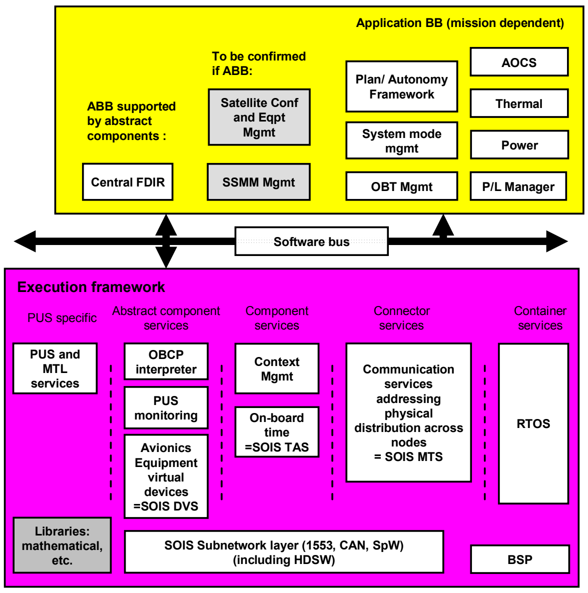
Application BB (mission dependent)
To be confirmed if ABB:
ABB supported by abstract components :
Satellite Conf and Eqpt Mgmt
Plan/ Autonomy Framework
AOCS
Thermal
System mode mgmt
Power
Central FDIR
SSMM Mgmt
OBT Mgmt
P/L Manager
Software bus
Execution framework
PUS specific
Abstract component services
Component services
Connector services
Container services
PUS and MTL services
OBCP interpreter
PUS monitoring
Avionics Equipment virtual devices =SOIS DVS
Context Mgmt
On-board time =SOIS TAS
Communication services addressing physical distribution across nodes = SOIS MTS
RTOS
Libraries: mathematical, etc.
SOIS Subnetwork layer (1553, CAN, SpW) (including HDSW)
BSP

Figure 5: The current status of the software reference architecture

6. OPEN POINTS AND WAY FORWARD

6.1 Hierarchical components

The adoption of hierarchical decomposition of components can be an effective means to master the complexity of software design as opposed to attaining containment relationships between components. A component could be decomposed to include a set of child components with delegation and subsumption relationships between the interfaces of the parent component and those of the child components.

However, in contrast to other approaches, the various non-functional dimensions applicable to the space domain complicate the picture.

The solution currently preferred is a form of "grey box" containment, in which the parent component reveals only information that is necessary to perform the required forms of analysis (schedulability analysis, stack analysis, etc..)

The solution to this problem shall in fact preserve the verification of composable and compositional properties, which is one of the foremost requirements placed on the component model and a cornerstone of the advocated approach.

6.2 Error management

The Fault Detection, Isolation and Recovery (FDIR) is a major function of the spacecraft platform system and software. It has a high complexity because it links together hardware behaviour, hardware failure, hardware reconfiguration, software, software monitoring, software reconfiguration actions such as mode changes, and last but not least, time, response time and latencies. Getting it to work is a difficult task. Its complexity could be reduced through a proper architectural design with separation of concerns. But this is a task that goes above the software architects, and must be tackled as well by various kinds of system engineers (avionics, control, dependability, etc).

It is considered up to now that FDIR can be subject to a reference architecture, where the software implements elementary actions which are highly parameterized and data driven. The setting of the parameters at system level allows configuring the FDIR, while keeping the software architecture relatively compliant to the separation of concerns goal.

6.3 Application or execution platform?

Mapping some functions on the architecture sometimes results in the multiplication of the same software mechanism into the application software. As a matter of optimization, it is sometimes useful to optimize the architecture by moving this common mechanism into the execution platform, although it belongs by nature to the application.

One example is the monitoring of the house keeping parameters, which allows determining the health status of the spacecraft. This monitoring is performed by all the functional chains on their own parameters. Duplicating this function as the same component in all the application building blocks would be heavy. Instead, it becomes an "abstract component". The monitoring common mechanism can be moved into the execution platform, as a "service for abstract components".

A more difficult case concerns the components that should have access to nearly all the existing provided interface. This would increase the complexity the architecture a lot and goes against the separation of concerns. An example is the so called OBCP (On Board Control Procedure), a sort of script which is send to the spacecraft during operation, interpreted on board, and which can activate many software resources. There is today no satisfactory solution for this sort of component.

6.4 Towards a new validation process?

Verification and validation activities are a major cost and schedule driver for on-board SW development, so a reference architecture aiming at reuse must take them into account.

The current software architecture mixes functional and non functional aspects. Therefore the preservation of assumptions can not be demonstrated.

Demonstrating the preservation of the assumption is much more difficult than rerunning all the tests (TSP or not). A new reference architecture that separates functions (e.g. a component model), that support the formalization of the assumptions and their verification in the implementation, should inverse the trade-off, and should allow a maximum reuse of the tests. For example, configuration parameters of the building blocks must also to configure the test suite. The specification of a building block should include requirements on the reuse of its test suite.

As a consequence, the traditional validation tests should be split into sub-steps, some of them being reusable as is.

6.5 Method and tools

The software architectural concept defined to realize the reference architecture strongly relies on the adoption of domain specific languages, methods and supporting tools.

Product lines engineering, model driven engineering and component orientation require specific modelling languages, automation or semi-automation of many steps and generation capabilities in support of property preservation, separation of concerns and variability.

In addition mapping of functional chains and variability factors to the software architectural concept are system level concerns that need to be further developed in the methodological frame

The overall approach requires a high degree of control over support tools to be adopted in order to obtain proper customizations and maintainability over time.

Control of tools can be obtained either through a collaboration among the space community and commercial tool vendors or through the adoption of open source tools.

However the today's limited number of commercial model driven tool vendors (nearly a monopoly), considering also the high number of tools that have been discontinued recently, the high costs to develop and maintain domain specific tools and the small dimension of the space market are all driving factors the shall be considered in the space community and lead to the adoption and to the investment on open source tools.

7. CONCLUSION

The elaboration of a reference on-board software architecture is ambitious. Key elements are essential to the success of the approach:

- an appropriate domain engineering to define the perimeter of reuse of the software
- identification of the variability factors within that domain
- separation of concerns, between functional and non functional, between application and execution platform
- adaptation of the engineering process, in particular for Validation, and elaboration of a clear reuse process allowing the identification of a minimum delta-validation.

The success of the approach will be measured by the level of reuse of the components. The operating system RTEMS is currently the first eligible space software building block. The expectation is that the space software industry is able to transfer its development effort from non sophisticated repetitive developments such as data handling, basic communication, control, thermal and power frameworks, towards sophisticated, value added, expert functionalities such as advance control laws, autonomy, intelligent fault detection isolation and recovery, on-board sensor fusion, which are essential for future missions such as global earth observation, deep space science or exploration.

8. ACKNOWLEDGEMENT

The authors acknowledge the contribution of their colleagues to this work, in particular the Savoir-Faire working group [P.Arberet (CNES), J.Li (SSC), M.Ferrugato (SSF), P.Hougaard (Terma), A.Rossignol (Astrium), S.Mazzini (Intecs), S.Fowell (SciSys), S.Montenegro (DLR), A-I.Rodriguez (GMV), G.Garcia (Thales Alenia Space)] and M.Panunzio (University of Padua).

ESA has recently achieved parallel activities [2], [3], [4], aimed at preparing the grounds for improving the onboard software reuse in a systematic manner:

− Component Oriented Development Techniques (COrDeT) – two parallel TRP activities each lead by one prime (Astrium and Thales Alenia Space)

− Domain Engineering (Domeng) – GSTP activity lead by GMV.

These contracts involve relevant stakeholders in the prime and software supplier sectors: Astrium Toulouse, Thales-Alenia Space Cannes, SciSys, GMV, Intecs, as well as academia and SMEs: the University of Padua, P&P Software and SoftWcare. The objectives of these projects were to:

− Define the knowledge domain relevant to spacecraft (satellites and space exploration systems)

− Define generic architectures (design models) for the future implementation of the components (building blocks)

− Propose a BB and interface standardisation scheme

− Identify suitable technologies to support the domain engineering.

9. REFERENCES

[1] M.Panunzio and T.Vardanega: "*On component-based development methods and high integrity real-time systems*", 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Beijing, China, August 24th-26th, 2009.

[2] A.-I. Rodríguez, P. Rodríguez, I.-L. Vera & E. Alaña, "*Modelling the Space Domain: Domain Engineering for Avionics/Embedded Systems*" DASIA, Palma de Majorca, 2008

[3] L. Planche, "*Component Oriented Development Techniques: Assessing a Domain Engineering Approach to Space Software Engineering",* DASIA, Palma de Majorca, 2008

[4] P. Rodríguez-Dapena, "*CORDETS (Component Oriented Development Techniques) and DOMENG (Domain Engineering)"* DASIA, Palma de Majorca, 2008

[5] Savoir-Faire working group, "*Savoir-Faire On-board software reference architecture*", TEC-SWE/09-289/AJ