Platform for Software Reconfiguration in Embedded Systems

Arthur Léger[1], Bertil Folliot[1], Damien Cailliau[2]
email: Arthur.Leger@lip6.fr, Bertil.Folliot@lip6.fr, Damien.Cailliau@obspm.fr

[1] Laboratoire d'Informatique de Paris 6 / CNRS
Université Pierre et Marie Curie

[2] Département d'études spatiales / CNRS
Observatoire de Meudon

position paper

Abstract

The software on quite all embedded systems (from smart cards to satellites) is difficult to change once it has left the factory. It is then difficult to modify their functionalities, to add new ones or to correct hardware or software bugs. Moreover, the constraints specific to the embedded system (on memory, CPU, power, communications and so on) make reconfiguration even more difficult. We propose in this paper a software reconfiguration platform, which is general enough to take into account a wide variety of constraints, and which is flexible and adaptable enough to be specific to a given embedded application.

I. Introduction

THE software on quite all embedded systems (ranging from smart cards and PDAs to satellites) is difficult to change once it has left the factory. It is then difficult to modify their functionalities or to add new ones (as could be needed in set-top-boxes) or to load new software to correct software or hardware faults (for instance when a radiation burst damages one of the sensors of a satellite). What is usually done is modifying the software off-line, then stopping the application, overwriting the entire code segment (or just diffs) in the memory of the system and re-starting it. This is known as the "patch" technique. It strongly limits the possibility of reconfiguration and the safety of such reconfigurations (for instance, there is no way to prevent the embedded system from drifting in an irreversible manner from its nominal behaviour), and it is mostly used in emergency situations to react to unpredicted behaviour. This may be acceptable for a smart card but could have dramatic consequences both from a scientific and economical point of view for a satellite or it could even be worse in human life critical applications.

We believe that there exists a whole family of solutions which must be derived for each application domain (smart cards, satellites, human critical systems...) or even to each particular system depending on its seriousness.

We propose in this paper a methodology for building reconfigurable software for embedded systems that fits to the application domain. This methodology relies on 3 principles:

- software encapsulation and decomposition by the means of proxies;
- a language (that may be dedicated to the particular domain) to express an initial configuration, a reconfiguration and the constraints which must be respected during the reconfiguration phase (eg. memory footprint, time, communication bandwidth or power consumption);
- and a reconfiguration execution engine (which executes on the running application a reconfiguration expressed within this language).

Encapsulation and decomposition allows reconfiguration to integrate into existing business software tools and specialists know-how, and also permits reconfigurations to be safer.

The reconfiguration language provides a high level of abstraction to the architects in charge of reconfiguring the software by mapping the design-time structure of the application onto the running software. A verification tool can then interpret this language in order to verify formal properties

automatically and to test several reconfigurations by simulation before choosing an appropriate one to commit.

The reconfiguration execution engine executes a reconfiguration script onto the running application onboard the embedded system. This task is often too complex and resource consuming for the embedded system itself. Thus, the reconfiguration execution engine is "distributed" between the ground base and the embedded system: the ground-based part compiles the high-level reconfiguration script into a low-level bytecode. This reduces both bandwidth consumption and the complexity of the embedded part.

This paper presents a methodology for software reconfiguration in embedded systems. It tries to generalize the concepts introduced in the PLERS prototype [FOL00][CAI01a][CAI01b] which was designed based on the contraints of the French satellite Corot [AUV99]. It is organized as follows: Section 2 describes the various constraints and difficulties in reconfiguring constrained embedded software, Section 3 gives an overall description of our reconfiguration methodology, Section 4 describes the reconfiguration language and the reconfiguration execution engine, Section 5 compares our approach with existing works. Section 6 presents conclusions and perspectives.

II. Constrained Embedded Software and Reconfiguration

In order to make reconfiguration of embedded software possible, not only must we face the inherent problems of software reconfiguration, but we must also deal with the constraints of the embedded world. These constraints can be roughly classified into four categories:

- scarcity of onboard resources,
- weakness of the communication link,
- real-time and other domain-specific constraints, and
- work habits and methods of the application domain specialists.

For economical (or technical) reasons, embedded systems are very poor in resources. For small systems which are produced in great quantities, the slightest reduction in the production cost of each unit becomes huge when millions of items are to be produced. For expensive systems such as satellites, where very few units are produced, one could think that increasing onboard resources is not an issue. However, the hardware for space missions must be specifically treated to support radiations and this is a very long and expensive process.

The main three resources which embedded systems are usually scarce on are: memory, CPU and power consumption. Scarcity of power is may be the most specific constraint of embeded systems. It expresses at several levels. First, the average power consumption is limited. For instance, on a spacecraft, electric power consumption cannot exceed what can be produced by the solar panels. Moreover, instantaneous power consumption is also limited. This means that onboard devices cannot all be active at the same time: to turn one of them on, it may be necessary to turn another one off. To end with, some sources of energy cannot be renewed during the whole mission. For instance, this is the case of fuel for a deep space probe.

These constraints possibly modify the reconfiguration itself. The scarcity of memory affects reconfiguration at three levels: the amount of onboard code necessary to execute the reconfiguration orders, the overhead induced on the application code by its capacity of being reconfigured and the size of a typical set of reconfiguration orders. The scarcity of CPU affects the code to execute the reconfiguration and the overhead induced on the application code. And the scarcity of power affects the way a given reconfiguration may be executed in the same manner as it already does for the application code.

The communication link between the ground base and the embedded system suffers some several weaknesses. These are the same ones as in the world of "usual" distributed systems: low bandwidth, long and variable delays, discontinuous connection and possible unreliability of transmission. But they can be way more serious. For instance, in the case of a deep space mission, delays can grow up to several hours (at light-speed, as for DeepSpaceOne) or total available bandwidth may be limited to 200Kb a day (because of intermittent visibility, as for Corot). At the opposite, a smart card can be deconnected for an arbitrary long period of time. Thus, the execution of a reconfiguration cannot always be driven with human-in-the-loop interactivity and the size of a typical set of reconfiguration commands should be compatible with the available bandwidth in order to be sent in little time (and if possible in one connection).

Depending on the application domain of each particular system, some other constraints can appear. Real-time is one of them: some systems must perform some operations before certain deadlines or at very precise dates even while other tasks of the same system are being reconfigured. For instance, reconfiguring an algorithm should preferably not disturb the task of keeping track of the embedded system's course (attitude recovery takes several days for Corot).

For each specific application domain of embedded systems, software architects have developped over the years knowledge, methods, tools and code. Reconfiguration cannot sensibly make them unusable and should integrate coherently within them.

III. Reconfiguration Process Overview

As stated above, our methodology relies on three main principles: software decomposition by the means of proxies, a high level language for expressing reconfigurations and their constraints and a reconfiguration execution engine.

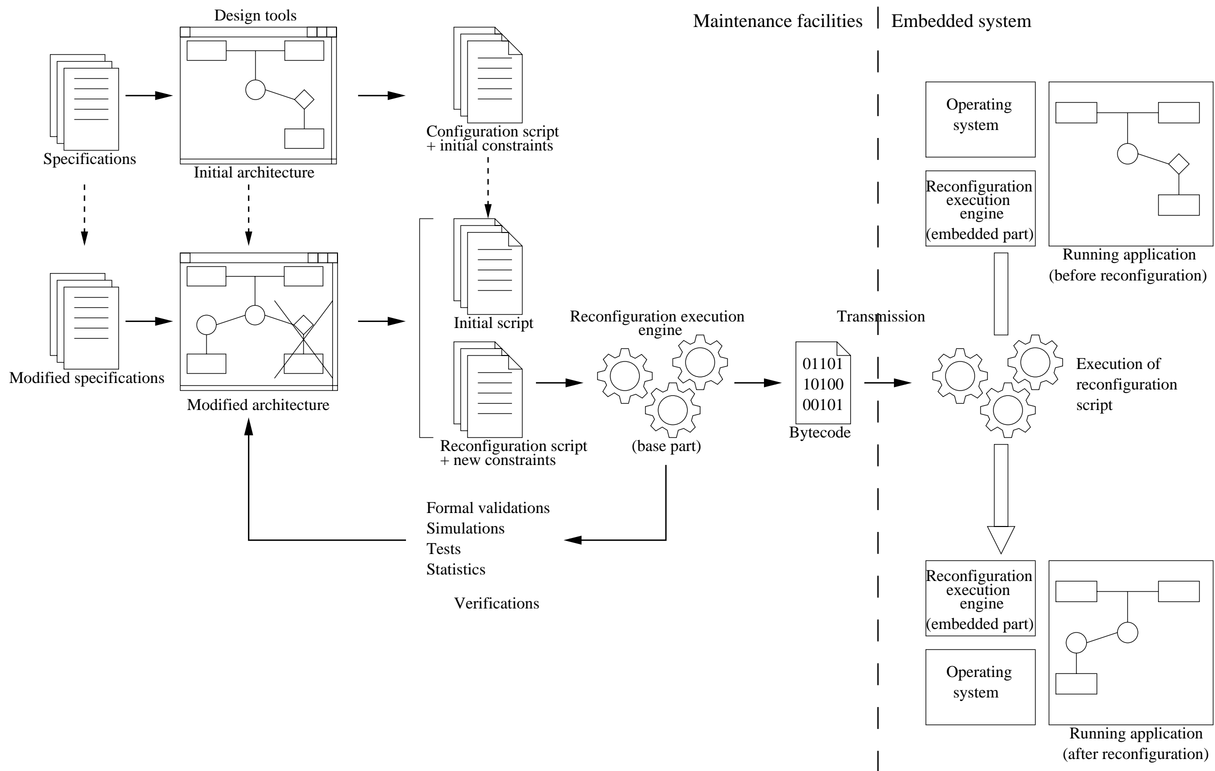

Fig. 1. **Reconfiguration process overview.** To reconfigure the system, the new architecture is defined within the same design tools as usual. The corresponding reconfiguration script must then be written. This script is executed by the maintenance-facilities part of the reconfiguration execution engine. When the verifications produced are satisfactory, the bytecode may be transmitted to the embedded part of the reconfiguration execution engine.

The general process of reconfiguring the application onboard an embedded system results from the necessity of changing the application's behaviour and consists in several steps:

1) informal description of the needed behaviour,
2) modification of the high-level description of the application in the design-time formalism (eg. UML),
3) determination of a set of low-level reconfiguration orders to apply on the running code,
4) testing of these orders,
5) transmission to the system,
6) execution of these orders on the embedded system, and

7) transmission of a diagnostic.

The aim of our solution is to keep the design-time structure of the application in the running code. The code is not monolithic any more, but is composed of smaller pieces - each one corresponding to one item of the high-level description - which are bound together by the means of proxies. Our prototype language (*see sec. IV*) expresses a reconfiguration as a set of bind modifications and items loading/un-loading. The reconfiguration execution engine executes these orders directly on the proxies and on the pieces of code.

The strength of our approach resides in making the reconfiguration expressed at the same level of abstraction as the design-time description of the application. Thus:

- there is only one formalism to manipulate,
- the time-consuming and error-prone step of hand-writing a sequence of low-level reconfiguration operations is eliminated,
- some properties' formal verifications are possible,
- extensive testing is possible (since scripts are easy to write), and
- diagnostic is smaller while more expressive (structured description of the items and of the binds instead of a whole dump of the binary code...).

IV. Reconfiguration Language & Reconfiguration Execution Engine

Some practical problems bring complexity to the general process presented in the previous section.

First, the design-time formalism varies with the application domain of the embedded system and with the company's culture. Since the reconfiguration language manipulates the same concepts and interacts with this design-time architecture description language, it depends on it. Two main options may be considered: either the reconfiguration language is derived into a new flavor adapted for each context, or the reconfiguration language is generic enough in order to be independent from the context. This point still is under investigation.

Moreover, a reconfiguration is an intrusive process which doesn't necessarily respect the application's normal stream of computing, since it is precisely intended at changing the behaviour of the application, and hence will break the continuity of treatment. Thus, any reconfiguration cannot be done anyhow at anytime: some constraints must be respected regarding the time and way a reconfiguration occurs. These constraints highly depend on the functional semantics of the application and even on its precise architecture. These constraints cannot be defined but by the specialists in charge of maintaining the application and will be expressed within the reconfiguration language.

Last, the size of a typical text-encoded reconfiguration script and, above all, the complexity of the task of interpreting such a script and of checking whether the specified constraints are respected can often not be done on board. Thus, the reconfiguration execution engine should be distributed between the maintenance facilities and the embedded system itself. In other words, the computation-intensive generation of a sequence of low-level reconfiguration orders implementing the specified reconfiguration and verifying all the constraints in the high-level reconfiguration script is done on the ground. The generated procedural sequence is then bytecoded for transmission to the embedded system. The work of the onboard part of the reconfiguration execution engine is then reduced to the straightforward execution of this low-level bytecode. This drastically downsizes the code which must be embedded onboard the system in order to handle the reconfiguration phase.

The semantics of the reconfiguration language is threefold.

i) First, it is used to describe each component of the application. This description includes all the properties which are necessary and sufficient in order to manipulate this component. For instance, the description of an end computation component like a procedure includes the size of the compiled code (necessary for memory management of the component), the prototype for calling it and maybe the worst-case execution time if there are real-time concerns for this application.
ii) Obviously, the reconfiguration language also enables description of the architecture of the application: the items used in the configuration (references to their above-mentioned descriptions), their state in this configuration (for instance, the base address at which the code of a procedure is loaded) and the binds between these items. To be more precise, what is described is rather

the loading/deletion of items, the modification of their state (as the modification of the priority of a task is a kind of reconfiguration) and their linking and un-linking[1].

iii) Finally, this language is used to describe the constraints which affect the reconfiguration of the system. For instance, a constraint can be the global duration of the reconfiguration phase or the maximum memory overhead. Or a constraint could even be the fact that the reconfiguration of a particular procedure must occur between the treatment of data packets of two precise types in order to enforce consistency.

We consider that this language must be as flexible as possible regarding the spreading of a reconfiguration description over several files. Thus, the description of a reconfiguration is itself decomposed into smaller pieces and can respect existing rules of project management. For instance, the description of a reusable component is very likely to be found in a dedicated file together with other formal specifications of its behaviour, its source code or its compiled form. In the same manner, the constraints which must be respected by the reconfiguration could be gathered in a separate file for direct use by a formal checker. Besides, for an application containing a great number of components or constraints, it would violate every conception methodology to put everything in the same huge monolithic file.

The reason why the constraints expressed within the reconfiguration language are so low-level comes from the genericity of this language. What must be kept in mind is that our methodology is not dedicated to any particular application domain. At the opposite, the high-level constraints on the application behaviour and the low-level constraints they induce on the software components come directly from the application domain and the precise system. For instance, the same high-level constraint "temperature must remain low" could be derived into "only one actuator can be active at the same time" for a system and into "reconfiguration must be accomplished through small periods of activity separated by long periods of inactivity" for another one. Thus, this translation cannot be done automatically and we think that describing it in the reconfiguration language would make it uselessly more complex.

V. Related Work

The need for some kinds of software reconfiguration in constrained embedded systems is not new. Some solutions have already been proposed and even applied.

The Chimera / Onika solution [STE92] supports dynamic reconfiguration of component-based applications. Nevertheless, it is rather confined to the specific domain of reconfigurable-hardware robotics and all possible reconfigurations must be planned at design-time and statically defined on the system.

The Software Maintenance Facility developped for the Beppo Sax mission [MAR97] provides an update environment including test and simulation tools of the onboard software. A reconfiguration needs the system to be stoped and is human-in-the-loop oriented.

The onboard fail-safe software of the Cassini mission [BRO96] shows a somewhat novative approach for spacecrafts. This backup software is reduced to a "boot-loader" which re-initializes the system with an image of the software from persistent storage which can be patched or reloaded from the ground as well as the "normal" software. Even though this whole technique was defined at design-time and deeply rationalizes the update process, it still shows some weaknesses. Updates are limited to binary patches and it is fundamentally *stop-patch-restart* oriented.

The work done on Remote Agent [MUS98] is probably the most ambitious change in the spacecraft domain. It was aimed at building an autonomous system which could be adapted to modifications of the mission or to hardware faults. The entire software is goal-based and is issued from AI techniques. A reconfiguration of the behaviour is thus obtained through the sending of a new set of goals to pursue. The main drawback is the resource consumption which is incompatible with most embedded systems. Furthermore, there is no real modification of the software and especially the goal-evaluation model cannot be changed.

Outside the range of embedded systems, several works have been achieved in the domain of distributed software reconfiguration over workstations networks.

[1]The possibility of using the same language for expressing a static configuration relies on an abstraction: a configuration is considered as a *re*-configuration from the *empty* configuration. The *empty* configuration is the unique configuration which is reduced to the bare system without any reconfigurable component: only the part which must not be reconfigured and could even reside in ROM.

Olan [BAL98] is an interesting solution since it permits to combine heterogeneous pieces of software into a uniform component-based model architecture. The strengths are to provide a middleware for managing different kinds of communication semantics and a Configuration Machine which deploys the application onto the network according to high-level deployment constraints. Though Olan doesn't handle reconfiguration of an already deployed application.

Conic [MAG89] handles dynamic reconfiguration of component-based applications. Moreover, reconfigurations are expressed in a declarative language, relying on a reconfiguration *daemon* for determination and execution of a consistent sequence of low-level reconfigurations. However, the complexity of this daemon and the code overhead induced on each agent of the application do not seem to be applicable onboard the embedded systems considered in this paper.

VI. Conclusions and Perspectives

Many embedded systems share the need for modification of behaviour without being called back to factory (or even without physical intervention), often for changing functionalities not known in advance, and sometimes after several years of service. But each one of these applications needs its own method of reconfiguration which would fit its particular constraints.

In this paper, we propose both a methodology for developping reconfigurable software and the general maintenance chain ranging from the language for specifying the reconfiguration to occur down to the software embedded onboard the system dedicated to the execution of this reconfiguration. The strength of our approach resides in our constant concern of integration within the existing design/development/test tools and specialists know-how.

This work generalizes the experience on the PLERS project and benefits from its first experiments. PLERS originated as part of a European satellite mission pre-studies for which software reconfiguration services are a necessity because its embedded data extraction algorithms rely on theoretical hypothesis. A full chain including language, compiler, simulator and interpertor are being developped for Corot and show the interest of our approach in this case.

The main efforts are currently directed towards one goal: real-case studies. This involves dynamic reconfiguration experiments onto different real applications conforming to the exact constraints to which these applications are supposed to abide, using dedicated flavors of our tools and language. The next step will then be to gather and classify the constraints for different domains and design formalisms, and try to extract a common core and a way to derive the language and tools for each case.

References

[FOL00] B. Folliot, D. Cailliau, I. Piumarta, R. Bellenger - "PLERS: Plateforme Logicielle Reconfigurable pour Satellites" In *Proc. 12th RENPAR*, Besançon, France, June 19-22 2000, pp 191-196

[CAI01a] D. Cailliau, A. Léger, O. Marin, B. Folliot - "A Joint Middleware/Configuration Language Approach for Space Embedded Software Update" To appear in *DASIA'2001*, Nice, France, May 28 - June 1 2001

[CAI01b] D. Cailliau, A. Léger, O. Marin, B. Folliot - "Conception d'un Système de reconfiguration de logiciels multitâches embarqués sur satellites" To appear in *CFSE'2*, Paris, France, Apr 24-26

[AUV99] M. Auvergne, A. Baglin, et al. - "Du coeur des étoiles aux planètes habitables, les enjeux de Corot" In *Journal des Astronomes Français*, No. 60, pp 27-34, 1999

[OLI99] J. P. Olive - "Soho Recovery" In *Proc Flight Mechanics Conference*, 1999

[BRO96] G. M. Brown, J. C. Hackney, Dr. R. D. Rasmussen, K. Zarnegar - "Storing and Loading the Flight Software for Cassini's Attitude and Articulation Control Subsystem: A Fault Tolerant Approach" In *Proc 15th AIAA/IEEE Digital Avionics Systems Conference*, October 1996

[MAR97] A. Martinelli, F. Torchia - "The Software Maintenance in the BeppoSax Scientific Mission", In *Proc. DASIA'1997*, Sevilla, Spain, pp 369-374, May 26-29 1997

[MUS98] N. Muscettola et al. - "Remote Agent: to Boldly Go Where No AI System has Gone Before" In *AI*, Vol. 103, No. 1-2, pp 5-48, 1998

[STE92] D. B. Stewart, R. A. Volpe, P. K. Khosla - "Integration of Real-Time Software Modules for Reconfigurable Sensor-Based Control Systems" In *Proc IEEE/RSI (IROS'92)*, Raleigh, NC, July 7-10, 1992, pp 325-332

[BAL98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, J. Y. Vion-Dury - "Architecturing and Configuring Distributed Appplications with Olan" In *Proc IFIP, DSP & ODP (Middleware'98)*, The Lake District, UK, Sept 1998

[MAG89] J. Magee, J. Kramer, M. Sloman - "Constructing Distributed Systems in Conic" In *IEEE Trans. on Software Engineering*, Vol. 15, No. 6, pp 663-675, June 1989