Issues in applying a model driven approach to reconfigurations of satellite software

Jérémy Buisson, Cecilia Carro, Fabien Dagnat

Issues in applying a model driven approach to reconfigurations of satellite software

Jérémy Buisson Cecilia Carro Fabien Dagnat

Institut TELECOM / TELECOM Bretagne
Université européenne de Bretagne
{Jeremy.Buisson,Cecilia.Carro,Fabien.Dagnat}@telecom-bretagne.eu

Abstract

Satellite software has to deal with specific needs including high integrity and dynamic updates. In order to deal with these requirements, we propose working at a higher level of abstraction thanks to model-driven engineering. Doing so involves many technologies including model manipulation, code generation and verification. Before we can implement the approach, there is a need for further research in these areas, e.g., about meta-transformations in order to maintain several consistent related code generators. We highlight such issues in regard to the current state of the art.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; K.6.3 [*Management of Computing and Information Systems*]: Software Management—Software process

General Terms maintenance, design process, satellite

Keywords reconfiguration, MDE, real-time

1. Introduction

In satellite architectures, the flight software is a real-time software that provides services that are common to whatever mission-specific payload the spacecraft is assigned. Typical services include driving the desired orbit and flight, controlling thermal regulation systems, managing power sources and communicating with ground stations. Even after the satellite has been launched, flight software must be adjusted. Satellites are subject to particles that may damage hardware components. Such damages cannot be fixed except installing software workarounds. Bug fixes and software updates should be propagated to flying satellites. In addition, mission extensions may require functional enhancements. All of these reasons motivate the need for reconfiguration.

As flight software is critical to the success of the mission, space industries and agencies have worked on engineering processes in order to help increase confidence. For instance, the European space agency has published standards (ECSS-E-40A; ECSS-E-40B) on software engineering and (ECSS-Q-80B) on product assurance. These standards do not prescribe a specific process. They rather formalize documents, list requirements of the process and assign responsibilities to involved partners. Regarding updates, standards state for instance that the type, scope and criticality must be documented; that updates must be validated; and so on.

Current industrial practices for updates are restricted by technological choices resulting from the document driven V process. Modifications are done by-hand either on the compiled assembly code or on the C/Ada source code. In order to take into account constraints such as the scarcity of the ground-satellite link, only differences between old and new binary images are sent to the satellite. Coding rules that reduce these differences are an important know-how of space companies. Trivial rules consist, for instance, in overwriting operators or inserting "nop" instructions in order to avoid moving the code. Similar practices are also employed in other areas. In the area of wireless embedded systems such as mobile phones, (von Platen and Eker 2006) have worked on specific linking schemes and MMU usage in order to minimize data movements at the time of the update.

Current approaches have two main drawbacks. First, developing such patches requires a high level of expertise. While managing small changes is somewhat easy, it is difficult to get confidence in case of complex patches due to the burden of patch efficiency. Second, each software version and memory mapping requires a specific patch. The approach does not scale to either satellite families or constellations, as for instance damages on memory banks result in each satellite having a unique memory mapping.

With regard to updates, switching to MDE gives the opportunity to increase abstraction up to the level of models when designing and developing reconfigurations. Technical

details of patch development would be downgraded as model transformations and code generation, which would in addition capture most of the involved expertise.

In collaboration with major European manufacturers, the SPaCIFY project (The SPaCIFY Consortium 2008) aims at bringing advances in MDE to the satellite flight software industry. It focuses on software development and maintenance phases of satellite lifecycles. The project advocates a top-down approach built on a domain-specific modeling language named Synoptic. In line with previous approaches to real-time modeling such as Statecharts and Simulink, Synoptic features hierarchical decomposition in synchronous block diagrams and state machines. SPaCIFY also emphasizes verification, validation and code generation.

In this exploratory paper, we focus on the case of reconfiguration. We first emphasize specific considerations and requirements when updating the software of flying satellites (Section 2). Then we consider the design process as laid out in the SPaCIFY project (Section 3). Given the current state of the art, we aim at highlighting challenges that we have to overcome in order to implement the SPaCIFY MDE process. Step by step, we depict the needs of each phase of the process. For each phase, we discuss how previous works can contribute to the process and we present issues that remain to be solved. That way, sharing our experience, we give some directions and motivations for future research.

2. Satellite-specific needs

In European satellites, the execution platform is resource constrained, typically made of a radiation hardened ERC32 or LEON processor with up to 10MB memory. Flight software consists of communicating periodic tasks that run on top of a fixed-priority real-time kernel. Depending on the implementation strategy, the number of tasks ranges from a dozen to nearly 40. Activation periods range from 50ms to 10s. In addition to periodic tasks that implement control laws, the software has to handle asynchronous events sent by operators (telecommand requests) and onboard devices.

Some vital tasks must not have their execution affected, otherwise the satellite's operation might be compromised. Despite reconfigurations, these tasks must respect real-time constraints. In case they have to be updated, constraints shall propagate to reconfigurations, which have then to complete upon given deadlines.

Real-time and criticality constraints also restrict code generation in the mainstream development process. In order to obtain high integrity in the generated code, it is required that the behavior of any piece of code can be determined statically. Among important properties, execution time must be statically bounded. Hence features such as indirect function calls, dynamic binding, dynamic memory allocation, dynamic task creation are not allowed.

In addition, the quality of the communication link between ground stations and satellites is commonly poor. The bandwidth is low in comparison to current wired networks. Furthermore, communication can only occur while the satellite is in range of at least one ground station. Therefore, the size of reconfigurations shall be kept as low as possible.

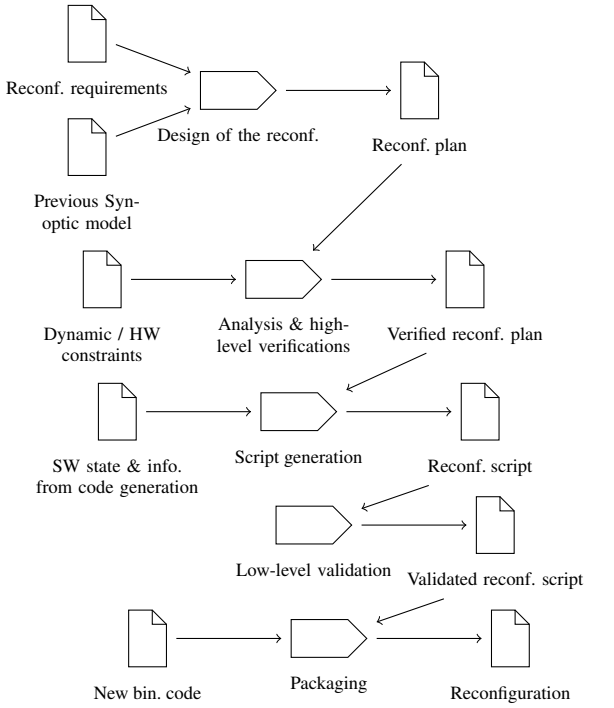

Figure 1. Overview of the workflow for reconfigurations.

In the case of telecommunication satellites and positioning systems, several satellites are manufactured from the same design. However, equipment aging and hardware damages make each satellite unique, even in a single series. Software has to be adapted specifically to each satellite. Consequently, copies of software tend to diverge, though some updates like bug fixes are relevant for whole families. In order to avoid duplicating development effort, reconfigurations shall be easily adaptable to variants of the software.

3. Relevant works to support the process

As outlined in (The SPaCIFY Consortium 2008), the process consists in enriching Synoptic models with non-functional properties such as the mapping of software elements onto tasks. The resulting architecture is then used in order to generate the source code of the application and the configuration of the runtime support environment.

We rely on the main phases of the process (Figure 1) to structure the discussion. Starting from the requirements and from the previous version of the application, a reconfiguration plan is designed (Section 3.1). Working at the level of models, the plan describes how the logical architecture of the application has to be updated. Hardware and dynamic constraints are used in order to enrich models with concrete architecture details (Section 3.3). Based on informa-

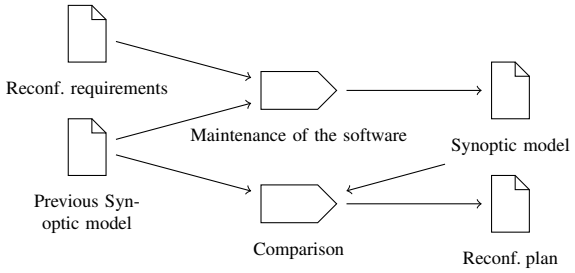

Figure 2. Design driven by the model of the software.

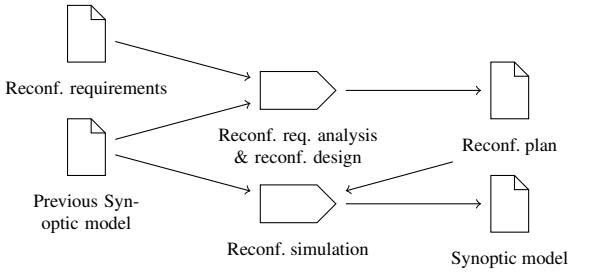

Figure 3. Design driven by the reconfiguration.

tion gathered during code generation and on the actual state of the executing software, the reconfiguration plan is translated (Section 3.2) into a script that works at the level of the code. A second validation step occurs on this new representation of the reconfiguration (Section 3.3). Last, the script is packaged (Section 3.4) along with the required binary code. Section 3.5 discusses involved reconfiguration languages.

3.1 Design of reconfiguration plans

The design of reconfiguration plans is triggered when the need for maintenance arises. Requirements result from either problem diagnosis, requests for improvements or problem prevention. Based on these requirements, the design phase consists in identifying the impact of reconfigurations on the model of the software. To do so, we present two alternative processes, which involve different tools.

In Figure 2, maintainers produce a fresh model. Doing so straightforwardly reuses the same tools as in the mainstream process. In order not to start over, a reference previous version of the application model can be used. As a second step, previous and new versions of the application model are compared. The reconfiguration plan is the resulting list of differences, i.e., instructions for adding and removing components and bindings. Several proposals exist to compute the differences between models. (Kim et al. 2007; Xing and Stroulia 2007) focus on finding structural differences between two class diagrams. In the context of object-oriented languages, (Apiwattanapong et al. 2007) try to highlight behavioral changes resulting from structural differences, e.g., due to subtyping and dynamic binding. (Phung-Khac et al. 2008) propose to memorize design choices such that differences are emphasized by the design process.

In Figure 3, maintainers design the reconfiguration plan by hand. The language shall be richer than the one of the first alternative, as maintainers can use sophisticated control structures that difference computing algorithms could hardly emit. As highlighted in FPath and FScript (Polakovic et al. 2007), a convenient way to refer to components is also desirable. Given the plan, new version models can be obtained thanks to simulations of the reconfiguration.

Options mainly differ in the involved reasoning. While the first one makes maintainers think of the resulting application model, the second option rather puts the focus on the deltas to reach that resulting model. Requirements can be expressed in both manners: extending mission statements specifies the result of a reconfiguration; fixing a bug specifies a difference. Options also impact the approach to validation. In the first option, reconfiguration plans are correct by construction, while in the second alternative one must ensure that plans can execute on given application models.

With regard to considerations of Section 2, the main challenge is to deal with many software variants. This involves *propagating* changes modeled by reconfiguration plans to the collection of variants. Model merging and composition (Brunet et al. 2006; Fleurey et al. 2007) allow assembling several pieces of models, some of them coming from the reconfigurations. Propagating model transformations (Alanen and Porres 2004; Tratt 2008) are transformations that propagate changes in source models while preserving modifications of target models, hence propagating reconfigurations over several version branches. As (Tratt 2008) acknowledges, results in this area are in an early stage.

3.2 Script generation

Previous works (Ketfi and Belkhatir 2005; Buisson and Dagnat 2008) abstract concrete component models in unified frameworks. Script generation is parameterized in order to map reconfigurations down to the concrete model.

In our case, concrete code results from well-known transformations of abstract models. Hence the challenge is to map these transformations from the application domain to the reconfiguration domain, as Figure 4 depicts. Meta-transformations, i.e., transformations of transformations (Varró and Pataricza 2004; Ward and Zedan 2005; Balogh and Varró 2006) have been used mainly as demonstrations for reflective transformation languages. In our case, we need a systematic approach such that code generation and script generation remain consistent.

Traditionally, reconfiguration of component-based software relies on dynamic memory management in order to instantiate components. However, as mentioned in Section 2, such techniques cannot be used in the context of satellite software. The replaceable unit and cell (Gagliardi et al. 1996) and service (Rasche and Polze 2005) concepts do not solve the above restrictions, despite their focus on reconfiguring real-time applications. (Zalila et al. 2008) have

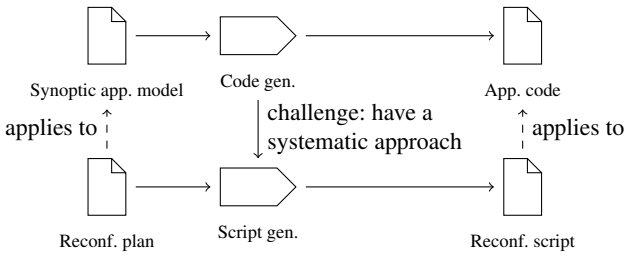

Figure 4. Application and reconfiguration code generation.

proposed a component implementation scheme that adheres to memory management restrictions, but they have not considered the case of reconfigurations. In order to apply the same restrictions to reconfigurations, one must do memory management offline, while the reconfiguration is being prepared. We are not aware of previous work about this topic.

3.3 Verification and validation

We emphasize two steps in verifications. The first one aims at asserting the application resulting from the reconfiguration. Standard verification techniques apply. The second step validates the reconfiguration itself. Several properties have been formalized as candidate correctness criteria: the future of the execution shall conform to the specification (Bloom and Day 1993); it should reach a reachable state of the new version (Gupta et al. 1996); or effective executions shall conform to specifications (Zhang and Cheng 2005).

In order to deal with real-time constraints, (Rasche and Polze 2005) have worked on bounding the reconfiguration time. However their model is coarse: stopping all the required components; then reconfiguring; then restarting components. The finer approach of FScript (Polakovic et al. 2007) does not provide any study about its predictability. In Simplex, (Gagliardi et al. 1996; Lee and Sha 2005) do not describe how to make reconfigurations fit real-time constraints. If bounding the reconfiguration time allows schedulability analyses, scheduling reconfigurations in a running system it usually not addressed. Sacrificing some application components would in addition allow some reconfigurations that would otherwise break schedulability of the system and therefore be rejected.

3.4 Reconfiguration packaging

During the design of a reconfiguration, the need for new pieces of code arises. The development of these pieces of code follows the mainstream process. As we have already mentioned in Section 2, it is essential to reduce the length of transmissions. In order to address this issue, current practices (e.g., computing binary differences – see Section 1) can be reused. Instead of applying them to the whole software, they are used on the code of each structural element independently one of the others. Last, the result is packaged with the reconfiguration script and uploaded to the satellite.

3.5 Reconfiguration languages

In the above description, several languages are used to express reconfigurations. Reconfiguration plans and reconfiguration scripts differ in the target they address. The former, targeting models, expresses reconfigurations in terms of adding and removing blocks and connectors (in block diagrams) and states and transitions (in state machines). The latter, targeting code, expresses reconfigurations in terms of overwriting values, loading code and changing the targets of control and memory instructions. Depending on whether plans are automatically generated or hand-written, high level constructs (such as control structures, functions, filtering) shall be relevant or not. Similarly, the manner to designate reconfigured elements can range from identifiers (e.g., in reconfiguration scripts) to pattern matching (e.g., in hand-written reconfiguration plans).

Having a modular way to build reconfiguration languages would ease obtaining all of the variants involved in the design process. We have observed that reconfiguration languages contain 3 fragments: reconfiguration instructions; expressions to designate reconfigured elements; constructs to combine instructions. Similarly, (Colosi and Smith 2008) have identified the need for combining several orthogonal domains in a single domain-specific language. This issue instantiates the "extensible expression" problem (Garrigue 2000; Zenger and Odersky 2004), which emphasizes the fact that it should be possible to build languages incrementally, i.e., that extending a language must not require to recompile the code that handle preexisting language fragments.

4. Conclusion

Satellite software has specific characteristics. In addition to being subject to resource constraints, it has to meet high integrity requirements. It must also support dynamic updates without hazarding satellites. Using a model-driven approach as the SPaCIFY project advocates increases the level of abstraction. Nevertheless, as we have shown, given the current state of the art in the relevant areas, there are still issues that have first to be solved in order to implement such a process. In the area of models, it appears that there is a lack of tools for comparing models and propagating changes. There is also a need for enforcing consistency between several transformations, in our case between code generation and script generation. Regarding verifications, we need finer analyses that would allow better insertion of reconfigurations in components' execution. In some situations, it is also desirable to be able to override real-time constraints of the application in a controlled manner. Last, as we have shown, several related reconfiguration languages are used. Increasing the modularity would help us obtain all of them.

In future work, we plan to focus on how to derive generation rules for reconfiguration scripts from code generation. To do so, we will build on our current work on abstract reconfiguration models (Buisson and Dagnat 2008) and on

reconfiguration languages. In the context of validation, reconfiguration models have to be enhanced in order to ease interleaving reconfigurations with application tasks. We plan to work on the runtime support for such models.

Acknowledgments

This work is funded by the French national research agency through the SPaCIFY consortium (ANR 06 TLOG 27). We thank Loic Plassart and Iulian Neamtiu for their comments.

References

M. Alanen and I. Porres. Change propagation in a model-driven development tool. In *Workshop in Software Model Engineering*, Lisbon, Portugal, October 2004.

T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: a differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, March 2007.

A. Balogh and D. Varró. Advanced model transformation language constructs in the VIATRA2 framework. In *Symposium on Applied Computing*, pages 1280–1287, Dijon, France, April 2006.

T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2): 102–108, March 1993.

G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *International Conference on Software Engineering*, pages 5–12, Shanghai, China, May 2006.

J. Buisson and F. Dagnat. Experiments with Fractal on modular reflection. In *6th International Conference on Software Engineering Research, Management and Applications*, Prague, Czech Republic, August 2008.

J. Colosi and D. Smith. TS-5693: Writing your own JSR-compliant, domain-specific scripting language. In *JavaOne*, San Francisco, California, USA, May 2008.

ECSS-E-40A. Space engineering: software – part 1: principles and requirements. Technical report, ECSS (European Cooperation for Space Standardization), Noordwijk, The Netherlands, November 2003.

ECSS-E-40B. Space engineering: software – part 2: document requirements definitions. Technical report, ECSS (European Cooperation for Space Standardization), Noordwijk, The Netherlands, March 2005.

ECSS-Q-80B. Space product assurance. Technical report, ECSS (European Cooperation for Space Standardization), Noordwijk, The Netherlands, October 2003.

F. Fleurey, B. Baudry, R. France, and S. Gosh. A generic approach for automatic model composition. In *Workshop on Aspect Oriented Modeling*, Nashville, Tenessee, USA, September 2007.

M. Gagliardi, R. Rajkumar, and L. Sha. Designing for evolvability: building blocks for evolvable real-time systems. In *Real-time Technology and Applications Symposium*, pages 100–109, Brookline, Massachusetts, USA, June 1996.

J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, Sasaguri, Japan, November 2000.

D. Gupta, P. Jalote, and G. Barua. A formal framework for online software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

A. Ketfi and N. Belkhatir. Model-driven framework for dynamic deployment and reconfiguration of component-based software systems. In *Symp. on Metainformatics*, Esbjerg, Denmark, 2005.

M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *International Conference on Software Engineering*, pages 333–343, Minneapolis, Minnesota, USA, May 2007.

K. Lee and L. Sha. A dependable online testing and upgrade architecture for real-time embedded systems. In *International Conference on Embedded and Real-time Computing Systems and Applications*, pages 160–165, San Francisco, California, USA, March 2005.

A. Phung-Khac, A. Beugnard, J.-M. Gilliot, and M.-T. Segarra. Model-driven development of component-based adaptive distributed applications. In *Symposium on Applied Computing*, pages 2186–2191, Fortaleza, Ceara, Brazil, March 2008.

J. Polakovic, S. Mazare, J.-B. Stefani, and P.-C. David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *Combonent-Based Software Engineering*, volume 4608 of *LNCS*, pages 242–257, Medford, Massachusetts, USA, July 2007.

A. Rasche and A. Polze. Dynamic reconfiguration of component-based real-time software. In *Workshop on Object-oriented Real-time Dependable Systems*, pages 347–354, Sedona, Arizona, USA, February 2005.

The SPaCIFY Consortium. The SPaCIFY project. In *Data Systems In Aerospace*, Palma de Majorca, Spain, May 2008.

L. Tratt. A change propagating model transformation language. *Journal of Object Technology*, 7(3):107–126, March 2008.

D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *International Conference on the Unified Modeling Language*, volume 3273 of *LNCS*, pages 290–304, Lisbon, Portugal, October 2004.

C. von Platen and J. Eker. Feedback linking: optimizing object code layout for updates. *SIGPLAN Not.*, 41(7):2–11, July 2006.

M. Ward and H. Zedan. MetaWSL and meta-transformations in the FermaT transformation system. In *International Computer Software and Applications Conference*, pages 233–238, Edinburgh, UK, July 2005.

Z. Xing and E. Stroulia. Differencing logical UML models. *Automated Software Engineering*, 14(2):215–259, June 2007.

B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *Symposium on Object-Oriented Real-time Distributed Computing*, pages 221–228, Orlando, Florida, USA, May 2008.

M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, EPFL, 2004. URL `http://lampwww.epfl.ch/papers/IC_TECH_REPORT_200433.pdf`.

J. Zhang and B. Cheng. Specifying adaptation semantics. In *Workshop on Architecting Dependable Systems*, pages 1–7, Saint-Louis, Missouri, USA, May 2005.