A DATA-CENTRIC APPROACH FOR MODELING AND IMPLEMENTING SATELLITE SOFTWARE

B. Jantscher[1], H.-J. Herpel[2], M. J. Kratschmer[2]

bernhard.jantscher@gmail.com, [hans-juergen.herpel | martin.kratschmer] @astrium.eads.net

[1] Hochschule Ostwestfalen-Lippe, Institute for Industrial IT (inIT),
Liebigstraße 87, 32657 Lemgo, Germany

[2] EADS Astrium GmbH, Data Processing & On-Board-S/W,
Claude-Dornier-Straße, 88090 Immenstaad, Germany

Abstract

Over the past decades, software development methods for satellites have undergone several changes. As in many other industries today, a lot of the systems functionality is implemented in software since this is more flexible than hardware solutions in many cases. This increases the demands and requirements regarding software and its development life cycle. New concepts must be investigated for finding better ways to tackle these challenges while at the same time using available hardware resources effectively. EADS Astrium is conducting concept studies regarding future software architectures for distributed on-board systems. This paper introduces the concept of a "global data pool" that acts as a communication interface for interacting software components. This results in a component-oriented software architecture that is based on a data-centric middleware layer. This paper elaborates the general principles of data-centric architectures and the possibilities of applying model-driven engineering, and discusses the current results shown by prototype implementations.

1. INTRODUCTION

Since its early beginnings, the spacecraft/satellite engineering domain - just as many others - has become heavily influenced by applied computer science and software engineering. Software cannot merely be considered a complementary tool for operating payload, simply due to the enormous amount of functionality and responsibility it carries. Software architecture and the software development process have become crucial parts in space missions. However, the functionalities and responsibilities that are being transferred to the software layer render software architectures very complex, making them difficult to handle. Several studies have already been undertaken in order to investigate the challenges software engineers are facing in current and future projects [1][2][3]. For instance, the Savoir-Fair working group has identified three key challenges regarding the software development life cycle [4]:

1) Releasing new versions of a software in a short time: final versions of the software are simply expected earlier, e.g., for Assembly, Integration and Testing (AIT).
2) Being tolerant towards late definitions or changes of some requirements: this can be seen in projects of other industries as well - changes in software seem to be cheaper than modifications on a physical subsystem. This, together with 1) actually shortens the overall development cycle for software.
3) Being flexible regarding integration strategies: the software shall be available for integration of certain subcomponents, while not all parts of the system can be integrated yet.

The Savoir-Fair working group suggested facing these challenges by designing architectures based on "building block" patterns, i.e., architectures shall be composed of exchangeable and reusable software modules.

However, despite these challenges in software development there are also new possibilities emerging. Developments on the hardware sector show that higher performance components will be available for future missions. This paper proposes a possible future software architecture that shall face the challenges of software development using the given possibilities of (relatively) high-performing hardware. The proposed architecture relies on a component-based, distributed, and data-centric approach. Component-based means that we focus on modularity in the architecture. We also presume that future software architectures need to operate in distributed environments, i.e., in a small network of on-board computers. Finally, we suggest a data-centric communication pattern for interaction between distributed software components on board a satellite.

2. MOTIVATION

In many software architectures of current satellite designs communication on board the satellite is message oriented. Generally, this means that communicating software components must address each other directly and hence need to "know" of one another. This results in a software architecture with very tightly coupled components. These are very problematic when it comes to making changes in the system, as changing one component can have significant impact on the other components as well. Relating this issue to the previously mentioned challenge of being tolerant towards changes shows that such an architecture is unlikely to be an adequate solution for facing this issue. Changing requirements in a system with tightly coupled components has high costs of change as a result.

3. A DATA-CENTRIC APPROACH

3.1. General Idea

The data-centric approach follows a different paradigm in communication. In literature it is also commonly referred to as "publish/subscribe" pattern [5][6]. It is based on the assumption that the most common purpose for communication within a system is the exchange of certain data. Following this idea, the concept suggests that communication should be focused on the data itself and not the entities producing or using it.

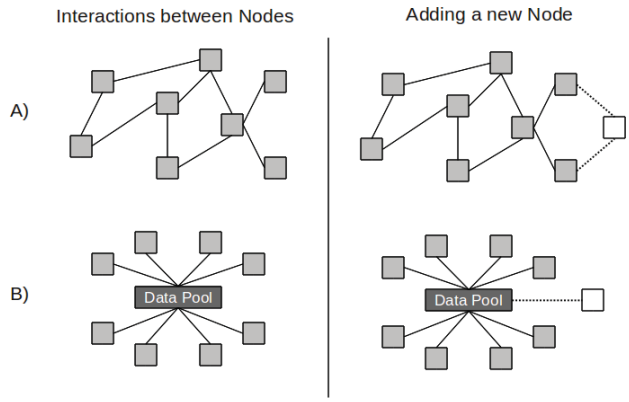

Figure 1: Message-Centric (A) and Data-Centric (B) Communication

Figure 1 illustrates roughly the interactions and coupling between software components in both the message-centric and data-centric approaches. Part A represents "message-centric" communication, where two communicating entities are aware of each other and exchange information by messaging. Part B represents "data-centric" communication, where the two communicating entities do not know each other, but exchange known types information by publishing and subscribing to a data pool. The key difference between the two approaches lies in the scalability of the resulting architectures. In a "message-centric" approach, we have a very tight coupling between components and the removal of one component can have impact on many other components. In the "data-centric" approach, we can remove and add components without any (direct) influence on the others. Of course, there eventually would be a problem if we removed a component providing certain data without replacing it adequately. However, the overall system remains flexible towards changes since components only rely on data, but do not care about other components involved in the processing. In general we can say that in a data-centric environment the responsibility of actually interacting with other software components is no longer a concern of the components themselves.

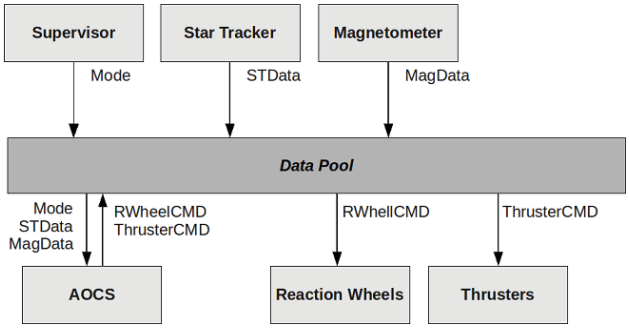

Figure 2: Example of Software Components Interacting using a Data Pool

Figure 2 explains the interaction between software components and the data pool using a very basic example. The figure shows a number of software components that publish "parameters" in the data pool. A parameter represents the state of a certain subcomponent of the spacecraft, e.g., housekeeping data or payload/equipment data. A parameter can be any kind of data that needs to be shared between software components. In this example the supervisor provides the current mode of the spacecraft, and a star-tracker and magnetometer provide their instrument data. The Attitude and Orbit Control System (AOCS) reads its required parameters from the pool, processes them, and publishes the output again in the data pool, making it available for other components. However, it is important to understand that even though the physical interaction only takes place between software components and the data pool, logically the components are still interacting with *each other*, as figure 3 shows. However, their communication interface is no longer a message protocol, it is now the parameter specification.

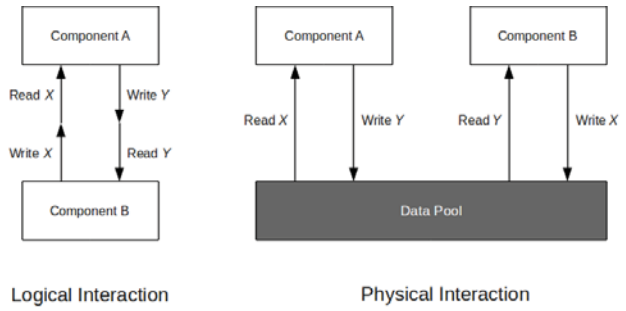

Figure 3: Logical and Physical Interaction of Software Components

Each parameter is uniquely identified in the data pool, it carries a value (i.e., the actual data) as well as other attributes. The most important attributes of a parameter are its current age and its maximum age. The current age is updated every time a parameter is written by a component. The time between two updates must never be longer than what is specified as maximum age; the parameter would otherwise be declared invalid. This constraint is very essential for communication. In a message-based approach, if component A sends a request to component B, and component B does not respond (for instance because it crashed), then component A realizes that the required data cannot be provided and reacts on this situation accordingly. In a data-centric architecture though, component B would crash but its parameters that were published would still remain available to component A. This is why the maximum age attribute and its automatic check are so important in this concept, they provide a way for software components to verify that a parameter is still up-to-date and usable.

3.2. Application as Satellite Middleware

Figure 4 illustrates a data handling system with a software architecture that applies the proposed data-centric approach. Each computer on board the spacecraft accommodates a set of software components that interact using the data pool, which is synchronized across the network. From a software engineering point of view, the software components access a data-centric middleware layer, which coordinates the interactions between the components and the data pool. An important characteristic of this architecture is that decoupling takes place not only at a logical level (i.e., avoiding the need of "direct addressing" between communication entities), but also physically, with respect to the computers the components are running on. This means that, given a data pool that is able to provide data across a distributed system, we do not necessarily need to consider where we place our software components in the design phase. It is also worth mentioning that software components may also act as interfaces to other equipment/payload, by making their equipment data available to other components via the data pool, as it is also seen in figure 4.

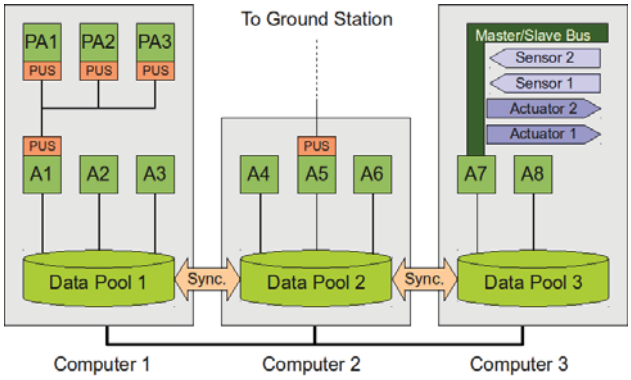

Figure 4: A Data-Centric Middleware Architecture

Changing the way of how software components interact in such a drastic way is a lot of effort, and hence it should also deliver benefits that justify this undertaking. The following list provides a set of expected improvements that should come about as a result of applying this concept:

- The reduction of coupling shall make the development of software more flexible regarding component architecture and the definition of execution locations. In other words, things like "where does a component run", "when and in which order does it provide/fetch data" and "which component is providing that data" do not need to be defined at an early design stage and shall allow changes of requirements at low cost. This is a key requirement that was discovered by several studies so far [7, Sec. 11.3.4] [3] [4].
- Providing the software development process with a clear definition of interfaces without direct coupling between components shall be beneficial for parallel development and integration of third-party software.
- In general, the development of software components shall become more simple. The middleware API shall reduce the efforts for transporting data. In fact, it shall free software components from the responsibility related to transporting data. This transportation responsibility is a very significant factor when it comes to costs related to changing requirements of a component. The overall aim is to have a system of relatively small and well-defined software components that can be reused and replaced easily.

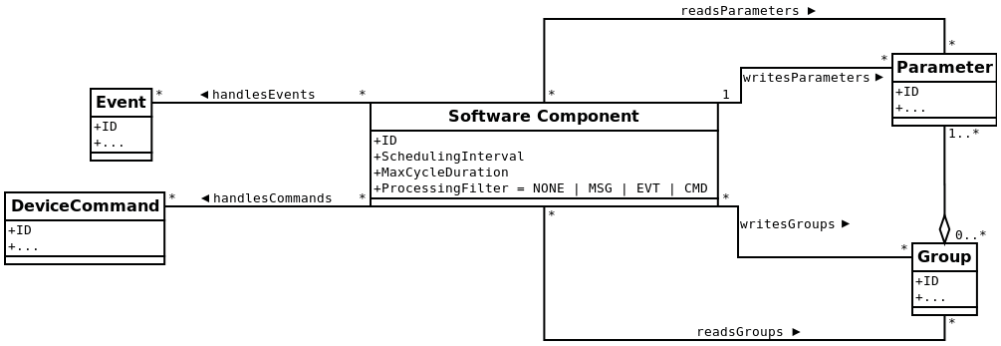

Figure 5: Proposed Metamodel in UML

– The use of a central data pool is very beneficial for Failure Detection Isolation and Recovery (FDIR) mechanisms. They can be provided with a snapshot of the overall health status of a spacecraft.
– This architecture shall allow redundancy on software level. This means that in a system with several processing units and a synchronized data pool, the failure of one computer could be compensated by migrating components of the failed computer to a backup system until the original computer is available again. This would be possible since the location of a software component is by definition not relevant to the system. However, exceptions are software components that depend on the hardware interfaces connected to their computer.
– A big potential for improvement is related to testing. Using the data-centric concept it is very easy to perform unit tests for software components, since the parameters that are needed for emulating a component's environment can be easily injected into the data pool via test scripts. In the same way it is even possible to test certain subsystems or even the whole system. This can also be very usable in early integration phases of the spacecraft. Parts of a satellite can already be integrated and tested, while simulating the subcomponents that are not available yet. This can also be useful for error isolation procedures when "debugging" the satellite or one of its subcomponents.

Another important advantage of this approach is that it provides the foundation for applying model-driven engineering, which is described in the next section.

4. APPLYING MODEL-DRIVEN ENGINEERING

Model-based design has become a very popular methodology in various engineering domains. When defining a software architecture we can apply model-driven engineering if our system infrastructure allows us to abstract a formal metamodel of it [8]. The introduced data-centric approach for a middleware design would allow us to do that.

4.1. Abstracting towards a Metamodel

It is important to understand that the degree of abstraction of a certain domain is arbitrary and depends on practicality. When considering what aspects of a domain should be modeled, a good assessment tool is detecting reoccurrence and potential reapplication of certain patterns. For instance, software components are a key element in the overall avionics architecture. They occur in various different types, but all of them have common features. Hence, they are a typical candidate for abstraction and a lot of information about them can be formally defined in our metamodel, for instance: identifier and name, invocation filter (e.g., event based invocation or cyclic invocation), maximum allotted execution time per invocation, maximum allotted memory, and the parameters that are read and written by the component. Parameters were briefly introduced earlier, and they too are key entities that must be modeled. Along with their identifier, value, age and maximum age, we can define boundaries such as minimum and maximum value, or additional information like their unit of measurement. This information together with some additional definitions let us derive a set of ground facts that are relevant for our abstraction:

1) The key entities within the middleware infrastructure are software components. These components have access to the data pool for reading and writing parameters, and are specified over a set of attributes.
2) A parameter consists of a unique identifier and is specified over several other attributes.
3) A parameter can be written by only one software component, but read by several.
4) Parameters can be assigned to groups, and whole groups can be read/written by software components.
5) Software components may need to handle certain events or device commands, which are also defined and handled in the data pool.

Figure 5 depicts a UML representation of a software component and its related entities. Of course, this is an incomplete abstraction, just to show the most important features that are necessary for understanding this concept and the possibilities it brings.

4.2. Resulting Benefits

As mentioned earlier, a model provides us with a formal definition of our architecture. This means that when we create a model instance, we can be just as flexible as the metamodel allows us to be. As a result of this, a lot of mistakes related to designing a software component can be avoided since the metamodel restricts our actions. A clear example is the restriction regarding writing a parameter. As figure 5 shows, there is a restriction in our metamodel that allows us to assign only one software component as the writer for a parameter. This definition prevents us from building a model instance, in which two or more software components act as writers. Or, from another point of view, we can make sure that every parameter is provided by some software component.

Of course, the restrictions we can define have their limits. In fact, the limits always depend on the technique used for metamodeling. In the previous example for instance, we defined that a software component can write a whole group of parameters. However, this implicitly requires that for every parameter of that group, there is no other writer but that component. UML has no practical way of defining this restriction. But since a model consists of formal definitions, it is possible to create verification mechanisms that investigate a model regarding additional constraints. Again, the possibilities available for both model-defined constraints and add-on constraints depend on the used technique.

In general, when applying model driven engineering on the proposed middleware architecture, the following requirements can be checked and verified in the modeled software components:

- Are there any collisions in identifiers?
- Is every parameter written by exactly one software component?
- If read/write permissions are modeled as well, are there any permission violations?
- Is every device command and event handled? Or, which events/commands remain to be handled?

Additionally, it is also possible to derive important information from the model, for instance a data flow diagram showing how software components interact using the interfaces. Another piece of information could be the minimal memory consumption based on a) the parameters that are requested by each component, b) requested persistent memory by component, and c) the total number of parameters in the data pool. Information for scheduling and CPU utilization can also be extracted based on the execution specifications defined for every software component.

4.3. Code Generation

Another important benefit of model-driven software engineering is the possibility of code generation. Due to the information provided by the model of the system, it is possible to generate framework source code for each software component, as shown in figure 6.

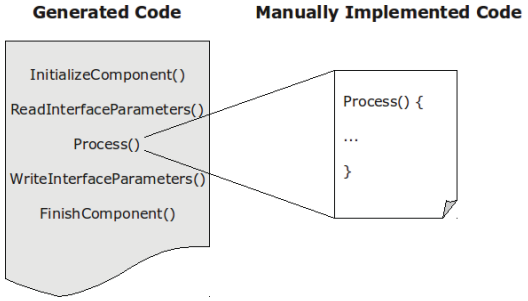

Figure 6: Generating Framework Source Code from the Model

Apart from the practical advantages like saving time and easier maintenance, a model driven design might lower the costs of the verification process. A verification process involves investigating the source code of a whole project. In the case of our architecture, it means we have to investigate the source code of every software component. However, when applying model-driven engineering and source code generation, the list of elements to be verified roughly consists of:

- The modeling environment, i.e., the metamodel and meta-metamodel, and the restrictions checker
- The middleware layer
- The code generator for producing framework code for software components
- The custom logic code for every software component
- Any other software module that uses our middleware but is not a modeled software component

At first glance, there seems to be a greater amount of components due to incorporating modeling tools. However, it is important to understand that the modeling environment, middleware layer and code generator are very static elements in the software development process. They usually undergo very little changes once they have reached a certain maturity level. Once all these tools are verified, what is left to verify simply boils down to functional code, since all

infrastructure code is being generated from the model. Three main consequences can be derived from this:

1) A project needs to be of a certain complexity in order to justify the efforts for applying a model-driven approach since there is more verification effort involved initially.
2) Once these overhead efforts are overcome, the verification efforts per functional software component are reduced to a minimum. This also means that the costs for a later change are minimal. For instance, adding another software component (which is already simplified due to the data-centric design) does not involve any efforts except verifying the added functionality source code. Without model-driven engineering, all infrastructure code (initializing the component, getting data, writing data, etc.) would also need to be included into the verification process, possibly not only for the new component, but also for all other components that interact with the new one.
3) A lot of verification effort (and thus, eventually, time and money) can be saved if a model-driven architecture can be reused in other projects.

5. CURRENT STATUS AND RESULTS

The concept of a data-centric middleware and the approach in model driven engineering are currently evaluated in the course of a research project at EADS Astrium in Friedrichshafen, Germany. The project has yielded a working prototype that is intended to be an evaluation platform for assessing the concept's feasibility regarding its offered functionality. This means that the developer and user (e.g., On Board Data Handling (OBDH) spacecraft operator) interfaces of the middleware are defined and evaluated, but the underlying layers are not yet implemented using technologies qualified for spaceflight. The execution platforms of the prototype environment are common GNU/Linux operating systems. Still, the API layer available to the user was designed with respect to coding restrictions required for qualification (e.g., avoiding issues like dynamic memory allocation, recursions, or platform specific data types). The following paragraphs give a short overview of the most important features provided by the prototype.

There exists a functional middleware that allows the operation of software components across a network. The software components consist of a processing logic that is invoked either cyclically or based on certain events. The software components are provided with a simple API for accessing data pool and its functionalities. The middleware infrastructure allows the automatic monitoring of all running software components. We have further shown that functional C code generated by Matlab can be easily integrated into the software architecture, by wrapping the software component framework around it. Input and output parameters of the Matlab model are being connected to parameters in the data pool. Also the automatic generation of source code based on a component model has successfully been demonstrated. It was shown that a lot of infrastructure code can be written by the code generator, leaving the developer with only the component logic and parameter processing to be added.

The middleware incorporates a framework for managing events. Events are categorized and occur asynchronously at runtime. They can be generated by different entities, e.g., by software components during their processing, or internal data pool operations. The architecture further allows software components to be implemented as specific event handlers that perform follow-up processing of an event.

The effective management of parameters is one of the core features of the concept. As mentioned earlier, every parameter has certain constraints (e.g., minimum value, maximum value, or maximum age) and statistics (lowest value, highest value, or mean value.) that are kept track of. Parameter constraints can often be directly derived from system requirements (e.g., lowest orbital altitude, highest velocity, or minimum internal temperature). For every parameter it can be specified if an event shall be generated, when constraints are violated. Parameters can further be organized in groups, for instance according to certain payload or housekeeping categories. The transformation from an input value to an engineering value no longer needs to be done by the component developer. The data pool keeps information about how to transform the parameter (e.g., over a polynomial function) and automatically calculates the engineering value when a parameter is written. In the data pool it is also defined, which components are allowed to read/write a parameter.

The architecture also provides the possibility of managing On Board Control Procedures (OBCPs). OBCPs are scripts that are executed for performing special tasks on board the spacecraft. These OBCPs can also be stored inside the data pool. Special software components can act as OBCP handlers by forwarding execution requests to an external interpreter. Additionally, an event can be linked to an OBCP that is executed then when the event occurs. For instance, the power subsystem constantly publishes the remaining battery voltage in the data pool. If the voltage goes below its allowed minimum,

the data pool automatically generates an event reporting the violation. The event handler sees that this specific event has an OBCP attached and orders the execution of this OBCP. The OBCP handler starts the OBCP interpreter which executes a script that shuts down all non-vital payload.

Similar to the concepts of events and OBCP execution, the middleware also supports automatic generation of messages according to templates. These messages can be processed by message handlers in order to forward data from the data pool to recipients that cannot access the data pool. For instance, a message template can be defined for generating a housekeeping report according to the Packet Utilization Standard (PUS, [9]). A dedicated message handler would read the request and the template from the data pool, generate and format a corresponding PUS message, and transmit it to the recipient. In a similar fashion, there are handlers which receive external messages, extract information from them, and publish this information in the data pool. These mechanisms show that the data pool is also capable of communicating with external entities.

6. CONCLUSION AND OUTLOOK

This paper introduced a data-centric approach for designing a component-based and distributed middleware for data handling on satellites. The motivation for investigating the feasibility of this concept is to face certain challenges in software development in a more effective way than with previous approaches. The concept and the advantages of this approach were introduced, as well as the possibilities of turning it into a middleware for satellites. Further, the idea of applying model driven engineering based on the data-centric architecture was discussed.

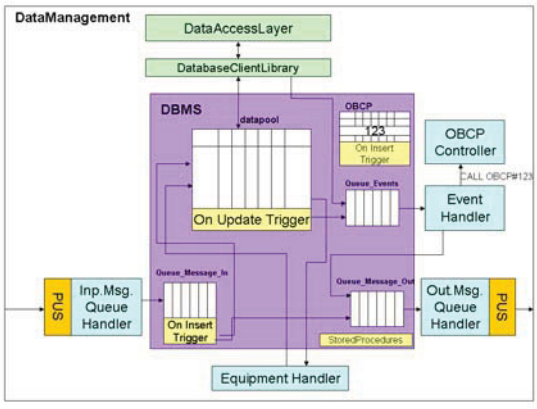

Figure 7: Middleware Implementation using an Embedded Database

The overall concept is still under evaluation, but prototype implementations have shown promising results and delivered interesting features and possibilities. The next step in the project will be the evaluation of technologies for implementing a version that could be qualified for spaceflight. Possible technologies under investigation are the Data Distribution Service (DDS, [10]) as well as several embedded database solutions (see figure 7). A second follow-up activity will be the definition of standard parameters for the middleware, on both component level as well as system level. A further task for the future is investigating the possibilities of automating the integration of Matlab models into the software architecture.

7. ABOUT THE AUTHORS

Bernhard Jantscher is a student enrolled in the International Master's Program for Industrial IT at Ostwestfalen-Lippe University of Applied Sciences. He is currently doing his master's thesis [11] on middleware in satellite systems in cooperation with EADS Astrium in Friedrichshafen, Germany.

Hans-Juergen Herpel has 25 years experience in embedded systems for safety critical applications. Currently, he is responsible for the R&D activities within the department of On-Board Software and Data processing.

Martin Kratschmer designed and implemented large parts of the middleware's functional prototype during his engagement as trainee in the department of On-board Software and Data processing.

8. REFERENCES

[1] The assert Project. Final Report (accessed 07/2011). http://www.assert-project.net/Publications

[2] The National Aeronautics and Space Administration (NASA). NASA Study on Flight Software Complexity. Technical report, Systems and Software Division, Jet Propulsion Laboratory, California Institute of Technology, 2009

[3] A. Pasetti and O. Rohlik. The Cordet Methodology. Technical report, P&P Software, 2007.

[4] SAVOIR-FAIRE Working Group. Space On-Board Software Reference Architecture. In DASIA (Data Systems in Aerospace) 2010, Budapest, June 2010.

[5] T. A. Bishop and R. K. Karne. A Survey of Middleware. Towson University Dept of Computer and Information, 2002.

[6] Andrew S. Tanenbaum and Maarten Van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[7] J.R. Wertz and W.J. Larson. Space mission analysis and design. Space technology library. Microcosm, 1999.

[8] J. Sprinkle, et al. Metamodelling: state of the art and research challenges. In Proceedings of the 2007 International Dagstuhl conference on Model-based engineering of embedded real-time systems (MBEERTS'07). Springer-Verlag, Berlin, Heidelberg, 57-76, 2007.

[9] European Cooperation for Space Standardization (ECSS). Telemetry and Telecommand Packet Utilization: ECSS-E-70-41A. ECSS Secretariat, ESA-ESTEC, Requirements and Standards Division, 2003.

[10] The Object Management Group. Data Distribution Service Portal (accessed 06/2011). http://portals.omg.org/dds/.

[11] B. Jantscher. Assessing a Data-Centric Design Approach as Middleware in Spacecraft Software. Master's Thesis. To be published in early 2012.