TOWARDS A PATTERN-BASED FRAMEWORK FOR SATELLITE FLIGHT SOFTWARE USING A MODEL-DRIVEN APPROACH

Walter Abrahão dos Santos, walter@dss.inpe.br
National Space Research Institute – INPE
Av. Dos Astronautas, 1758 CEP 12227-010, S. José dos Campos – SP – Brazil

Adilson Marques da Cunha, cunha@ita.br
Aeronautical Institute of Technology – ITA
Praça Mal. Eduardo Gomes, 50 CEP 12228-900 S. José dos Campos – SP – Brazil

Abstract. *This paper proposes the application of a pattern-based framework to guide architectural instantiations for the satellite flight software using a model-driven approach to the problem. Satellite computer systems and their software enhance tremendously the on-board capability, but drive system cost and complexity. Space software projects are prone to last-minute software specification changes and to recurring architectural solutions from one project to another. Both aspects can be tackled with a certain degree of adaptability, which can be mapped into code, and rapidly prototyped with the help of design patterns, and a domain-specific model-driven development approach. The pattern-based framework allows one to capture and document ones main architectural assets.*

Keywords*: Flight Software, Real Time Embedded Systems, Software Reuse, Frameworks, Model-Driven Development.*

1. INTRODUCTION

Software is commonly employed for implementing highly complex functions. The ability and flexibility to deal with them makes software an essential part of increasing demand on artifacts for space programs. Software Engineering practices have been applied in all levels of space technology from on board systems functions to firmware. Spacecraft computer systems and their software enhance tremendously the on board capability, but drive system cost and complexity (Larson and Wertz, 2004).

Each new satellite project has a subset of recurring issues concerning the embedded software development. These issues need to be isolated, understood and incorporated to new projects, enabling reuse. Furthermore, as new artifacts are introduced and re-engineered in the flight software domain for satellites, high complexity levels and increasing autonomy demands are constantly present. Hence the main motivation of this work is to tackle chronic problems on software development that solely object orientation cannot solve. In order to deal with these issues, critical innovations should be considered in four key areas: systematic reuse, development by assembly, model-driven development, and process frameworks (Greenfield *at al*., 2004).

Briefly posed, the main goal of this research is to provide adaptability to space real-time embedded software projects in order to increase reusability as well as coping with unexpected changes.

This paper is organized as follows. Section 2 compares current solutions to the proposed approach. Section 3 introduces the main background topics, space flight software and model-driven development. Section 4 presents key concepts on enabling a pattern-based software development process. Section 5 describes a domain-specific pattern-based framework for space flight software. Section 6 points to future work on this research topic. Finally, Section 7 outlines the main conclusions.

2. RELATED WORK AND PROPOSED APPROACH

In order to deal with complexity, the area of satellite OBS has followed the growth of abstraction over time from early assembly-based subroutines all the way up to object-oriented software frameworks.

Hereafter, some current solutions are presented pointing out to key technologies to cope with the OBS issues mentioned earlier. Unfortunately, each surveyed solution is concerned with a specific OBS aspect, like the attitude and orbit control subsystem (AOCS), thus lacking a systems approach to the underlying problem, which in this research is obtained by employing the Rational Unified Process (RUP) with an object-based MDA solution tailored for real time environments.

The work in *(*Garrido *et al.*, 2000) is focused in satellite software architectural aspects. It is shown that there is a close relationship between the general architecture and the software life cycle model. The paper approaches software architecture by decomposing it in three different components, each one to be developed by a different company.

Software components tailored for the Attitude Control Subsystem at NASA is dealt on (Reid *et al.*, 1998) using object-oriented design for sensor data processing, attitude determination, attitude control, and failure detection.

The object orientation paradigm is explored in (Dos Santos *et al.*, 1997) as an option for the software analysis phase of an INPE´s scientific satellite.

In (OBOSS, 2007) it is shown that frameworks can be built upon purely object-based systems in this case using Ada83 and a later version on Ada95. Nevertheless, some OBS projects have to rely on C/C++ coding. The work deals mainly with telemetry and telecommand services.

In order to enable reuse via design patterns, (Pasetti, 2002) advocates a novel and articulate C++ object-oriented component-based framework approach for the satellite AOCS. The framework was built as a research prototype. It has some performance limitations and cannot be used to generate flight-proven software yet. Finally, it tackles mainly AOCS aspects.

2.1. Proposed Solution

The proposed solution is underpinned on the following points:

- The recurring software architectural aspects present from one satellite project to another will be tackled employing intent specifications (Dos Santos *et al.*, 2006) and a customized domain-specific pattern-based framework (Pasetti, 2002). This solution saves application developers from the inconvenient of inspecting a standard pattern catalogue to identify relevant structures and, if necessary, adapt them to their concerns; and
- A Model-Driven software development approach will be taken in which code is (semi-) automatically generated from Unified Modeling Language (UML) abstract models. The architectural customization process starts from the key quality attributes and ends up with a set of appropriate design patterns from the initial domain-specific pattern-based framework.

3. BACKGROUND

This section introduces the main background topics of this paper, viz. satellites and their flight software, the model-driven approach taken for the software development.

3.1. Satellites and their Flight Software

Briefly posed, a satellite has generally two main parts: (a) the bus or platform, where the main supporting subsystems reside, and (b) the payload, the part that justifies the mission.

A typical satellite bus has a series of supporting subsystems as depicted in Fig. 1. The satellite system is built around a system bus, as shown on Fig. 2, also called the On Board Data Handling (OBDH) bus. The main On Board Computer (OBC) is the bus master. The clients on the system bus are the various satellite subsystems and the payloads. Satellite subsystems may have their own computer and may even have an internal bus. The flight software manages all the spacecraft functionalities into a coordinated fashion.

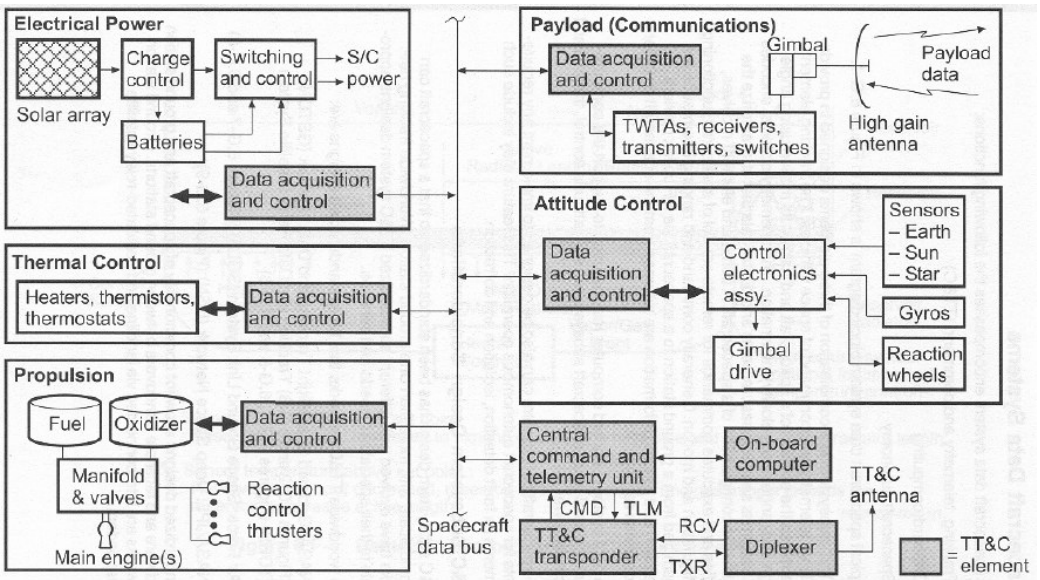

Figure 1. Block diagram of a typical satellite

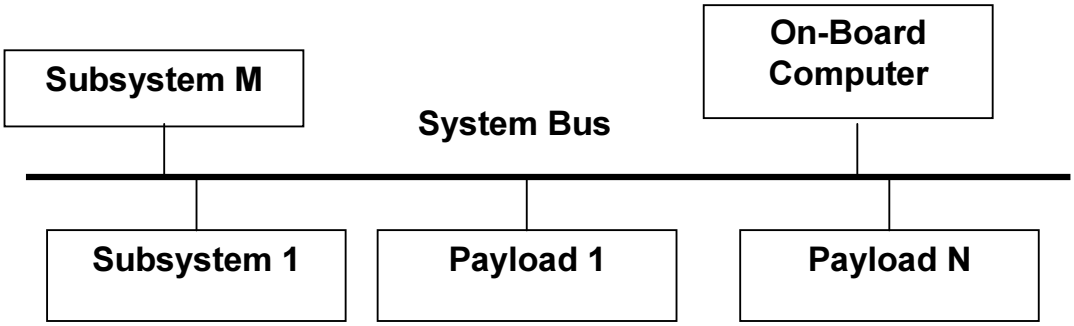

Figure 2. Computational abstraction of a satellite computer system (Pasetti 2002)

The data exchanged over the system bus are telecommands and telemetries. The OBC sends telecommands to the subsystems and payloads, while the subsystems and payloads send telemetries to the OBC. Most normal communications between these components take place over the system bus. Other links do exist for use in special situations such as initialization, power-up, failures or other non-nominal tasks (Larson and Wertz, 2004).

The On Board Software (OBS) is a mission critical on board subsystem that manages all the spacecraft functionalities into a coordinated mission. It enables the satellite to exercise all its states, from ground to on orbit operations, as depicted in Fig. 3.

The OBS is divided into two parts:

- Operating system software – manages computer's resources such as input/output devices, memory, and scheduling of application software. Usually, it will not increase after Critical Design Review (CDR); and
- Application software – mission-specific software that does the work required by the user or the mission, rather than supporting the OBC. It continues to increase as new requirements are brought up and problems encountered.

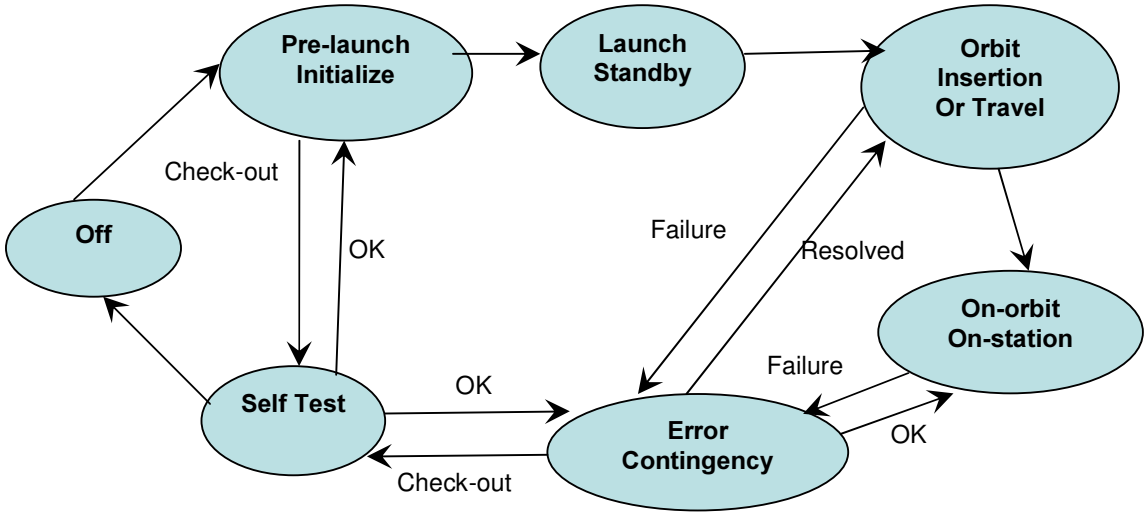

Figure 3. Typical state diagram for On board Computer System, adapted from (Larson and Wertz, 2004)

This paper deals only with the application software, hereafter referred just as OBS, since it is considered where the major issues may be encountered.

In order to reduce complexity of the whole OBS, it will be adopted the abstraction of UML packages as they provide means of decomposing a potentially unmanageable object space into smaller, more-manageable pieces. Fig. 4 identifies the main packages showing the core domains for the OBS and their inter-dependencies (Dos Santos and Cunha, 2005).

In the layered model, software systems are partitioned into layers (physical resources, resources access services, system services, and application). The lower layers are assigned to the prospective on-board processor - ERC32 (ERC32, 2008) and the candidate Real Time Operating System RTOS as the Real Time Multiprocessor Executive System (RTMES) (RTEMS, 2008).

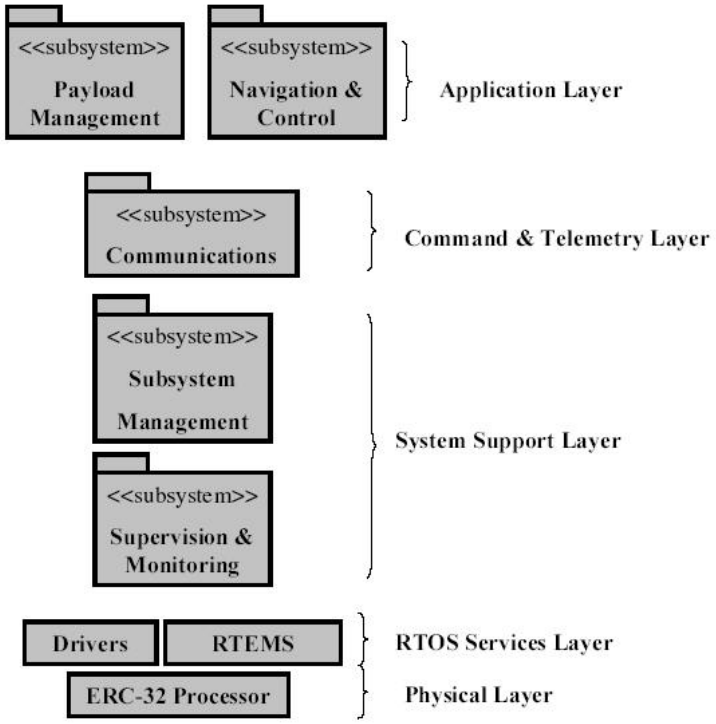

Figure 4. A multi-layered OBS Architecture (Dos Santos and Cunha, 2005)

3.2. Satellites and their Flight Software

At the same time the Object Management Group (OMG) launched the UML 2.0, it introduced its Model-Driven Architecture (MDA) initiative (OMG, 2006) which is a special category of model-driven development. This defines a conceptual framework for a model-driven approach to software development broader than just supporting documentation and high-level design "drafting".

Many software projects today require flexibility to cope with a dynamic environment of requirements, components, technologies, etc. The following features can promote flexibility:

- Visual modeling to decrease the communication gap between systems experts and software analysts/developers;
- Incremental development, even at high-level architectural models, based in the RUP 4+1 architectural views (Kruchten, 2000) for higher quality software and more predictable delivery cycles;
- Round-trip engineering to help maintain the consistency of multiple, changing software artifacts; and
- Model execution where one may find problems and issues that white boarding and document reviews are unable to find.

Flexibility can be attained by exploring the use of MDA as one instance of software engineering automation. MDA is an initiative that considers primary artifacts of software development not the final programs themselves, but rather their underlying models created from modeling languages. Unified Modeling Language (UML) artifacts extended to real time applications are used to create a Platform Independent Model (PIM), which then is mapped by a model compiler into a Platform Specific Model (PSM). As shown in Fig. 5, the development process is defined as a transformation of models, which constitute instances of meta-models defined by domain experts.

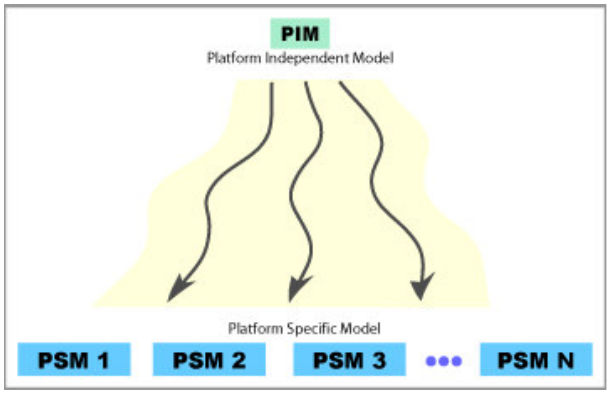

Figure 5. The model-driven approach: PIM to PSM transformation

4. PATTERNS AND FRAMEWORKS

In order to enable a pattern-based software development process as shown in Fig. 6, the following steps are recommended (IBM, 2007):

- **Analyze the target domain** - Identify the context, examine the recurring problem, and find a recurring exemplar solution. A solution to a more complicated problem may involve multiple patterns. Analyze the target domain to get a clear definition of inputs, outputs (what you are trying to produce), and points of variability (what input should be left to the user and what should the pattern provide). Patterns describe things that have been done before, so gather real examples of the deliverables you want to produce and the best practices;
- **Design the recipe** - Determine the best solution to the problem and a natural and realistic entry point for design. Fig. out what information is available at the start and what is needed at each stage. Create an outline of what will happen in each stage;
- **Implement patterns** - At this point you have a set of design activities defined and know the information content at the entry point and exit points. For each pattern there is a pattern implementation, a type of micro-tool that can be created to drive individual pattern applications. An invocation environment needs to be established to provide mechanisms that will configure, trigger, and execute applications of the pattern. A key decision point to the pattern implementation is determining whether output will be presented as a model or as text; and
- **Package and deploy** - Once the pattern implementation is developed, results need to be packaged for distribution, along with documentation suitable for the intended user community. The use of the Reusable Asset Specification (RAS), adopted as an OMG standard in 2004 (OMG, 2007), should be considered in conjunction with the use of a RAS repository.

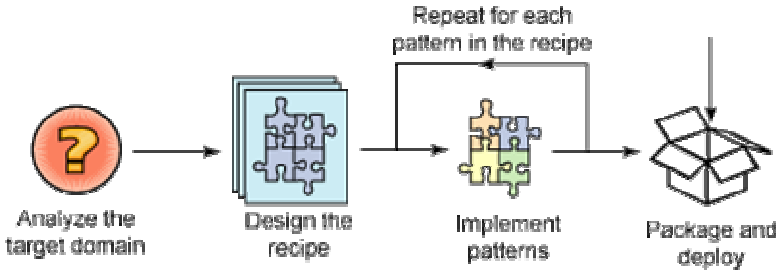

Figure 6. Enabling a pattern-based software development process (IBM, 2007)

An important aspect of patterns development is a standard way to organize, describe, and package assets. RAS defines a standard way to describe and package reusable software assets. A reusable asset provides a solution to a problem for a given context and can have variability points that can be customized by the asset consumer.

A comprehensive list of differences between design patterns and frameworks can be found in (Gamma *et al.*, 1994). Design patterns are the gist of some frameworks. The latter is intended to make architecture reusable and the former encapsulates atomic architectural solutions. Together they aid in the standardization and optimization of approaches to software development.

Standard catalogues of design patterns present design solutions that are generic and which are seldom "ready to use" leaving many implementation and design options at leisure. Conversely, frameworks should instead offer design patterns which, although possibly inspired by standard patterns, have been refined and specialized for use in a specific domain, in this work, the space flight software domain.

Formally a framework is a set of cooperating classes that structures a reusable design for an application-specific domain. Frameworks dictate the architecture of your application as it captures the design decisions that are common to its application domain hence emphasizing design reuse over simply code reuse. This implies that not only applications can be faster built, but the applications have similar structures. Therefore they are easier to maintain, and they seem to be more consistent to their users. On the other hand one loses some creation freedom as many design decisions have been already taken.

The main idea behind frameworks is depicted in Fig. 7 where the non-shaded area represents the architectural backbone shared by all applications in the framework domain and made available to application developers who tailor their needs by plugging into the framework components that implement the application-specific behavior represented by the darker boxes.

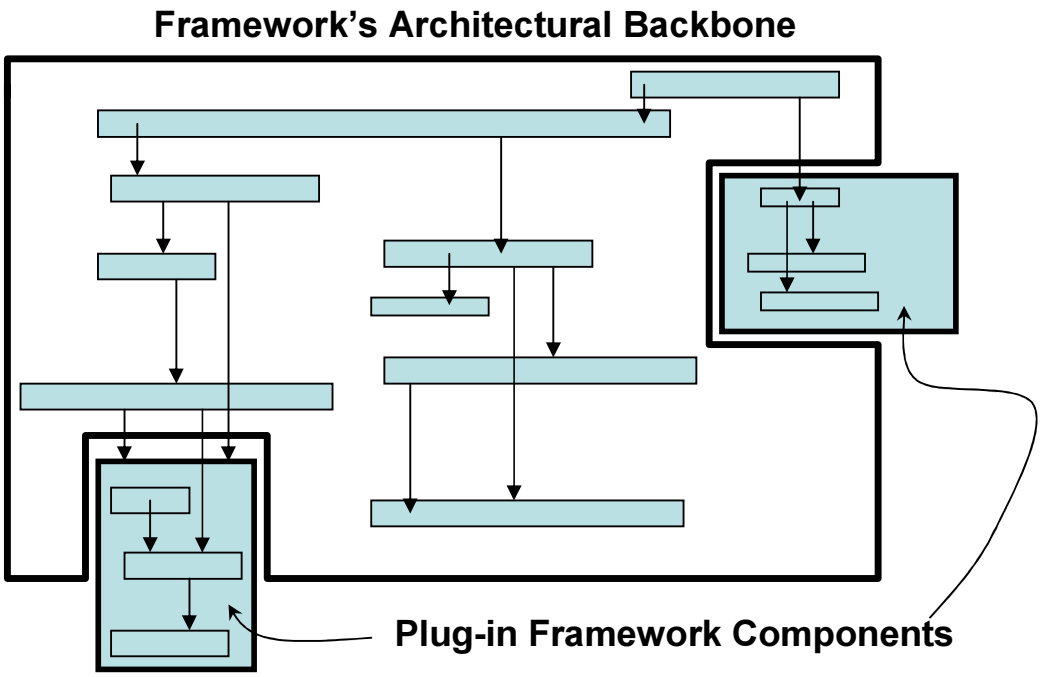

Figure 7. The basic abstraction provide by software frameworks (Pasetti, 2002)

A framework that employs design patterns are far more likely to obtain high levels of design and code reuse than one that does not. Mature frameworks usually incorporate various design patterns (Gamma *et al.*, 1994).

Frameworks allow the investment that is made into designing the architecture of an application to be paid off by making it available across projects and across design teams. Furthermore, direct benefit comes from the documentation of framework by its underlying design patterns since people who are familiar with the patterns can gain insight of the framework.

5. A FRAMEWORK FOR SPACE FLIGHT SOFTWARE APPLICATIONS

This section describes a domain-specific pattern-based framework for space flight software and how it is customized for a particular application.

5.1 A Domain-Specific Pattern-Based Framework

In order to deal with complexity, the area of satellite OBS has followed the growth of abstraction over time from early assembly-based subroutines all the way up to object-oriented software frameworks.

In order to enable reuse via design patterns, (Pasetti, 2002) advocates a novel and articulate C++ object-oriented component-based framework approach for the satellite Attitude and Orbit Control Subsystem (AOCS). The framework,

called AOCS Framework, was built as a research prototype and it tackles mainly aspects of the attitude and orbit control satellite subsystem.

The AOCS framework is organized as a set of 13 design patterns called framelets to simplify the design and the description of a framework by allowing it to be broken up into smaller and simpler entities. This set of design patterns implements the following key functionalities:

- The Telemetry Functionality covering the formatting and dispatching of housekeeping information to the ground station;
- The Telecommand Functionality covering the processing and execution of commands received from the ground station;
- The Failure Detection Functionality covering the implementation of checks to autonomously detect failures in the AOCS software;
- The Failure Recovery Functionality covering the implementation of the corrective measures to counteract the failures reported by the failure detection functionality;
- The Controller Functionality covering the management and implementation of the control algorithms for the control loops operated by the AOCS;
- The Reset Functionality covering the control of the reset functions that bring the AOCS components to some initial default state;
- The Configuration Functionality covering the control of the configuration functions to clear all configuration information in the AOCS components, and to check that all components are correctly configured and ready to start normal operation; and
- The Unit Functionality covering the acquisition of data from and the forwarding of commands to the external units managed by the AOCS (the sensors and the actuators).

Only one of the design patterns will be shown here and further details can be found in (Pasetti, 2002).

5.1.2 An Example of a Design Pattern

In current AOCS systems, telemetry processing is controlled by a so-called telemetry handler that directly collects telemetry data, formats and stores them in a dedicated buffer, and then has them transferred to the ground station. In order to accomplish its task, the telemetry handler needs a complete knowledge of the type and format of the telemetry data: it has to know which data, in which format, and from which objects have to be collected. It is this coupling between telemetry handler and telemetry data that makes the former application-specific and making difficult its reuse (Pasetti, 2002).

The solution to this is based on a design pattern – the telemetry design pattern– that calls for a separation of telemetry management from the implementation of telemetry data collection. This is achieved by endowing selected components in the AOCS software with the capacity of writing themselves to the telemetry stream. Telemetry processing is then seen as the forwarding to the ground of an image of the internal state of some of the AOCS components.

The telemetry design pattern architecture and the pseudo-code of its core component, the telemetry manager, can be both viewed in Fig. 8 and 9.

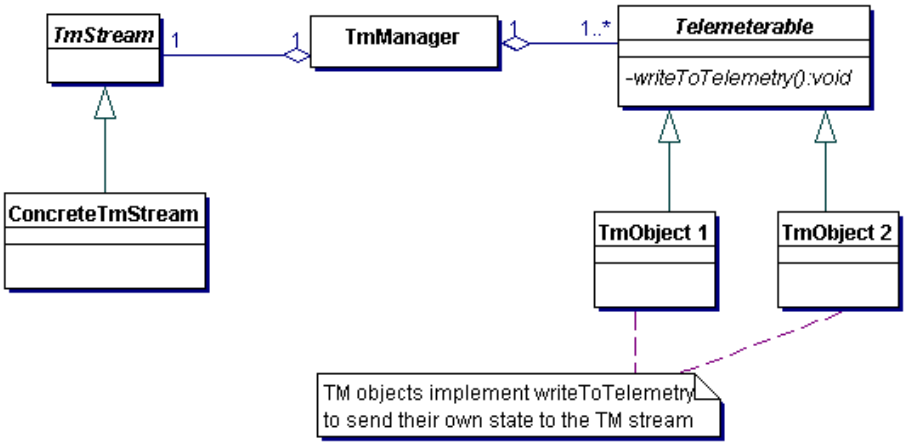

Figure 8. The architecture for a telemetry design pattern adapted from (Pasetti, 2002)

```
Telemeterable*    tmList[N_TM_OBJECTS];
      TelemetryStream*  tmStream;
      . . .
      void activate()  {
        for (all TM objects 't' in tmList)
        { t->setTelemetryFormat(tmFormat);
          tmDataSize=t->getImageLength();
          if (object image fits in TM buffer)
             t->writeToTelemetry(tmStream);
          else
             . . .   // error!
        }
      }
```

Figure 9. The pseudo-code for a telemetry manager component adapted from (Pasetti, 2002)

5.2. How MDA Fits into the Pattern-Based Framework Software Development

Using MDA, systems developers are aided to create UML model constructs having a pattern-based framework as a starting point. The framework is customized to a particular application by creating application-specific subclasses of abstract classes from the framework. Afterwards, developers generate the implementation code, compile, run and debug the application.

Framework designers gamble that its architecture will work for all application in the domain. Any change to the framework's design would reduce its benefits considerably, since the framework's main contribution to an application is its underlying architecture. Hence the employment of the MDA approach as one of the key aspects of this work since it provides a suitable environment for frameworks to be designed as flexible and extensible as possible.

In this work a development-oriented framework will be used to structure the repository of design patterns according to the RUP development process. The envisaged model-driven tool allows the incorporation and definition of frameworks, represented as a set of patterns described by its respective UML models.

In order to achieve project adaptability the following features will be considered: (a) Visual modeling; (b) Incremental MDA development, based in the RUP 4+1 architectural view; (c) Round-trip engineering; and (d) Model execution. In this way, the development process is defined as a transformation of models, which constitute instances of meta-models that are defined by space domain experts.

The MDA definition considered here is the one in which code is (semi-) automatically generated from more abstract Unified Modeling Language - Real Time (UML-RT) models. UML-RT is a dialect of the UML extended for Real Time applications.

A model-driven development tool tailored for real time applications, Rational Rose RealTime® (RRRT) (Rational, 2003), is used as it supports UML-RT constructs for architectural documentation and provides a software development environment that enables Real Time Embedded Systems analysis, design, and implementation.

6. FUTURE WORK

Since the original framework proposed in (Pasetti, 2002) has some performance limitations and cannot be used to generate flight-proven software yet, hence this work intents on using an object-based C implementation. The source code generated by RRRT will be compiled in a Linux environment for hardware-in-the-loop test and simulations.

The tested code will then be cross-compiled and its executable code downloaded to an ERC-32 development kit running the RTEMS RTOS. The ERC32 is the target space-qualified SPARC (Scalar Processor Architecture) processor envisaged at INPE.

This work is currently investigating alternative ways for generating the framework and its underlying patterns either as models or as text-like inputs to the model-driven environment.

7. CONCLUSIONS

The main contribution of this paper is the description of how the problem of satellite flight software can be tackled by using MDA having a pattern-based framework as a starting point. The framework is already available to be

extended. This approach impacts the way space flight software development is presently done using critical innovations in four key areas: systematic reuse, development by assembly, model-driven development, and process frameworks.

In the proposed model-driven development approach, systems developers are aided to create UML model constructs to a particular application from a customized framework by creating application-specific subclasses of abstract classes from the framework. Afterwards, developers generate the implementation code, compile, run and debug the application. This helps to deal with to last-minute software specification changes in which space software projects are prone.

In order to elucidate the framework structure, a single design pattern – the telemetry framelet – was exemplified. The pattern-based framework solution presented may add value to the technological asset as it captures and documents recurring architectural solutions from one project to another.

8. REFERENCES

Dos Santos, W. A., Saturno, M.E., Neri, J. A. C., Bianchi, J., *Object Oriented On-Board Computing: Implications and Perspectives*, Proceedings of the XII Chilean Congress on Electrical Engineering, Temuco, Chile, Nov. 1997.

Dos Santos, W. A., Cunha, A. M., *An MDA Approach for a Multi-Layered Satellite On-Board Software Architecture*, Proceedings of Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA 5), Pittsburgh, USA, Nov. 2005.

Dos Santos, W. A., Yano, E.T., Cunha, A. M. , *Towards Intent Specifications for Safe Reuse in Model-Driven Real Time Software – a Case-Study on Satellite Flight Software*, Proceedings of the The 27th IEEE Real-Time Systems Symposium - Work in Progress Session (RTSS-WiP 2006), Rio de Janeiro, Brazil, December 5-8, 2006.

ERC32 Home Page, http://estec.esa.nl/wsmwww/erc32/erc32.html, 2008.

IBM, *IBM Pattern Solutions*, available online at http://www-128.ibm.com /developerworks/rational/products /patternsolutions/education.html, 2007.

Garrido, B., Alfaro, N., de Miguel, J., García, A., *Software Subsystem and Life Cycle Model for a Small Earth-Observation Satellite*, Proceedings of the DASIA 2000, Montreal, Canada, 2000.

Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Greenfield, J. *et al.*, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, Wiley, 2004.

Kruchten, K. C., *The Rational Unified Process – An Introduction*, 2nd ed.; Addison-Wesley, 2000.

Larson, W. J. and Wertz, J. R., *Space Mission Analysis and Design*, McGraw-Hill, 2004.

OBOSS, *OBOSS Home Page*, available online http://spd-web.terma.com/Projects/ OBOSS/Home_Page/, 2007.

OMG, *MDA - Model Driven Architecture*, available online http://www.omg.org, 2006.

OMG, *RAS – Reusable Asset Specification,* available online http://www.omg.org/technology/ /documents/ formal/ras.htm, 2007.

Pasetti, A., *Software Frameworks and Embedded Control Systems*, LNCS 2231, Springer Verlag, Berlin, 2002.

Rational Software Corp, *Modeling Language Guide - Rational Rose RealTime*, version 2003.06.00, Product Documentation, 2003.

Reid, W. M., Hansell, W., and Phillips T. , *The Implementation of Satellite Attitude Control System Software Using Object Oriented Design*, Proceedings of the 12th AIAA/USU Conference on Small Satellites, 1998.

RTEMS, *RTMES Home Page*, available online http://rtems.com, 2008.

9. RESPONSIBILITY NOTICE

The authors are the only responsible for the material included in this paper.