A JOINT MIDDLEWARE/CONFIGURATION LANGUAGE APPROACH FOR SPACE EMBEDDED SOFTWARE UPDATE

Damien Cailliau(1), Arthur Leger(2), Olivier Marin(3), Bertil Folliot(2, 4)

(1)Observatoire de Meudon, Département d'Etudes Spatiales/CNRS, 5 Place Jules Janssen, 92195 Meudon Cedex, France, Damien.Cailliau@obspm.fr
(2)Laboratoire d'Informatique de Paris 6 / CNRS, Université Pierre et Marie Curie 4 Place Jussieu F-75252 Paris Cedex 05, France, Arthur.Leger@lip6.fr, Bertil.Folliot@lip6.fr
(3)Laboratoire d'Informatique du Havre, 25 rue Philippe Lebon, BP 540 76058 Le Havre cedex, Olivier.Marin@univ-lehavre.fr
(4)Institut National de la Recherche en Informatique et Automatique, Projet Système à Objets Répartis, Domaine de Voluceau, Rocquencourt, 78153 Le Chenay Cedex, France

ABSTRACT

The joint LIP6 - INRIA Rocquencourt Virtual Virtual Machine project and the Observatory of Paris-Meudon associated themselves to propose a reconfigurable space-embedded software platform (PLERS). This is one of the first attempts to define a systematic and provable way to update software embedded in the very constrained space systems. Our approach relies on a reconfigurable execution environment on-board the satellite and a software lifecycle that incorporates software update and maintenance together with software design and development in a common framework. An Architecture Construction Language is defined. It recursively describes a software configuration by expressing the architecture changes to do on a previous version. Compact low level reconfiguration commands are automatically produced. They are uploaded and interpreted by the flight software execution environment.

1. INTRODUCTION

Digital systems embedded on-board scientific spacecrafts have historically evolved from simple command sequencers to real distributed computing systems. One of the reasons for this evolution relies on the use of software: it brings the flexibility needed to implement the desired fault tolerance policy and optimize the instruments sent in space. Indeed software is specifically programmed to provide the requested functionality. It can also interpret operator-provided commands to modify its behavior. Another crucial feature is that it can autonomously adapt itself either by checking some state variables ("if-then") or by using more sophisticated artificial intelligence techniques. Finally, provided that programmable memory is available on-board, software can be modified after the spacecraft has been launched to provide new or improved functionnalities.

In return, this flexibility increases the system complexity. But the more complex the system is, the harder it is to verify and validate its software and the commands sent to control it. In addition, the more complex the software is, the harder it is to get adequate knowledge of its exact state during the run. Consequently, space-fields engineers have to make careful trade-offs between implemented functionality and requested ones in order to mitigate the risks of permanent failure.

Nevertheless, the needs in functionality and flexibility keep on increasing, leading to a strong pressure to make space systems software-dominant systems. The Remote Agent Experiment [13] is a good example of this tendency to send highly flexible software-intensive systems in space. In this case the software was controlled through high level goals and was able to achieve them in most situations through the use of AI-based autonomy. Although very innovative, it was only able to account for hardware failures. One could think that the possibility of updating the software during the mission would facilitate this evolution. But software maintenance has its own drawbacks and brings its own risks.

The PLERS project results of a cooperation between the Observatoire de Paris-Meudon, the computer science laboratory of the University of Paris 6 (LIP6) and the INRIA (National Institute for Research in Computer Science and Automation). Created during the pre-studies phase of the european satellite Corot, it aims at proposing a reconfigurable space embedded platform that would allow easier and safer flight software update. Our key idea is to account for software maintenance from design by proposing an iterative software lifecycle. This study is part of broader work on highly configurable operating systems, the Virtual Virtual Machine approach [7].

This paper presents the result of a the first 2 years of research on this subject. After presenting the various issues of space-embedded software update in section 2., we describe section 3. the execution environment which provides the reconfigurability to the flight system. In section 4. we describe our software maintenance process and the related tools. Last, section 5. presents the conclusions and the perspectives of this work.

2. ISSUES OF FLIGHT SOFTWARE UPDATE

Since the Mariner VI spacecraft, modification of flight software during the mission has become a routine. Many reasons drive those updates. The most obvious one is to correct the spacecraft behavior when a fault occurs because of a software bug, some command errors, or a hardware failure. In addition, due to high cruise times for planetary probes and the diversity of processing needed during the journey and at the targeted location, on-time upload of needed functionality allows (i) better use of time for software development, thus implementation of more mature technologies without delaying the start of the mission, (ii) optimization of memory usage. Besides, as hardware quality and lifetime in space environment increase, uploading new mission software makes possible the use of a spacecraft for new science experiments without the burden of very expensive hardware development and launch. Finally, while science instruments embedded on spacecrafts collect more and more data, the bandwidths available for data transfer to the ground remain constant. More and more processing has to be held on-board to extract and format the scientific information. The raw data is thus completely modified by using theoretical models that have often not been fully validated. This is the case of the european mission Corot. In the same way JPL's REE [10] project investigates the possibility of embedding super-clustering technologies to increase the spacecrafts processing capacities. It is extremely important to be able to adapt at runtime those science processings and the underlying models to achieve the best science results.

Now, some of this software has to participate in critical function such as handling the spacecraft attitude or providing some kind of autonomous control using a knowledge base. In other cases, as for telecommunication satellites, the spacecraft can't suffer any interruption of its activities. Stopping the system to modify the software can dramatically reduce its quality of service. Therefore software update must be low intrusive, that is the unaffected parts of the software must remain active during the modification process.

Classical patching techniques such as those used for Cassini [2], Soho [16] or Minisat01[8], rely on low level intervention by the ground segment (e.g. see [11]). They require long design and validation periods for the new portions of code and for the set of highly detailed commands needed to correctly incorporate them into the flight software. But as software complexity increases, so does the possible modifications. Update in itself become more and more complicated and risky.

We believe that one way to simplify the software maintenance procedure is to have it occur at a higher level, by incorporating it in an iterative cycle including the design and development phases. Instead of working at the binary code level, we consider that the software is built as an aggregate of components linked together through some connectors. The components interface being fixed and known, they can be developed as independent pieces of relocatable binary code. The software has then to incorporate an operating system able to load such components, instantiate some connectors and link them together. Modifications of the application can therefore be expressed as the creation or the removal of such items. In that way software updates only consist on the description of some reconfiguration operations to switch from an old configuration to a desired one. The binary code implementing each component's functionality can be uploaded in a database in the spacecraft memory prior to and independently of the reconfiguration commands.

Many studies have been held to configure distributed applications ([3], [12], [4]). Those solutions all adopt the configuration language approach to describe and install new configurations. Some of them also allow dynamic reconfiguration. References [17] and [9] propose an execution environment and a related tool suite that supports reconfiguration of robotic systems. Although their constraints and frameworks significantly differ from ours, these researches greatly inspired us.

3. BUSARD - A MIDDLEWARE DEDICATED TO HANDLE DYNAMIC RECONFIGURATION

More and more spacecrafts use multitask kernels as software executive: the pre-studies for Corot selected Express Logic's ThreadX as the optimum core for its OS. Most of them offer the same abstractions, i.e. tasks, software timers and communication media (message queues, counting semaphores, binary events). They run some applications built by linking those items together to form some concurrent processing chains. We define our components as tasks, software timers or procedures (relocatable block of code having a unique entry point and viewed as a functional unit). The communication media are our connectors (see [1] for details on this granularity).

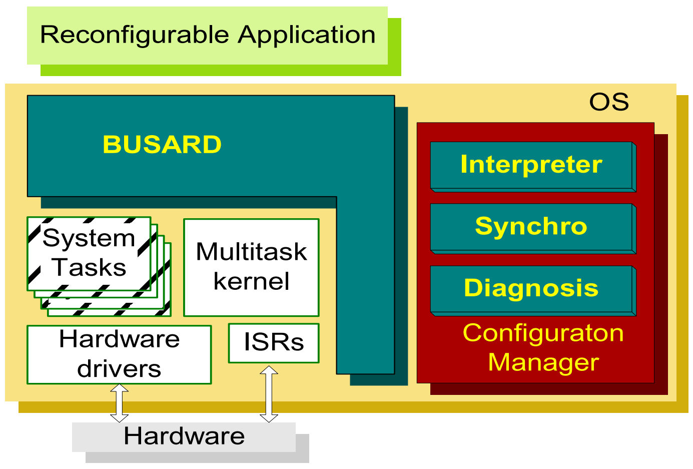

Fig. 1. BUSARD and the execution environment

Most of the COTS kernels don't offer the independence between those software items, necessary for efficient and low intrusive reconfiguration. Neither do they offer methods to control the application execution (runtime item creation, destruction, link, task stop, start and synchronization, etc.). To provide a generic solution and bypass those issues, we developed a middleware, BUSARD (literally Harrier, french acronym for "dedicated software bus handling reconfigurable application"). It virtualizes a minimum operating system, at least including the selected multitask kernel and the hardware drivers, and provides mechanisms to handle the component/connector approach and the dynamic reconfiguration (Fig. 1.). It is composed of a set of proxies linking the items together and providing the necessary independence. It offers two interfaces as a set of C-written primitives. The first implements the ports of the components by providing access services to the connectors. The underlying kernel is invoked through this API. The second one offers methods to manage the configuration and achieve the reconfigurations. In addition the application communicates with the OS and the associated hardware (the sensors and actuators) through specific, permanent, BUSAR-provided connectors.

The modified chains must be stopped during the reconfiguration. The remaining parts of the software can continue their execution, provided that the frontier tasks handle the data unavailability. BUSARD trashes all the data in transit in the reconfigured chains. At the end of the modification the chains have to be resumed at a date defined by the operator for synchronization with the application.

BUSARD has been ported to ThreadX and shows very good performance: the additional overhead due to the middleware on the kernel calls is not greater than 10% (60 cycles), and the memory needed for its code is 2.5 kilo-words (30% of the size of ThreadX). In the same way the amount of data needed by the middleware is negligible.

4. THE SOFTWARE MAINTENANCE PROCESS

Fig. 2. presents the maintenance procedure and the tools used to achieve it. Each single reconfiguration operation corresponds to a composition of some of BUSARD primitive calls. To optimize the communication link, each operation is bytecoded to form a command of several binary words reduced to the strictly necessary information. These commands are decoded and executed by a specific interpreter, incoporated in the on-board Configuration Manager. We identified 58 different commands, composed of at most eight 32-bits words (the format of our CPU), called *bytecodes*. Each command is composed of an identifier (the *opcode*) and several specific parameters. For the interpreter to be able to verify the correctness of the fetched instruction, each of its bytecode possess a type identifier, stored in its 4 uppermost bits, the remaining 28 bits being affected to the associated information.

These commands have to be uploaded and interpreted following a logical sequence. Since a configuration is a composition of a finite number of components, we describe it using a custom high level textual language, CURL (for Corot Unified Reconfiguration Language).This description is sufficient to uniquely represent a configuration, install it and incrementally modify it (i.e. reconfigure). The use of a language allows to verify the coherence of the instruction sequence before emission. CURL unifies the ADLs (Architecture Description Language) the (R)CLs ((Re)Configuration Language) in that it expresses the high level operations to upgrade a starting configuration to a desired one. We name such a language Architecture Construction Language (ACL). Therefore, a unique language recursively describes a configuration, and provides the deployment operation and the dynamic software update commands. A script written in CURL starts with a declarative part, either referencing the initial configuration for a reconfiguration script, or defining the invariant connectors reflecting the OS access. It also holds the definition of the characteristics of all the needed components. Follows an imperative part. It presents the succession of instructions to load/remove the components, instantiate/destroy the connectors, link them together and control the execution of the application.

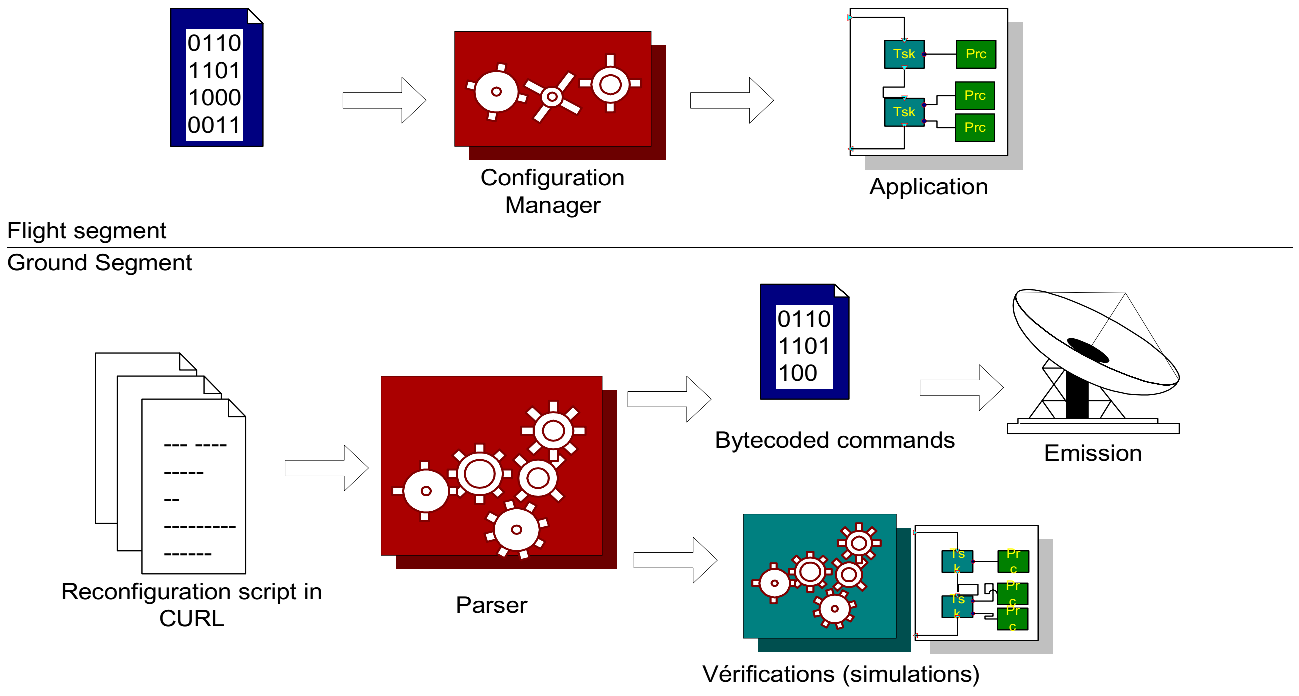

Fig. 2. Maintenance process

PsyCoPaT (for Corot Parser Tool, a rather long way for an acronym) compiles such a script and produces the sequence of bytecoded commands to be uploaded to the interpreter. It performs some few low level (syntactical and grammatical) verifications and produces an object representation of the configuration defined. Such a graph could be used in a structural verification tool that would ascertain the relevance of the configuration according to some fixed rules. Those rules are of the type: "all the tasks shall be linked together to form processing chains", "all the processing chains shall be linked to at least one input and one output persistent communication media", etc.

The Configuration Manager (Fig. 1.) produces a diagnosis of the update operation, first by echoing each BUSARD primitive called and its result, and second by gathering structural information about the installed configuration. This diagnosis is sent to the ground for further analysis by the operator. In the case something goes wrong during the procedure, the Configuration Manager stops the execution of the received commands and installs a pre-defined configuration to enter a safe mode. This default configuration is also defined as a series of commands carefully validated and stored in memory. Last, the Configuration Manager is responsible for the synchronization of the application: it starts the modified chains according to the temporal pattern provided with the command sequences.

We developed a reconfiguration simulator as a first step to validate CURL. It mocks up BUSARD and the Configuration Manager by providing the same API and behavior. This simulator uses the configuration graphs produced by psyCoPaT completed by a description of each component's temporal behavior. Thus, by injecting stimuli in the system, we can simulate the temporal behavior of a whole application. Then, at a given time, a sequence of bytecodes is provided to the simulator and its interpretation merges in the application simulation. This gives some temporal traces of the execution of a configuration and of a reconfiguration. Such a tool can be used to quickly evaluate whether a given configuration will respect the system time constraints and if it remains so during the reconfiguration process.

To validate the whole maintenance process, we developed a fully functional simulator of the embedded system, CHESS, for Corot Heavenly Embedded Software Simulator (see [15] for a discussion on simulation for space systems). It provides a mock-up of an embedded minimum operating system using a Windows version of ThreadX. Having the same interface as the final target's version, few modifications had to be brought to BUSARD and the Configuration Manager. The system inputs are simulated by automatically fetching data to the invariant connectors. The whole system communicates with a mock-up of the ground segment through some UDP links.

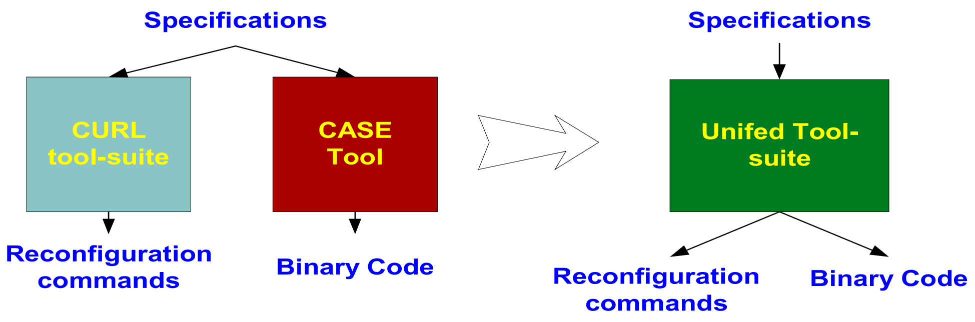

Fig. 3. Towards a unified software development and maintenance tool-suite

5. TOWARDS A NEW FLIGHT SOFTWARE LIFECYCLE

In this paper we proposed a new approach for flight software update. It is based on (i) the use of an Architecture Construction Language (CURL) and the associated tool suite, and (ii) the use of a specific lightweight execution environment onboard the spacecraft. Unifying ADL and (re)configuration languages, the ACL provides a unique framework to: (i) recursively design an application following the software specifications and, (ii) automatically define the deployment or reconfiguration operations needed to install each version of the software on a physical target. The lightweight execution environment embedded on the target provides access to the hardware and all the basic services for dynamic application construction and update through a generic interface. It also hides whichever mission-specific feature has been defined to be permanent (un-modifiable). It handles the applications as a set of components linked together through their ports by some connectors.

Using this approach, software maintenance, or software update, becomes part of the software development lifecycle. The design team defines the application architecture by describing which modifications need to be held on the existing one. The starting configuration can be either the empty configuration, an intermediate prototype version, or an obsolete final image of the software. This description yields to the creation of a set of low level commands that the execution environment on the target uses to build the desired application. The components are developed independently of any specific configuration, with the only constraint that their code has to be fully relocatable: every internal reference has to be PC-relative and the external references are limited to OS calls. If their addresses are fixed and known, the references to the OS primitives can be resolved by the tool suite. If not, the component loader (one of the services of the OS) does this job when the component's binary is loaded in RAM for execution.

Therefore, the components binary code and the reconfiguration commands automatically produced form the ACL script can be used to build the application at every stage of the software lifecycle, from early prototype on a simulated environment, to intermediate prototype on a physical target, to software update during the mission. This software lifecycle thus become fully iterative with automated deployment. Besides, as proposed by JPL's Mission Data System [5] [6], such a process yields to a software development and control architecture integrating the ground and the flight segment.

We advocate that such an approach greatly simplifies not only software maintenance, but also software development. By designing software to be reconfigurable and hiding the related details in the development process, it also makes software update safer. We believe that the complexity and the functionality of flight software could greatly increase by applying those principles. This is especially true for the scientific payloads which could benefit from more sophisticated instruments and experiments by embedding more processings and therefore limiting the impact of reduced up/downlink data rates.

As shown Fig. 3., one of the next step of this research would be to merge those concepts in commonly used CASE tools (e.g. UML-based software development processes) to provide a unified software development and maintenance tool-suite. Other possible improvements would be to enrich our framework to handle more sophisticated components and distributed computing, for instance, adding some new connector types to allow true objects interaction and synchronization (e.g. light-weight ORB-like connectors). Last, it would be necessary to develop more sophisticated validation and verification tools, together with optimization tools, to fully benefit from the formalism provided by the ACL.

6. ACKNOWLEDGMENT

We would like to thank Dr. D. Dvorak, Dr. N. Rouquette and Mr. S. M. Krasner from JPL (Jet Propulsion Laboratory) for very valuable thought, ideas and advises that greatly helped us to start this study. This work has been partially financed by the LIP6 project PLERS.

7. REFERENCES

[1] Cailliau D., Bellenger R. " The Corot instrument's software: towards intrinsically reconfigurable real-time embedded processing software in space-borne instruments", Proc. of the 4th IEEE International Symposium on High Assurance System Engineering (HASE'99), Washington, DC., p. 75--80, Nov. 17-19, 1999.

[2] Brown G. M., et al. " Storing and loading the flight software for Cassini's Attitude and Articulation Control Subsystem: a fault tolerant appraoch", Proc. of the 15th AIAA/IEEE Digital Avionics Systems Conference, Oct., 1996.

[3] Chung E. et al. -- Dcom and corba side by side, step by step, and layer by layer -- C++ report, Vol. 10, No. 1, p 18-30, 1998.

[4] De Palma N., et al.. " Dynamic Reconfiguration of Agent-Based Application", Proc. of the 3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal, 23-28 April 1999.

[5] Dvorak D., et al. " Software architecture themes in JPL's Mission Data System", Proc. of 2000 IEEE Aerospace Conference, Big Sky, MO, March, 2000.

[6] Elson A. "Software lifecycle themes for JPL Mission Data System (MDS) project", Proc. AIAA Space Technology Conference and Exposition, Sept 28-30, 1999, Albuquerque, NM.

[7] Folliot B. "The Virtual Virtual Machine Project", Invited talk at the Simposio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC'2000), Brasil, October 2000.

[8] Garrido B. et al. "MINISAT01 on-board software maintenance", Proc conference on Data Systems µIn Aerospace (DASIA98), p65-69, Athens, Greece, 25-28 May, 1998.

[9] Gertz M. W., Stewart D. B., Kholsa P. K. "A human-macine interface for reconfigurable sensor-based control systems", Proc. of the AIAA Conf. on Space Programs and Technologies, Huntsville, Al., Sept., 1993.

[10] Katz D. S., Springer P. L. "Development of a Spaceborne Embedded Cluster", Proc. of the IEEE International Conference on Cluster Computing (CLUSTER2000), Technische University Chemnitz, Saxony, Germany, Nov. 28 - Dec 02, 2000.

[11] Martinelli A., Torchia F. "The software maintenance in the BeppoSax scientific mission", Proc. of the Data Systems in Aerospace Conference (DASIA'97), Sevilla, Spain, p.369--374, May 26-29, 1997.

[12] Magee J., Kramer J., Sloman M. "Constructing distributed systems in Conic", IEEE Trans. on Software Engineering, Vol. 15, No. 6, pp 663-675, Jun. 1989.

[13] Muscettola N. et al. "Remote Agent : to boldly go where no AI system has gone before", Artificial Intelligence, Vol. 103, No. 1-2, p 5-48, 1998.

[14] Paoli F., et al. " COROT: a small satellite in low earth orbit for asteroseismology and the search for exoplanets", Proc. 15th International Astronautical Congress, Amsteram, Ned., Oct. 4-8, 1999.

[15] Reinholtz K. "Applying simulation to the development of spacecraft flight software", Proc. of the IEEE Aerospace Conference, Snomass, CO, Mar. 6-13, 1998.

[16] Stevens J. S., Johnson G. L. R. "Updating the Soho AOCS ACU onboard software", Proc. of the Conference on Data Systems In Aerospace (DASIA2000), p347-352, Montreal, Canada, 26-28 May, 2000

[17] Stewart D. B., Schmitz D. E., Kholsa P. K. "The Chimera II real-time operating system for advanced sensor-based control applications", IEEE Trans. on Systems, Man, and Cybernetics, Vol. 22, No. 6, 1992.