A Component-based Framework for Space Domain Software Applications

Thomas Vergnaud, Mathieu Le Coroller, Guillaume Véran

A Component-based Framework for Space Domain Software Applications

A. Thomas Vergnaud[1], B. Mathieu Le Coroller[2], C. Guillaume Véran[2]

1: Thales Communications, Palaiseau, France

2: Thales Alenia Space, Cannes, France

Abstract: This paper presents research carried on by Thales on component based software engineering for the space domain. We outline the space domain context and give the general architecture of MyCCM, our component framework. We explain how we implemented a space-specific component framework with MyCCM and what results we got from experiments. Applying component design to on-board space applications induces a very light overhead while allowing automatic code generation, as well as code reuse and application redeployment. It thus helps cut development costs and improve the reliability of software development.

Keywords: Component based software engineering, real-time embedded systems, Lightweight CCM

1. Introduction

Component-based design approaches have initially been created for information systems. They provide more reuse capabilities and modularity than traditional object oriented approaches [1].

Research has focused on applying the component based approaches to real-time embedded (RTE) systems. Such systems have specific constraints regarding reliability, strict execution deadlines, memory and computing power limitations, which are not considered in information systems. Therefore, component based software engineering (CBSE) has to rely on tailored frameworks rather than on general purpose ones. Thales has developed such a framework, named MyCCM.

In this paper, we describe experiments made in Thales to apply CBSE to the space domain: we developed a use case named System Engineering and Middleware based on standards for Space domain (SEMS). This is a joint work between Thales Communications, Thales Research & Technology and Thales Alenia Space, driven in 2009 in the scope of Thales innovation platforms.

We first describe the context of the space domain. We then recapitulate general considerations on component-based approaches and present the MyCCM framework. The two sections after describe some requirements from the space domain, and how MyCCM fits them. We then present some results and measurements to show the relevance of our approach, and conclude by showing the connection with research projects such as Artemis CHESS and ITEA2 VERDE.

2. Need for Component-based Design in Space Domain

The very nature of industrial programs in the space domain has led to stringent requirements for reliability and availability; hence implying high quality insurance criteria. Some of them obviously trace to the software product. In that context, Thales Alenia Space targets the mastering of the design of its embedded software at both interfaces and real-time levels.

The model-based approach has been considered on two facets:

Internals of applications: Model-Driven-Engineering is used as a UML-based development environment dedicated to the implementation of the various items in the software-system.

Composition of applications: In order to build the software-system, we selected a Component Oriented Architecture.

Component-Oriented Architecture is indeed a key-to-success in software development in the European Space community. The particular constraints of European programs in the space domain often lead to outsourcing and co-contracting, somehow in complex and multi-national consortiums when geo-return comes into play.

Component-oriented techniques allow for mitigating the risks at integration by emphasizing on the interfaces and contracts of the components, and handling the glue between the deployed instances. In addition to these organizational concerns, component-oriented techniques also favour the decoupling of applications, hence promoting reusability, and the definition of on-the-shelves products.

We leveraged the outcomes of internal studies on component-oriented architecture [2] when deploying one on the on-board software of the Globalstar-2 constellation.

At that time, the architecture relied on the OMG IDL textual notations, and generative techniques. Though already expressing lots of benefits at the interface-level, the methodology shown limits as it lacked a deployment view for components, therefore hampering the capability to drive automated verifications on the overall embedded software design. Besides, proprietary extensions to the standard IDL had shown necessary so as to cope with space-specific standards. Moreover, IDL only covered the functional interfaces of our components, letting the non-functional concerns being handled elsewhere.

We needed an evolution of the component-oriented techniques we used, as seen in the ASSERT project [3], while still bringing into lines open and well-spread standards and in particular the ones from the OMG.

For space domain software architecture, the homogeneity of interfaces is at stack; around twenty percent of the code is devoted to communication respecting space specific standards. The use of standards so as to ease communications with commercial and institutional partners, and the capability to adapt them to the actual technical needs are of utmost importance.

3. Overview of Lightweight CCM and MyCCM

In this section, we recapitulate the main principles of component based software engineering (CBSE) and present our component framework, MyCCM.

3.1 Component based software engineering

Components are pieces of functionality that are to be assembled one with another in order to provide the full functional coverage of the system. This allows breaking down the whole system into smaller pieces, truly independently manageable, easier to develop and to reuse. Component-based approaches typically rely on three concepts: component types, component implementations and component instances.

A component type describes the services the component provides, as well as the ones it requires from other components. In that sense, components can be seen as an evolution for application design, compared to classes of object-oriented languages: classes only describe what services (methods) they provide to other classes.

Provided or required services are described by ports; ports are associated with component definitions. Ports thus describe interaction points; they define the types of exchanged data and the semantics of these exchanges. Components are to communicate one with another only through ports. Depending on the component model, ports can implement complex interaction semantics, or can represent very basic interactions (e.g. operation call). In this later case, the interaction semantics in itself is then deported to what is called a connector. Connectors are used to connect ports of components while providing complex communication mechanisms.

Different component implementations can be associated to a given component definition. A component implementation represents the internals of the corresponding component definition, the same way a class implements an interface in object-oriented languages. It thus holds the business code.

Component instances are the actual components to be used in the architectures, just like objects compared to classes. Ports of component instances are to be connected one with another in order to create a complete architecture. They can be associated with tasks, mutexes, etc. in order to control component entry points and the execution of component business code.

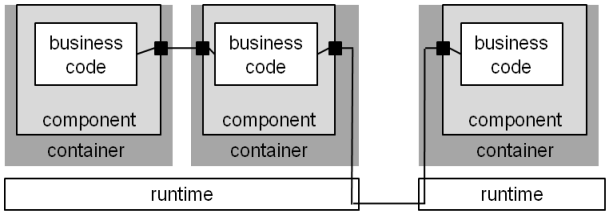

Figure 3-1: Implementation of a component-based architecture

Figure 3-1 summarizes the resulting organization of a component-based application: business code is encapsulated within components instances that isolate them from the execution environment. Components instances are controlled by containers that manage communications and execution resources. Containers rely on the runtime. Inter-component communications can be local operation calls between two containers, and use runtime mechanisms (e.g. for remote communications). In both cases, this is completely transparent for the nested business code.

3.2 Lightweight CCM

CCM (CORBA Component Model) is an OMG standard [4] that describes a component model. Lightweight CCM is a standard subset of the CCM. It is dedicated to providing a CCM compatible component model suitable to the needs of distributed embedded systems.

Lightweight CCM defines the notion of software component as an envelope that wraps the user business code, isolating it from the execution environment. User code thus communicates with the outside of the component only through the component envelope. This envelope is usually

described using the IDL3 language. IDL3 defines two communications ways offered to the user code: interfaces and events. Interfaces are sets of operations; they can be either provided by a component (facets) or required by it (receptacles). The same way, events are either sent by the component (event sources) or received by it (event sinks).

IDL3 constructions are transformed into sets of programming interfaces that are to be implemented either by the component framework (i.e. the component envelope) or by the user code. This later case corresponds to the services provided by the component, described in its IDL3 declaration. This set of programming interfaces is named CIF (Component Implementation Framework).

CCM is originally defined as the CORBA 3 standard, and thus typically relies on an ORB to manage communications. However, this is not mandatory, as the CCM purpose is to hide the ORB from the business code, nested in components. Therefore, one can use virtually any runtime to support the execution of CCM architectures, provided that it can manage the two communications paradigms (operations and events).

Data types used in CCM are the CORBA ones: long, short integers, floats, etc.

IDL3 itself does not address the description of component deployment. The CCM is thus usually associated with another OMG standard, D&C [5] that covers the deployment and the configuration of components. D&C is a very rich and complex standard, mainly adapted to the deployment of complex, dynamic information services. It lacks several configuration elements required to deploy real-time systems (e.g. thread priority definition).

3.3 MyCCM

MyCCM is a custom implementation of the Lightweight CCM standard [6]. MyCCM stands for “Make your CCM”; it is developed by Thales. It is designed to address the specific needs of Thales division in various domains (naval, robotics, space, etc.) while providing a general, standard-based, framework. It is not a single, monolithic piece of software, but a collection of frameworks, tailored for each situation.

The application design approach induced by MyCCM implies a clear separation between functional code (inside components) and infrastructure code (outside components). The infrastructure code (task management, local or remote communications between components, etc.) is to be automatically generated by MyCCM from the architecture descriptions. The functional code is completely isolated from the execution environment; it is controlled by the component envelopes, managed by the component framework. This allows reuse and redeployment of components without altering their implementation code. A consequence of this design approach is that all control features (especially tasks) must be declared at the component level and associated with ports. Components are not supposed to have internal tasks, as such tasks would not be part of the architecture description, and thus could not be managed by the code generator or analysis tools.

It is important to note that the code generators of MyCCM produce infrastructure code that would else be written by hand. MyCCM actually automatically cares of the technical part of the code; it leaves the intelligent part (i.e. the business code) to the designer.

The MyCCM framework relies on a set of internal meta-models to represent architectures. We use the CCM meta-model to represent data and component types. The other information (e.g. component implementations and instances, allocation on physical nodes, task configuration, etc.) is stored in a specific deployment meta-model. Our deployment meta-model is greatly inspired by D&C [5], as well as how allocation and configuration are represented in standards like UML/MARTE [7] and AADL [8]. Having a non-quite-standard representation for deployment and configuration help have flexibility and adaptability in the framework while keeping things simple to manage.

4. Requirements for the Space Domain

Space domain implies specific requirements regarding the component framework capabilities. We provide here a global overview of such particularities.

4.1 General requirements

The technical context of space on-board software applications is particular due to both technical and programmatic constraints.

On the technical point of view, the on-board software grew more and more complex in the decade whereas the harsh physical environment have led to rely on robust yet limited computing resources. Typical figure is to run the software of the whole spacecraft avionics on a single (yet redundant) 14 MIPS processor with 4Mo of RAM.

On the programmatic point of view, the space industry uses specific standards for the handling of communications between spacecrafts and ground stations (CCSDS TC-TM). The European space industry has also adopted standard defining common interfaces spacecrafts shall provide for ground operations [9].

These constraints imply difficulties in the use of standard solutions for the capture of components, and their implementation. For example, the sometimes limited band-width allocated to satellite communications makes it difficult to use GIOP.

In addition to these constraints, Thales Alenia Space requires extensions to Lightweight CCM in its component-based framework. These extensions tackle specific viewpoints of the system to be designed. The capture of these viewpoints on the component model allows specializing the code generation. This specialized code generation in-turn allows handling domain-specific concerns directly on the component envelope code.

4.2 Detailed Definition of types

The software component-model tends to be used as an interface model for various users: software architects, system engineers, control-law engineers. The more the team focuses on physical concepts, the more it uses engineering data.

In opposition to types defined with IDL, the engineering data model requires the capability to capture things like the legal range of values for that type, and a unit (for example, radians). Obviously these engineering types eventually map to IDL ones, but the information in that data model is also a major part of an interface specification.

In addition, as the available communication bandwidth is somehow limited, a constrained definition of types allows automating the communication data to a minimum amount of bits.

The component-based framework therefore targets Ada in order to natively benefit from the Ada strong-typing features at the component level.

Of course, conversions towards the standard CCM type are still possible; models and components designed with the extensions being therefore compatible with pure CCM models.

4.3 Real-Time Behaviour

The application complexity as well as the limited computing resources makes it very difficult to meet all deadlines when using only synchronous communications. As a consequence, an asynchronous operation invocation is sometimes required.

Rather than the use of event-based communications, the use of operations is sometimes favoured since it allows capturing in the interface model the expected response to a request. Nevertheless, selecting an asynchronous message invocation for all inter-component communications would neither be effective.

Therefore, the component-based framework needs to propose the capability to capture specialized non-functional properties as the communication semantics on a per-operation basis.

Besides on that non-functional topic, the component-based framework shall respect the quality-requirements on the Space software development. They prohibit the use of dynamic memory allocation, and limit the use of dynamic dispatching of subprogram calls; hence the standard IDL mapping for Ada cannot be directly applied since it widely uses runtime polymorphism.

By the way, since the mastering of the real-time design is also a requirement, the real-time constructions that are used by the component framework shall be amenable to verification, and in particular static verification of the scheduling of the application.

Fitting to the Ravenscar Computational Model from the Ravenscar profile for Ada [10] is therefore targeted, since it ensures that the design can be analysed using static techniques such as RMA. The use of a library of Ada generic packages developed on top of Thales Alenia Space in-house real-time operating system Ostrales (POSIX compliant) is a start for restricting the code to legal constructions.

4.4 Use of PUS communications

In addition, as around twenty percent of the code is devoted to ground-board communications, the component-based framework shall handle ground-to-board communications using PUS for the identification of embedded services.

5. Implementation in MyCCM

We adapted the MyCCM framework to fit the exact requirements of on-board space software.

5.1 Design of the framework

The MyCCM generators are coded in Java for the Eclipse platform. They use the Eclipse Modeling Framework (EMF) to represent and manipulate models. The general architecture of the MyCCM generators consists of four main parts, as described on figure 5-1:

- the implementation of the MyCCM meta-models in Java;
- a front-end;
- an applicative-side back-end;
- a runtime-side back-end.

The front-end translates concrete syntax into MyCCM models. The EMF meta-models act as a backbone for all MyCCM modules; front-end and back-ends are connected to it.

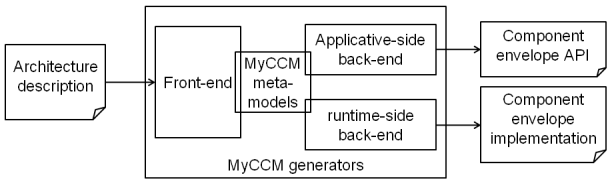

Figure 5-1: General architecture of MyCCM

Our architecture has two back-ends. The applicative-side back-end generates the component envelope code; it implements the application API (CIF) that is provided to the business code. This API only depends on the component type and data type definitions; it is not affected by component deployment, and thus remains stable from a configuration to another.

The runtime-side back-end generates the adaptation code that lies between component envelopes and the low-level runtime itself. This code typically manages tasks, communication mechanisms, etc. It thus highly depends on the deployment information and is supposed to change from a configuration to another.

Having two separate back-ends allows the generation of the applicative API for business code, even if the eventual deployment is not defined. This allows early component implementation.

5.2 Processing of architectures

The space domain brings very specific requirements regarding the description of data types and communication mechanisms. Therefore, we could not rely only on the existing MyCCM meta-models to manage space architectures. In addition, since these particularities (PUS, etc.) are specific to space domain, there is no point in adding these notions to the code MyCCM meta-models. We thus developed additional meta-models that extend the MyCCM code meta-models.

As the input models carry specific information, we chose to implement a graphical front-end. Graphical syntax is usually more attractive for engineers to learn, and more efficient to describe architectures. Hence, though MyCCM internally uses a CCM meta-model, we do not use IDL3 textual syntax.

We had to deal with very specific communication models. As stated in section 4.3, on-board software requires message passing and operation invocation, both in synchronous and asynchronous manners. Hopefully, the Lightweight CCM standard defines event transmission and operation call. We extended these notions by adding the necessary information in the complementary meta-models, so that the framework could differentiate these four communication mechanisms at modeling level, and thus handle them for code generation.

On the back-end side, the generated code relies on the Ostrales runtime. Ostrales acts as an Ada runtime, and provides basic mechanisms to manage tasks (in the POSIX way), semaphores, etc. Communications (including remote communications with the ground station) are managed by a separate library. Higher level mechanisms, such as protected queues, are implemented by MyCCM on top of the Ostrales primitives.

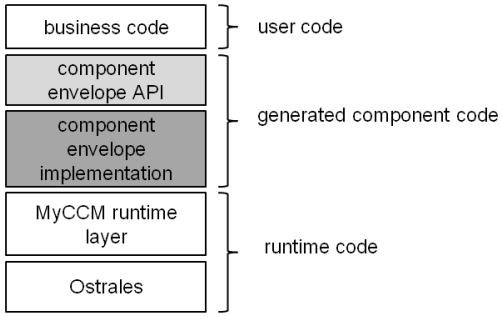

Figure 5-2: architecture of the application code

The MyCCM framework defines an intermediate layer on which the generated code relies. This layer provides an abstraction of communication and execution mechanisms (task management, all the kinds of communication available at design level, etc.) and relies on the actual runtime kernel to implement them. This way, the generated code is somewhat independent from the exact capabilities of the runtime kernel, which helps in case of the replacement of the kernel by another runtime. In addition, the intermediate layer constitutes stable code. As a consequence, the code generated by MyCCM is simpler in its design, and thus more reliable.

6. Results

As results of the SEMS study, extensions to CCM were designed to capture the constrained types, specialized communication semantics, and PUS identification of the on-board software entities.

We performed experiments on the generated code. Typical figures have been extracted from the case study of a prototype satellite. The software system was a deployment of twenty instances of components. In average, each component provided half-a-dozen interfaces, and required as many. In the case study, the average amount of operations in an interface is five. Each component also exposes a bunch of ten observable and/or configuration attributes.

Figure 6-1 exposes a subset of a component of the prototype.

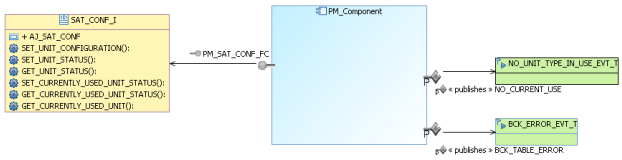

Figure 6-1: Sample subset of a component

Board-to-board communications are fully statically configured; inter-connections are hence known at code generation time. This is particularly important since it is a prerequisite for the direct execution of most software static-analysis tool. This is a major outcome of the study: before the use of the MyCCM SEMS for deployment, connections were known only after an initialization phase; that made usual static analysis tools fail.

Concerning code size, the overhead induced by the component-based approach and the automated code generation is kept minimal; benches we drove on the former component-architecture have indicated that the volume of code generated by the new component-based framework is similar to the one formerly hand-written.

Since the code is generated, this implies major gains on the development and validation efforts.

In addition, the overhead of the non-dispatching solution for board-to-board inter-component synchronous communications corresponds in the worst cases to two subprogram calls. This overhead is thus very light compared to the capabilities of reuse and redeployment of components.

Moreover, as a qualitative attribute, the component envelope code is fully independent from the implementation code of the component, and from the implementation of the required interfaces. Thus it follows the separation promoted by the component-based philosophy. The resulting applications therefore comply with the need for reusability.

7. Conclusion

In this paper, we outlined constraints of the space domain for on-board satellite software. We described MyCCM, a framework developed by Thales that supports component based software engineering (CBSE), based on the Lightweight CCM standard from OMG. We explained how we rely on CBSE to improve application design and production.

Compared to traditional, program-centered processes, CBSE allows a clearer design and thus eases the reuse and the redeployment of software elements. It implies a separation between business and infrastructure codes. As it is well isolated, the production of the infrastructure code can be delegated to automatic code generators such as MyCCM.

Our experiments showed that, though the component-related code would be complex to write by hand, its quality is greater than non component-based ones, and it does not expose code overhead. As this code is very technical, it can be efficiently managed by code generators, thus reducing the total application development costs.

Hence, CBSE brings very little overhead compared with its benefits.

8. Perspectives

SEMS environment (modelling language, model transformation engines and code generators) currently only addresses the functional dimension of components, but does not address their non-functional characteristics (e.g. timing, input and output accuracy, robustness). One investigated evolution is to take into account such non-functional requirements as soon as possible by mapping them onto the architectural model. These requirements would then be captured by the expression of extra-functional properties attached to components and finally preserved at run-time.

To achieve this, Thales Alenia Space and Thales Communications are currently involved in an Artemis project called CHESS that seeks industrial-quality research solutions to the problem of property-preserving component assembly in real-time and dependable embedded systems. CHESS targets to support the description, verification, and preservation of non-functional properties of software components at the abstract level of component design as well as at the execution level. The results of CHESS are expected to be integrated in SEMS framework at the end of the project.

Another axis of investigation is to rely on the adopted component model to enable early validation and verification activities. By integrating major technologies from Model-Driven Engineering, Validation & Verification techniques and Component-based execution platforms, it is expected to enable a rapid prototyping of the system through a projection and execution on the platform. This axis is mainly investigated in the frame of ITEA2 VERDE project.

9. Acknowledgement

The authors would like to thank their colleagues who participated to the design and the realization of the study presented in this paper: Jérôme Chauvin, from Thales Communications; Anupam Beri, from Thales Research & Technology; Franco Bergomi, from Thales Corporate Services; Florian Beaufay, Gérald Garcia and Aurélien Vionnet, from Thales Alenia Space.

10. References

[1] W. Emmerich, N. Kaveh, "*Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model*". 24th International Conference on Software Engineering (ICSE'02), 2002.

[2] C. Moreno, G. Garcia, *"Plug & Play Architecture for On-Board Software Components"*, DASIA 2003, Prague, Czech Republic, 2003.

[3] M. Bordin, T. Vardanega, *"A Domain-specific for Reusable Object-Oriented High-Integrity Components"*, 7th OOPSLA Workshop on Domain-Specific Modeling, Montréal, Canada, 2007.

[4] OMG: "*CORBA Component Model specifications (version 4.0)"* http://www.omg.org/spec/CCM/4.0/, 2006.

[5] OMG, "Deployment and Configuration of Component-based Distributed Applications Specification", OMG, 2006

[6] É. Borde, G. Haïk, V. Watine, L. Pautet: "Really Hard Time Developing Hard Real Time", Workshop Control Architecture of Robots 2007 (CAR'07), 2007.

[7] OMG: *"UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems"*, http://www.omg.org/spec/MARTE/1.0/, 2009.

[8] SAE: "*Architecture Analysis & Design Language (AADL) v2.0*" (proposed draft). Technical report AS5506A, SAE, 2009.

[9] ECSS, *"Space Engineering - Ground Systems and Operations - Telemetry and Telecommand Packet Utilization Standard"*, ECSS, 2003.

[10] A. Burns, B. Dobbing, T. Vardanega, *"Guide for the use of the Ada Ravenscar Profile in high integrity systems"*, University of York Technical Report, 2003.

11. Glossary

CBSE: Component-based Software Engineering

CCM: CORBA Component Model

CCSDS: Consultative Committee for Space Data Systems

D&C: Deployment and Configuration

ECSS: European Cooperation on Space Standardization

ESA: European Space Agency

IDL: Interface Description Language

OMG: Object Management Group

PUS: Packet Utilization Standard

RMA: Rate Monotonic Analysis

SEMS: System Engineering and Middleware based on standards for Space domain

TC: Telecommand

TM: Telemetry