SPaCIFY: a Formal Model-Driven Engineering for Spacecraft On-Board Software

P. Arberet[a], J.P. Bodeviex[d], F. Boniol[b], J. Buisson[f], G. Cannenterre[h], D. Chemouil[b], A. Cortier[a,c], F. Dagnat[f], F. Dupont[i], M. Filali[d], E. Fleury[j], J. Forget[j], G. Garcia[g], F. Herbreteau[j], E. Morand[a], J. Ouy[e], G. Sutre[j], A. Rugina[d], M. Streker[c], J.P. Talpin[e]

[a]CNES. 18, av. E. Belin, F-31401 Toulouse.
[b]ONERA. 2 av. E. Belin, F-31055 Toulouse.
[c]IRIT, Univ. Paul Sabatier. 118, route de Narbonne F-31062 Toulouse.
[d]EADS Astrium. 31, rue des Cosmonautes, Z.I. du Palays F-31402 Toulouse.
[e]IRISA. Campus de Beaulieu, F-31042 Rennes.
[f]Institut Télécom. Technopole Brest Iroise, F-29238 Brest.
[g]Thales Alenia Space. 100, Boulevard Midi F-06150 Cannes.
[h]Anyware Technologies. ZAC de l'HERS, Allée du LAC F-31672 Labège.
[i]Geensys. 120, rue R. Descartes, F-29280 Plouzané.
[j]LaBRI. 351 cours de la Libération F-33405 Talence.

Abstract: *The aim of this article is to present a model-driven approach proposed by the SPaCIFY project for spacecraft on-board software development. This approach is based on a formal globally asynchronous locally synchronous language called Synoptic, and on a set of transformations allowing code generation and model verification.*

1 Introduction

SPaCIFY is a research project aiming at developing a design environment for spacecraft flight software. The project promotes a top-down method based upon multi-clock synchronous modeling, formally-verified transformations, exhaustive verification through model-checking and a middleware featuring real-time distribution and dynamic-reconfiguration services. The various tools developed are released under FLOSS (free/libre/open-source software) licenses, favouring cost-sharing and sustainability. The project is led by the French Space Agency and gathers prime contractors Astrium Satellites and Thales Alenia Space, ISV's GeenSys (formerly TNI Software) and Anyware Technologies, and academic teams Cama from TELECOM Bretagne, Espresso from Irisa, MV from LaBRI and Acadie from IRIT. It is a 3-years project (starting in February 2007) partly funded by the French Research Agency (ref. ANR 06 TLOG 27).

1.1 Context: space systems

Space systems [11] generally differs from other ground systems by their environmental constraints:

- Most of the time, the communication with a spacecraft is only intermittent, depending on visibility phases. There are two important consequences: the first one is the small amount of data communicable with the ground segment; while the second one lies in the inability for the ground to react quickly to an on-board anomaly.
- Once a spacecraft has been launched, there is obviously no way to operate and observe it except through TC (telecommands) and TM (telemetries). This implies that flight software should have been designed in a suitable way, such that it remains possible to operate the spacecraft in ways unpreviewed at the time of specification.
- Another aspect is that hardware is often limited compared to standard computers, both for what regards memory size and computational power.

- Validation is also generally a harder task than for other ground systems, mostly because some parts of the system simply cannot be tested "for real" before launching.

- Finally, the whole lifecycle of flight software may last from 10 to 20 years. Platform and software designers usually do not keep following the same project for so many years, which may cause a loss of knowledge about the way the product works. This problem becomes strong when update operation are required on a space system specification and design, even when in flight.

1.2 Spacecraft on-board software

Spacecraft on-board software, and more precisely satellite software, can be either central or remote, which means that various pieces of software can run on the central processing unit or on remote terminal units (e.g., a star-tracking software, a processing unit dedicated to the AOCS, etc.). Modeling techniques for spacecraft on-board software should then take into account such a distributed nature.

The main functions implemented by on-board platform are derived from functional chains (power management, thermal management, AOCS (Attitude and Orbit Control System) management, etc.) and are supported by real-time executives (such as RTEMS) providing basic services (task management, scheduling, synchronization and communication facilities, memory management, etc.). Modern central flight software usually comprises around 10 to 20 tasks, a few of them being periodic, the others being sporadic or completely aperiodic. These tasks are generally managed through fixed-priority static scheduling (rate-monotonic style).

1.3 The adressed problem and the SPaCIFY contribution

The hardest part in developing on-board software, both platform and payload parts, lies in the ability, or not, to verify and to validate it. It already happened to find very well-designed software that was almost unverifiable statically (this being due for instance to a large use of function pointers). Obviously, the most difficult tasks concern dynamic and performance matters. Finally, FDIR (Failure Detection, Isolation, and Recovery) is also problematic, as validating it means being able to guess possible failures and to simulate them (which is sometimes impossible for some space equipment). Moreover, some FDIR strategies are so complex that they are almost impossible to validate (for this reason, there has been a movement towards using simple FDIR strategies, possibly needing ground intervention, rather than smart but hard to understand strategies).

The use of formal methods (FM), i.e., fundamental notations and techniques for modeling, analysis, validation of systems in a provably sound way, coupled with model-driven engineering (MDE) approach all along the conception could be an efficient help for designer to ensure some quality in the development.

The SPaCIFY project aims at bringing advances in FM and MDE to the spacecraft on-board software industry. It focuses on software development and maintenance phases of spacecraft life-cycles. 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. Thanks to the formal semantics of Synoptic, formal transformation from Synoptic to two other formal languages have been developed. The first transformation, from Synoptic to AltaRica enables formal verification on Synoptic models. The second one from Synoptic to Signal enable efficient embedded code generation. All these models and transformations form the SPaCIFY development process presented in the following sections.

2 The SPaCIFY approach

2.1 Overview

The SPaCIFY development process is depicted figure 1. The main ambition of this process is to enable developers to focus on design rather than on implementation, and to work with design models promoting a high level of abstraction. In this context, the main *manual* part of the design process is the edition and the definition of a "functional" Synoptic model. Other models are obtained using a bunch of (almost) automated transformations (each arrow figures a chain of various transformations).

The SPaCIFY development process is strongly based on the Synoptic language. Synoptic allows to describe an on-board software at different layers and from different viewpoints:

1. At the highest level, only functional aspects are described by using syntactic constructions such as dataflow diagrams, automata (including mode automata). Functional components may be imported from other formalisms such as Simulink/Stateflow. At this level, and from a semantic point of view, the systems is considered as a GALS (Globally Asynchronous Locally Synchronous) systems composed

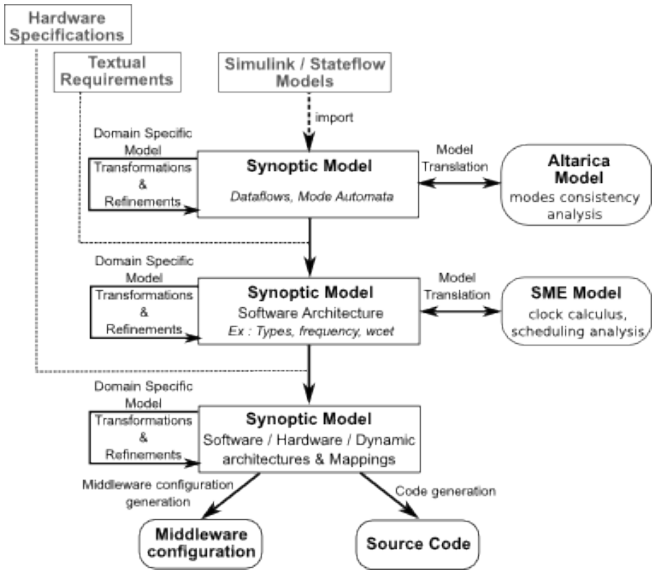

Figure 1. SPaCIFY **development process**

of a set of communicating synchronous islets. Communication between islets are supported by *external variables* seens as middleware ports. This Synoptic funcitonal model may be refined by successive transformations (automata ellicitation, function splitting, etc.). Notice that "functional" does not mean untimed. Temporal information such as rate, frequency, latency, etc. are taken into account in order to allow a complete and formal specification of the expected behavior of the system. Formal verification can be achieved at this level by transforming a Synoptic model into an AltaRica one and by model checking techniques. Such a formal verification method allows to check on the Synoptic specification functional properties such as recovery failure requirements.

2. Plateform information such as hardware and software architectures (in terms of threads, resources, and mapping from the firsts to the seconds) are taken into account at a lower level. The dynamic software architecture and the hardware architecture (specified in an external formalism, for instance, AADL) are then introduced. Synoptic functional components are split or grouped into threads components and mapped onto resources according to architectural choices defined by the designer.

3. Finally, two transformation are applied at the low level model: code generation, and middleware configuration generation. These transformation are supported by an intermediate transformation from Synoptic to the synchronous Signal language, enabling the use at the Synoptic level of the Polychrony suite tools.

2.2 The Synoptic modeling language

Synoptic is a Domain Specific Modeling Language (DSML) which aims to support all aspects of embedded flight-software design. As such, Synoptic consists of heterogeneous modeling and programming principles defined in collaboration with the industrial partners and end users of the SPaCIFY project. The aim of this section is only to give a brief overview of this language. See [13] for a detailled presentation oo Synoptic.

Used as the central modeling language of the SPaCIFY model driven engineering process, Synoptic allows to describe different layers of abstraction: at the highest level, the software architecture models the functional decomposition of the flight software. This is mapped to a dynamic architecture which defines the thread structure of the software. It consists of a set of threads, where each thread is characterized by properties such as its frequency, priority and activation pattern (periodic, sporadic). At the lowest level, the hardware architecture permits to define devices (processors, sensors, actuators, busses) and their properties.

Synoptic permits to describe three types of mappings between these layers:

- mappings which define a correspondence between the software and the dynamic architecture, by specifying which blocks are executed by which threads;
- mappings which describe the correspondences between the dynamic and hardware architecture, by specifying which threads are executed by which processor,
- and mappings which describe a correspondence between the software and hardware architecture, by specifying which data is transported by which bus for instance.

Figure 2 depicts the principles discussed above. Our aim is to synthesize as much of these mappings as possible, for example by appealing to internal or external schedulers. However, to allow for human intervention, it is possible to give a fine-grained mapping, thus overriding or bypassing machine-generated schedules. Anyway, consistency of the resulting dynamic architecture is verified by the SPaCIFY tool suite, based on the properties of the software and dynamic model.

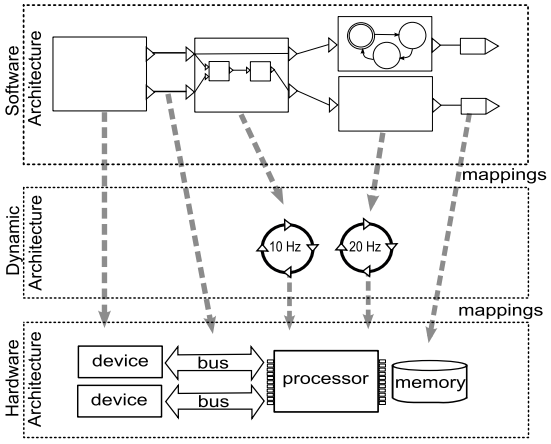

Figure 2. Global view : layers and architecture mappings

At each step of the development process, it is also useful to model different abstraction levels of the system under design inside a same layer (functional, dynamic or hardware architecture). Synoptic offers this capability by providing an incremental design framework and refinement features.

To summarize, Synoptic deals with dataflow diagrams, mode automata, blocks, components, dynamic and hardware architecture, mapping and timing.

In this section we focus on the functional part of the Synoptic language which permits to model software architecture. This sub-language is well adapted to model *synchronous islands* and to specify interaction points between these islands and the middleware platform using the concept of *external variables*.

Synchronous islands and middleware form a Globally Asynchronous and Locally Synchronous (GALS) system.

The development of the Synoptic software architecture language has been tightly coordinated with the definition of the GeneAuto language [24]. Synoptic uses essentially two types of modules, called blocks in Synoptic, which can be mutually nested: dataflow diagrams and mode automata. Nesting favors a hierarchical design and allows to view the description at different levels of details.

By embedding blocks in the states of state machines, one can elegantly model operational modes: each state represents a mode, and transitions correspond to mode changes. In each mode, the system may be composed of other sub-blocks or have different connection patterns among components.

Apart from structural and behavioral aspects, the Synoptic software architecture language allows to define temporal properties of blocks. For instance, a block can be parameterized with a frequency and a worst case execution time which are taken into account in the mapping onto the dynamic architecture.

The formal semantics of the Synoptic language relies on the polychronous paradigm. This semantics was used for the definition of the transformation of Synoptic models towards the synchronous language SIGNAL and SME models [4] following a MDE approach. This transformation allows to use the Polychrony platform for model transformation hints for the end user, such as splitting of software as several synchronous islands and simulation code generation purposes. On the other hand, the formal semantics definition allows for neat integration of verification environments for ascertaining properties of the system under development.

Synoptic is equipped with an assertion language that allows to state desired properties of the model under development. We are mainly interested in properties that permit to express, for example, coherence of the modes ("if component X is in mode m1, then component Y is in mode m2" or "... can eventually move into mode m2"). Specific transformations extract these properties and pass them to the verification tools.

2.3 Verification methodology

Verification of the Synoptic model takes place at an early step in the software development process. It is mainly intended to track conception inadequacies and logic errors in the model. Thus, the model-checker is fed with the Synoptic model and does not handle the lower levels of the development process (source code generation and middleware configuration).

In short, the Synoptic model and its specifications are first translated into the AltaRica language [1, 21] and then verified with Arc[1] or Mec 5[2] [14]. When an error is discovered by the model-checker, a concrete run is extracted from the symbolic trace and back-translated and mapped onto the Synoptic model.

2.3.1 AltaRica Language and Tools

AltaRica is a high-level language designed for the modelling of complex systems. The well-defined semantics of AltaRica is based on the Arnold-Nivat model but applied to *Constraint Automata*. Synchronisation of constraint automata is made up of: (a) the standard Arnold-Nivat strong synchronization of events; (b) boolean constraints on variables shared by automata; ans (c) a weakest synchroniza-

[1] http://altarica.labri.u-bordeaux.fr/tools:arc

[2] http://altarica.labri.u-bordeaux.fr/tools:mec_5

tion mechanism of events that is similar to a broadcast mechanism.

Several tools exist for the analysis or the compilation of theAltaRica language. Below, are listed the ones developed mainly by the MF/MV theme of the LaBRI. There also exist commercial tools [2, 18, 19] but these tools work with restricted variants of the AltaRica language.

- ARC is an AltaRica model checker based on both explicit and symbolic representation of transition systems. ARC computes the semantics with a syntactic flattening of the AltaRica hierarchical model. ARC can also be used as a gateway with other languages. In particular, there exists translators linking the AltaRica and Lustre languages.

- Mec 5 is a model-checker based on the Binary Decision Diagram (BDD) technology [5] and uses AltaRica descriptions as input. Mec 5 computes the semantics inductively based on the hierarchy of the AltaRica model. It can be used to check μ-calculus formulas against AltaRica models but it can also be used as a simple μ-calculus calculator over finite domains. The current version of Mec 5 does not support the full syntax of the AltaRica.

Basically, the difference between the two tools can be summed up as follow: The ARC model-checker is used to deal with big models and simple properties, whereas Mec 5 is a more academic tools intended to deal with small models and complex properties.

2.3.2 Translation from Synoptic to AltaRica

The process of translating the Synoptic model into an AltaRica model is preserving: the hierarchy of the original model, the operational semantics coded in the automata. Many other features are supported but the float variables and clocks.

The main problem raised while conceiving the automatic translation was to deal with the asynchronous model when AltaRica is synchronous. Anyway, this issue has been solved by forcing the synchronicity of all the components and exploring the different shifts that can occur.

It also worth noticing that the translation has been done in a way that preserve back-translation of error trace onto the original model. It is thus possible to syntactically relate the error trace sent by the model-checker to the original Synoptic model.

3 SPaCIFY middleware

In order to support the execution of code generated from Synoptic model, a middleware has been defined. This SPaCIFY middleware addresses two main points: the scarceness of resources in a spacecraft, and the large variety of spacecraft platforms from one project to another.

To take into account the scarceness of resources, the middleware is tailored to the domain and adapted to each specific project. This notion of generative middleware is inherited from the ASSERT[3] project which has studied proof-based engineering of real-time applications but is here specialised to the area of satellite software. ASSERT defines a so-called virtual machine, which denotes a RTOS kernel along with a middleware and provides a language-neutral semantics of the Ravenscar profile [9]. It relies on PolyORB-HI [15], a high integrity version of PolyORB refining the broker design pattern [10], which fosters the reuse of large chunks of code when implementing multiple middleware personalities. Satisfying restrictions from the Ravenscar profile, PolyORB-HI can suitably be used as a runtime support for applications built with AADL code generators. In our context, the support of a precise adaptation to the need of the middleware is obtained thanks to its generation based on the requirements expressed in the Synoptic models (mainly through the interaction contracts attached to external variables).

3.1 Middleware architecture

Figure 3 depicts the overall architecture of the SPaCIFY middleware. Following previous work on middlewares for real-time embedded systems [3, 22], the SPaCIFY middleware has a microkernel-like or service-based architecture. That way, high flexibility allows to embed only the services that are required, depending on the satellite. While previous work has focused on techniques to reduce the time during which the reconfigured system is suspended, along with an analysis to bound that time, our work aims at: (a) using application model to specify reconfiguration, (b) proposing a contract approach to the specification of relation between applications and the middleware, and (c) providing on demand generation of the needed subset of the middleware for a given satellite.

The RTOS kernel offers the usual basic services such as task management and synchronisation primitives. The core middleware is built on top of this RTOS and provides core services of composition and communication. They are not intended to be used by application-level software. They rather provide means to structure the middleware itself in smaller composable entities, the services. Upon

[3] http://www.assert-project.net/

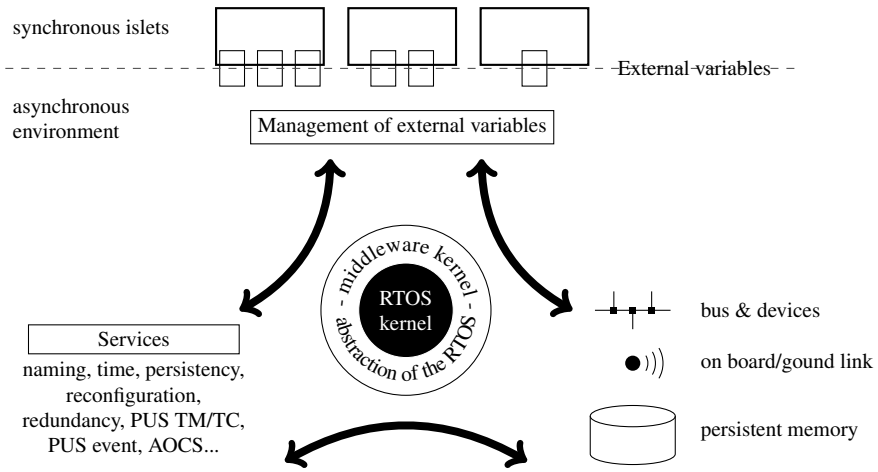

Figure 3. Architecture of the middleware.

these abstractions are built general purpose services and domain specific services. These services are to be used (indirectly) by the application software (here the various synchronous islets). They can be organized in two categories as follows:

- The first layer is composed of general purpose services that may be found in usual middleware. Among them, the naming service implements a namespace that would be suitable for distributed systems. The persistency service provides a persistent storage for keeping data across system reboots. The redundancy service helps increasing system reliability thanks to transparent replication management. The reconfiguration service, further described in subsection 3.3, adds flexibility to the system as it allows to modify the software at runtime. The task & event service contributes real-time dispatching of processors according to the underlying RTOS scheduler. It provides skeletons for various kinds of tasks, including periodic, sporadic and aperiodic event-triggered tasks, possibly implementing sporadic servers or similar techniques [23, 16].
- The second layer contains domain-specific services to capture the expertise in the area of satellite flight control software. These services are often built following industry standards such as PUS [12]. The TM/TC service implements the well established telemetry/telecommand link with ground stations or other satellites. The AOCS (attitude and orbit control system) controls actuators in order to ensure the proper positioning of the satellite.

To support the execution the platform use hardware resources provided by the satellite. As the hardware platform changes from one satellite to another, it must be abstracted even for the middleware services. Only the implementation of specific drivers must be done for each hardware architecture. During the software development lifecycle, the appropriate corresponding service implementations are configured and adapted to the provided hardware.

As already exposed, one particularity of the SPaCIFY approach is the use of the synchronous paradigm. To support the use of the middleware services within this approach, we propose to use an abstraction, the *external variable*. Such a variable abstracts the interaction between synchronous islets or between a synchronous islet and the middleware relaxing the requirements of synchrony. In the model, when the software architect wants to use a middleware service, he provides a contract describing its requirements on the corresponding set of external variables and the middleware is in charge to meet these requirements. Clearly, this mediation layer in charge of the external variables management is specific to each satellite. Hence the contractual approach drives the generation of the proper management code. This layer is made of *backends* that capture all the asynchronous concerns such as accessing to a device or any aperiodic task , hence implementing asynchronous communications of the GALS approach. The middleware is in charge of the orchestration of the exchange between external variables, their managing backends and the services while ensuring the respect of quality of service constraints (such temporal one) specified in their contracts.

3.2 The middleware kernel and external variables

As stated above, the middleware is built around a RTOS providing tasks and synchronisation. As the RTOS cannot be fixed due to industrial constraints, the middleware

kernel must provide a common abstraction. It therefore embeds its own notion of task and for specific RTOS an adaptor must be provided. The implementation of such adaptors has been left for future until now. Notice that the use of this common abstraction forbids the use of specific and sophisticated services offered by some RTOS. The approach here is more to adapt the services offered by the middleware to the business needs rather using low level and high performance services.
The task is the unit of execution. Each task can contain Synoptic components according to the specification of the dynamic architecture of the application. Tasks have temporal features (processor provisioning, deadline, activation period) inherited from the Synoptic model. The middleware kernel is in charge of their execution and their monitoring. It is also in charge of provisioning resources for the aperiodic and sporadic tasks.
Communication inside a task results from the compilation of the synchronous specification of the various components it must support. All communication outside a task must go through external variables limiting interaction to only one abstraction. External variables are decomposed in two sub abstractions:

- The *frontend* identified in the Synoptic model and that constitutes the interaction point. It appears as usual signal in the synchronous model and may be used as input or output. The way it is provided or consumed is abstracted in a contract that specify the requirements on the signal (such as its timing constraints). An external variable is said to be asynchronous because no clock constraint is introduced between the producer of the signal and its consumer. In the code generated access to such variables are compiled into getter and setter function implemented by the middleware. The contract must include usage requirements specifying the way the signal is used by the task (either an input or an output). They may also embed requirements on the freshness of the value or on event notifying value change.

- The *backend* specified using stereotypes in the contract configure the middleware behavior for non synchronous concerns. For example, persistency contract specify that the external variable must be saved in a persistent memory. Acquisition contracts can be used by a task to specify which data it must get before its execution. Such backends are collected and global acquisition plan are built and executed by the middleware.

As the middleware supports the reconfiguration of applications at runtime, tasks can be dynamically created. The dynamic modification of the features is also provided by the middleware. The middleware will have to ensure that these constraints are respected. Beware that any modification must be made on the model and validated by the SPaCIFY tools to only introduce viable constraints. Each task has a miss handler defined. This defensive feature makes the middleware execute corrective actions and report an error whenever the task does not respect its deadline.

3.3 Reconfiguration service

Among, domain specific services offered by the middleware, the reconfiguration service is the one that had the most impact on the middleware conception.

The design of the reconfiguration service embedded in the SPaCIFY middleware is driven by the specific requirements of satellite onboard software. Reconfiguration is typically controlled from ground stations via telemetry and telecommands. Human operators not only design reconfigurations, they also decide the right time at which reconfigurations should occur, for instance while the mission is idle. Due to resource shortage in satellites, the reconfiguration service must be memory saving in the choice of embedded metadata. Last, in families of satellites, software versions tend to diverge as workarounds for hardware damages are installed. Nevertheless, some reconfigurations should still apply well to a whole family despite the possible differences between the deployed software.

In order to tackle these challenges, the reconfiguration service rely on the structural model (Synoptic models) of the OBSW enriched by the current state of the satellite as described in Figure 4. Using the structure of the software allows to abstract the low-level implementation details when performing reconfigurations. While designing reconfigurations, operators can work on models of the software, close to those used at development time, and specify so called abstract reconfiguration plan like: *change the flow between blocks A and B by a flow between A and C*. An abstract reconfiguration plan use high level elements and operations which may increase the CPU consumption of reconfigurations compared to low level patches. Typical operations such as pattern matching of components make the design of reconfiguration easier, but they are compute intensive. Instead of embedding the implementation of those operations into the satellite middleware, patterns may be matched offline, at the ground station, thanks to the knowledge of the flying software. We therefore define a hierarchy of reconfiguration languages, ranging from high-level constructs presented to the reconfiguration designer to low-level instructions implemented in the

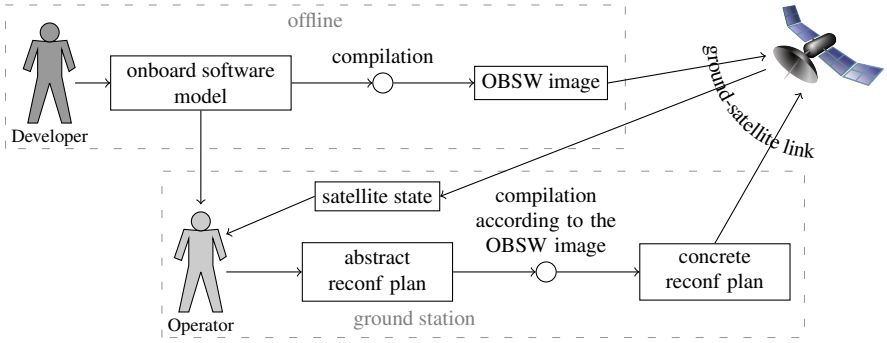

Figure 4. Process and responsabilities for reconfigurations.

satellite runtime. Reconfigurations are compiled in a so called concrete reconfiguration plan before being sent to the satellite. For instance, the abstract plan *upgrade A* may be compiled to *stop A and B; unbind A and B; patch the implementation of A; rebind A and B; restart A and B*. This compilation process uses proposed techniques to reduce the size of the patch sent to the satellite and the time to apply it [25]. Another interest of this compilation scheme is to enable the use of the same abstract reconfiguration plan to a whole family of satellites. While traditional patch based approaches make it hard to apply a single reconfiguration to a family, each concrete plan may be adapted to the precise state of each of the family member.

Applying the previously model driven approach to reconfigurations of satellite software raise a number of other issues that are described in [6]. Work remains to be conducted to get the complete reconfiguration tool chain available. The biggest challenge being the specification of reconfiguration plan. Indeed, considering reconfiguration at the level of the structural model implies to include the reconfigurability concern in the metamodel of Synoptic. If reconfigurability is designed as an abstract framework (independent of the modeling language), higher modularity is achieved when deciding the elements that are reconfigurable. First experiments made on Fractal component model [7] allow us to claim that focusing on the points of interest in application models, metadata for reconfiguration are more efficient. Indeed, separating reconfigurability from Synoptic allows to download metadata on demand or to drop them at runtime, depending on requested reconfigurations.

Lastly, applying reconfiguration usually require that the software reach a quiescent state [20]. Basically, the idea consists in ensuring that the pieces of code to update are not active nor will get activated during their update. For reactive and periodic components as found in the OBSW, such states where reconfiguration can be applied may not be reached. We have proposed another direction [8]. We consider that active code can be updated consistently. Actually, doing so runs into low-level technical issues such as adjusting instruction pointers, and reshaping and relocating stack frames. Building on previous work on control operators and continuation, we have proposed to deal with the low level difficulties using the notion of continuation and operators to manipulates continuations. This approach do not make updating easier but gives the opportunity to relax the constraints on update timing and allow updates without being anticipated.

4 Modeling environment

The SPaCIFY design process has been equipped with an Eclipse-based modeling workbench. To ensure the long-term availability of the tools, the Synoptic environment rely on open-source technologies: it guarantees the durability and the adaptability of tools for space projects which can last more than 15 years. We hope this openess will also facilitate adaptations to other industries requirements.

The development of the Eclipse-based modeling workbench started with the definition of the Ecore meta-model of the Synoptic language. The definition of this meta-model has relied on the experience gained during the GeneAuto project. This definition is the result of a collaborative and iterative process. In a first step, a concrete syntax relying on the meta-model has been defined using academic tools such as TCS (Textual Concrete Syntax) [17]. This textual syntax was used to validate the usability of the language through a pilot case study. These models have helped to improve the Synoptic language and to adapt it to industrial know-how. Once the language was stabilized, a graphical user editor was designed. A set of structural and typing constraints have been formalized, encoded in OCL (Object Constraint Language), and integrated into the environment.

5 Industrial case study

The industrial partner of the project (Astrium and Thales Alenia Space) intend to experiment the SPaCIFY technologies and tool chain using two complementary case studies. The first one is a real industrial use case and the experimentation is meant to cover a horizontal slice of the SPaCIFY engineering process. The second one is a simplified use case meant to allow us, in turn, to cover the entire SPaCIFY process. The first case study targets on the one hand the evaluation of the Synoptic modeling language for early system engineering phases, and on the other hand the evaluation of the Altarica-based model-checker, especially with respect to its scalability. During early system engineering phases, the main concern is to ensure the correctness of the system specifications that are defined. Such specifications include system architecture in terms of sensors, actuators, and command/control mode automata (in a later phase, the control algorithms are plugged on this architecture). Errors at this level usually have disastrous impact on the rest of the project, thus ensuring the correctness of the mode automata specifications is of prime importance. For classical satellites, this task is based on expert human reasoning. The increasing complexity of space systems, such as constellations of satellites flying in formation, makes it more and more difficult or even impossible, as the human faces combinatorial explosion. Formation flying is of increasing interest in the space domain, as it offers the possibility to distribute complex instruments over several spacecrafts. Indeed, the performance of today's space science missions is restricted by limitations on instrument size. The capabilities of telescopes, interferometers, coronagraphs, etc. are directly related to the size of their optics. Thus, several European Formation Flying missions composed of 2 or 3 satellites are under preparation. The main specificity of such missions is the importance of the coordination among satellites, especially for collision avoidance. Within the SPaCIFY project, we aim to model the command/control and FDIR (Failure Detection Isolation and Recovery) automata in Synoptic and to use the Altarica-based model-checker to automatically check the expected properties of system the architecture. The resulting models may become the actual specification, while the model-checking results would definitely bring confidence in the specifications. The second case study targets the experimentation of the entire SPaCIFY process by going through all its phases. This experimentation will allow us to evaluate the SPaCIFY process in a realistic context and identify possible areas where tuning may be necessary. For practical reasons, the experimentation will be based on a simplified but sufficiently representative flight software for a classical satellite.

6 Conclusion

The SPaCIFY ANR exploratory project proposes a development process and associated tools for hard real time embedded space applications. The main originality of the project is to combine model driven engineering, formal methods and synchronous paradigms in a single homogeneous design process. The domain specific language Synoptic has been defined in collaboration with industrial end-users of the project combining functional models derived from Simulink / Stateflow and architectural models derived from AADL. Synoptic provides several views of the system under design: software architecture, hardware architecture, dynamic architecture and mappings between them. Synoptic is especially well adapted for control and command algorithm design. The GALS paradigm adopted by the project is also a key point in the promoted approach. Synoptic language allows to model synchronous islands and to specify how these islands exchange asynchronous information by using the services of a dedicated middleware. In particular, SPaCIFY proposed a contract approach to the specification of the relations between applications and the middleware.

References

[1] A. Arnold, G. Point, and A. Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2-3):109–124, 2000. IOS Press.

[2] P. Bieber, C. Bougnol, C. Castel, J.-P. Heckmann, C. Kehren, S. Metge, and C. Seguin. Safety assessment with AltaRica - lessons learnt based on two aircraft system studies. In *Proceedings of the 18^{th} IFIP World Computer Congress (WCC'04)*, 2004.

[3] U. Brinkschulte, A. Bechina, F. Picioroagă, and E. Schneider. Open System Architecture for embedded control applications. In *International Conference on Industrial Technology*, volume 2, pages 1247–1251, Slovenia, December 2003.

[4] C. Brunette, J.P. Talpin, A. Gamatié, and T. Gautier. A metamodel for the design of polychronous systems. *Journal of Logic and Algebraic Programming*, 78(4):233–259, 2009.

[5] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[6] J. Buisson, C. Carro, and F. Dagnat. Issues in applying a model driven approach to reconfigurations of satellite software. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008.

[7] J. Buisson and F. Dagnat. Experiments with Fractal on Modular Reflection. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, Washington, DC, USA, 2008.

[8] J. Buisson and F. Dagnat. Introspecting continuations in order to update active code. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008.

[9] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Lett.*, XXIV(2), 2004.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[11] D. Chemouil. The design of space systems: an application to flight software. In *Model-Driven Engineering for Distributed Real-Time Embedded Systems' Summer School*, Brest, France, September 2006.

[12] ESA. European space agency. ground systems and operations - telemetry and telecommand packet utilization (ecss– e– 70), January 2003.

[13] A. Cortier et al. Synoptic: a domain specific modeling language for embedded real-time flight software design. In *ERTS*, Toulouse, France, May 2010.

[14] A. Griffault and A. Vincent. The Mec 5 model-checker. In Springer, editor, *Proceedings of the 16th International Conference on Computed Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 13–17, Boston, MA, USA, July 2004.

[15] J. Hugues, B. Zalila, and L. Pautet. Combining Model Processing and Middleware Configuration for Building Distributed High-Integrity Systems. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, Washington, DC, USA, 2007.

[16] D. Isovic and G. Fohler. Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints. In *21st IEEE Real-Time Systems Symposium (RTSS'2000)*, pages 207–216, Orlando, USA, November 2000.

[17] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th int. conference on generative programming and component engineering*, New York, NY, USA, 2006.

[18] C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann, and S. Metge. Advanced multi-system simulation capabilities with AltaRica. In *Proceedings of the 22^{nd} International System Safety Conference (ISSC'04)*, 2004.

[19] C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann, and S. Metge. Architecture patterns for safe design. In *Proceedings of the AAAF 1st Complex and Safe Systems Engineering Conference (CS2E'04)*, 2004.

[20] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[21] G. Point and A. Rauzy. AltaRica – constraint automata as a description language. *European Journal on Automation*, 33(8-9):1033–1052, 1999. Hermes.

[22] E. Schneider. *A middleware approach for dynamic real-time software reconfiguration on distributed embedded systems*. PhD thesis, INSA Strasbourg, 2004.

[23] B. Sprunt, J.P. Lehoczky, and L. Sha. Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm. In *IEEE Real-Time Systems Symposium*, Huntsville, USA, December 1988.

[24] A. Toom, T. Naks, M. Pantel, M. Gandriau, and I. Wati. GeneAuto: An Automatic Code Generator for a safe subset of SimuLink/StateFlow. In *European Congress on Embedded Real-Time Software (ERTS)*, 2008.

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