A Component Model for On-board Software Applications

Marco Panunzio and Tullio Vardanega
Dept. of Pure and Applied Mathematics, University of Padova, Italy
{panunzio, tullio.vardanega}@math.unipd.it

Abstract

Component-based development has become more common news than exotic novelty in mainstream industry. Its adoption has accrued high facilitation potential for software reuse and better organization of the product. Surprisingly however, component-oriented approaches have fared far worse in the domain of high-integrity real-time systems. One of the fundamental reasons of this difficulty stems from the larger incidence of extra-functional concerns than in conventional industry and the more stringent demands on the ultimate product quality. It is not intrinsic however that component-based software engineering cannot just make it in that domain. Hence adopting it may be an effort worth pursuing. In this paper we present the main conceptual and methodological steps taken by an initiative of the European Space Agency aimed at the creation of a component model suited for the development of on-board software.

1 Introduction

The development of on-board software for the space domain, as well as high-integrity real-time software in general, is subject to stringent requirements at both process and product level. The way the software product implements its functional services must meet extra-functional demands in the dimension of, e.g., time and space predictability, dependability, safety, security. The high-integrity nature of the domain is reflected in the standards [6] that regulate the software development process and set the criteria to prove that the resulting product exhibits the expected level of quality while providing the required functional services.

As a general trend, which equally applies to the space domain, the industrial suppliers are faced with a steady demand for increasingly complex functional services together with a contraction of the development schedule and faster time-to-market. Any error or misjudgement in responding to these needs may result in severe cost overruns.

In response to these needs, the European Space Agency (ESA) has recently launched a string of strategic activities. The technical solution chosen for satellite on-board software is the definition of a *software reference architecture*, to be adopted in as many ESA missions as possible. The initial steps of the initiative are being coordinated by a working group called SAVOIR-FAIRE, comprised of staff members of ESA, national space agencies, prime contractors and main software suppliers. One of the authors of this paper has been an active member of that group.

The prerequisite for the realization of the software reference architecture is a thorough investigation and comprehension of the body of knowledge [8] mobilized by the design of embedded real-time systems. Following [12], we contend that the key ingredients on which the overall approach should be based are: (i) a component model; (ii) a computational model; (iii) a programming model; and (iv) a conforming execution platform;

The component model enables the design of software as a composition of reusable software units. The computational model relates the design entities of the component model, their execution needs and their extra-functional properties in the dimension of concurrency, time and space to a framework of analysis techniques to ensure that the architectural description of the system is statically analyzable. The component and computational model address some overlapping concerns and thus their relationship in the approach is delicate, especially in the regard of the semantics assumed for extra-functional properties. The programming model warrants that the implementation of the design entities conforms with the assumptions, constraints and semantics allowed by the computational model. This need can be satisfied by a tailored subset of a concurrent programming language that rejects all constructs that entail non-conforming semantics. The execution platform that fits our concept actively preserves at run time the software and system properties asserted by static analysis.

Given this overall vision, in this paper we focus on the aspects related to the design of the component model. In particular we reason on how the industrial needs of the domain influence the conception of the chosen technical solution. We illustrate the main choices and underlying principles of the component model that is the cornerstone of the

design activity. We also elaborate on the open problems of the approach which are currently being investigated. Even though the approach may still need refinements for it just entered the realization phase, we uphold its soundness and applicability by drawing from the lessons learned in the ASSERT[1] project [3] over the period 2004-2009.

ASSERT was partially funded by the FP6 of the European Commission, coordinated by ESA, participated by prime contractors, SME, tool vendors and academic partners. ASSERT aimed to the definition of a model-based development process for on-board software. The cornerstones of the ASSERT approach were: (i) a computational model with sufficient expressive power for the application domain; (ii) a correct-by-construction form of Model-Driven Engineering: on inspiration from [4], we constrained the design space so that the user could specify only design entities conforming to the chosen computational model, and we guaranteed the preservation of their properties through a programming model and a coherent execution platform; (iii) rigorous separation of concerns from specification to implementation, centred around full automated generation of the code addressing extra-functional constraints.

One outcome of the project was a proof-of-concept development environment where the user could design platform-independent models of the software. Although the project approach lacked an explicit notion of component model, it was able to perform full automatic generation of platform-specific products in the same style as discussed in this paper. The ASSERT results hence successfully proved the validity of the core principles of our approach and formed the starting point of this investigation.

2 Related work

In [11], the authors discuss the industrial requirements gathered for a component model suitable for vehicular systems. The main drivers for adoption of CBSE are: (i) use of a design approach that fosters the creation of a good software architecture and facilitates the subdivision of the software in small, clearly-defined subprojects, irrespective of any reuse concern; (ii) creation of reusable building blocks used to design and implement a component-based system. Interestingly, (i) is perceived as the key advantage by players experienced with the use of component-oriented approaches, whereas (ii) is for those novice to CBSE.

SaveCCM [7] targets vehicular systems. The design model (specified with an extended subset of UML 2 components diagrams) is turned into models amenable to various forms of static analysis, like timing automata with tasks (for response time calculation) or finite state process models (absence of deadlocks). The framework generates glue code to interface with their multitasking RTOS of choice.

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

Ada-CCM [13] bases on LwCCM[2] and its associated IDL for component specification. The framework generates the containers and connectors for the components. Interfaces are decorated with metadata used to specify timing attributes used to feed the MAST analysis tool[3].

The core principles of the *Predictable Assembly from Certifiable Components* (PACC) project [16] by CMU/SEI are very close to our work in that they base on: (i) a design framework encompassing a Component Model and a set of Analysis Models, which permit the derivation of analysis views specialized for timing, safety and availability analysis; (ii) restriction of the design expressiveness to reflect the assumptions of the applicable body of analysis, and therefore make the resulting software analyzable by definition.

Unlike other component models, our effort still lacks a solid solution for hierarchical components (see section 5.4 for an initial discussion on this problem). A distinctive aspect of our work though is that we seek property preservation also at run time, by active monitoring of the compliance of execution with the analysis stipulations and notification to designated handling authorities in the event of violations.

3 Industrial needs

The ESA initiative is a harmonization of methodologies and technologies around the Agency charter (and its "geographic return" policy), seeking to earn benefits for all involved stakeholders: ESA itself as the procurement agency, prime contractors and software suppliers.

In this section we discuss the main industrial needs, summarised in Table 1, which triggered the initiative and drive the quest for the solution.

ID	Industrial need	Impacts on
IN-01	*Reduced development schedule*	Methodology
IN-02	*Product quality*	Methodology, process, technology
IN-03	*Increased cost-effectiveness of software development*	Methodology, process, technology
IN-04	*Reduced effort intensiveness of Verification and Validation*	Methodology, process
IN-05	*Multi-team development and product policy*	Process, technology, market structure

Table 1. The main industrial needs and the aspects they impact on. No relative priority was set on them.

Reduced development schedule. Future projects will require software to be developed in a shorter schedule because: (i) the definition and finalization of software requirements occur later in the project schedule than they used

[2] http://www.omg.org/technology/documents/formal/components.htm

[3] http://mast.unican.es

to; the mission definition and system engineering phases take longer as they are the project activities where most of the value added resides. Hence, the implementation activities, including software development, are consequently compressed; (ii) the release of the product is usually tied to astronomical events (which constrain the launch window) or other external factors (e.g., the procurement of the launcher). Moreover, staggered releases of the on-board software are required so that electrical integration and HW/SW integration tests can be performed incrementally and the relevant workmanship is more efficiently deployed.

Thus, although the software itself only accounts for a minor fraction of the total satellite cost, delayed software releases may have large impact on the overall project schedule by holding the work of the development team, and consequently increasing cost.

Product quality. The quality exhibited by the on-board software (in both functional and extra-functional terms) shall be no less than attained with current practices. Adhering to a well-defined development process and adopting qualified methodologies and technologies shall stay as central prerequisites to the novel development approach.

Increased cost-effectiveness of software development. The budget availed for software development is not going to rise in the foreseeable future. Instead, it may be cut down in favor of other cost elements. Conversely however, the performance and complexity of core functions of the satellite platform (Attitude and Orbit Control System, Data Handling System, etc.) may well grow in response to the demands of end users (e.g., the precision of the AOCS control laws) while new complex functions may also be required (i.e., formation flying, advanced autonomy, etc.).

This predicament calls for better cost-effectiveness in the software development, thus ultimately increasing the "value" of the software product delivered in the given budget envelope. Cost-effectiveness can be increased by singling out and abating the recurring costs; this permits specialized resources to focus their skills on their side of the problem and yield as much value added as possible off it (the functional contents) and to reduce development costs while delivering the same set of functions.

Abatable recurring costs in this context also arise from those parts of the software that do not directly deliver value added and are not mission-specific, e.g. device drivers, real-time operating system, communication services, archetypal parts of the application software. The more they can be factored in the automated elements of the development the better for the economy of the development.

Reduced effort intensiveness of Verification and Validation. In the space domain, V&V activities are by far the largest and most labor-intensive contributor to the software development cost: they typically range 40% to 60% of the total cost. The new development approach shall therefore strive to contain the labor intensiveness of V&V.

Our quest for this goal bases on a design process centered on two pillars: (i) strict separation of concerns between functional and extra-functional aspects; (ii) application of Correctness by Construction principles [4] at design and implementation level. The envisioned process should facilitate early analysis of the software design and drive the subsequent development within the stipulated bounds.

Multi-team development and product policy. Decomposition of the software product and the possibility to easily and cleanly subcontract part of it to different suppliers are crucial factors to the geopolitical economy of the space domain. This need arises from the "geographical return" policy sanctioned in the ESA charter, which requires to "ensure that all Member States participate in an equitable manner, having regard to their financial contribution, in implementing the European space programme".

4 High-level requirements for the approach

The analysis of lessons learned from past experience, compounded by technical and programmatic discussions held within the SAVOIR-FAIRE group, led to the derivation of high-level requirements on the novel development approach, proceeding from the industrial needs discussed in section 3. Table 2 highlights the core requirements that were assigned highest priority in view of their strategic role in the build up of the methodology and the demonstration.

ID	High-level requirement	Originated from
HR-01	*Separation of concerns*	IN-01, IN-02, IN-04, IN-05
HR-02	*Reuse of software*	IN-01, IN-03
HR-03	*Reuse of V&V tests*	IN-01, IN-02, IN-04
HR-04	*Adoption of CBSE*	IN-01, IN-02, IN-03, IN-04, IN-05
HR-05	*Static analyzability*	IN-02, IN-04
HR-06	*Property preservation*	IN-02, IN-04
HR-07	*Heterogeneous systems*	IN-03, IN-05

Table 2. High-level requirements and tracing to industrial needs.

Separation of concerns. This practice is used to cleanly separate different aspects of software design, in particular functional and extra-functional ones. This discipline permits separate (hence more focused) reasoning on and specification of those concerns. The resulting design facilitates reuse of functional contents independently of extra-functional concerns.

Separation of concerns fosters the realization of software that is cleanly defined according to good software engineering practice. It contributes to attaining quality in the software as the separation of functional and extra-functional

concerns permits a rationalization of V&V activities, which can be independently performed on a per-concern basis. Software suppliers can also benefit from separation of concerns, as they can develop software addressing only functional concerns and validate it in isolation.

Reuse of software. Software reuse is an obviously attractive practice to reduce project development costs. This can happen if "black box" reuse is achieved, as opposed to "almost reuse" that occurs if even slight modifications to existing software are required. Each domain has its own flavor for the desired kind of reuse. For on-board software, our primary aim is the reuse of components across subsequent projects of the same organization; and then reuse across different execution platforms. Currently, the software prime contractors use distinct development approaches and adopt different technologies and execution platforms: the net result is that a software supplier can competitively provide services only in a single supply chain. With the approach we propose, suppliers should be able to furnish software components to different primes.

Reuse of Verification and Validation tests. There is little interest in software reuse without also reusing the results of the V&V activities already performed on it. As the volume of V&V effort required by on-board software is massive, if software reuse does not permit to reuse its V&V value too, then its attractiveness becomes nil.

In a CBSE approach with strict separation of concerns, the approach shall aim to the reuse of the majority of tests for the functional part of the software (i.e. the components). In spite of its importance, this was the only high-priority requirement whose fulfilment is set in a longer time frame due to its technical complexity and the investigation on the normative support to accept the reuse of qualification proofs.

Adoption of Component-Based Software Engineering. In classic CBSE, the whole software is designed as a composition of components that are units of reuse with no separation of concerns in them and among them. Where separation of concerns matters, components are pure functional units: they only comprise sequential code; timing, concurrency and other extra-functional concerns are addressed outside of them. Components are the unit of reuse and the whole development process is geared to enable their reuse in multiple projects and with different execution platforms.

The other fundamental aspect of the approach is the verification of properties. In particular we are interested in: (a) *composability*, which is ensured when properties stipulated on individual components hold upon component composition; and (b) *compositionality*, which is the ability to determine a property of a component assembly from some transformation of properties of the constituing components. The composability or compositionality of properties are determined by the methodology that underpins the approach.

Static analyzability. Early static analysis can confirm (or outline problems on) the software design in the related dimension of concern. The design process and methodology used for the reference architecture shall support the verification at design time of functional and extra-functional properties, through e.g., schedulability analysis, stack analysis, dependability analysis.

Property preservation. When certain extra-functional properties are used as input for some form of analysis, they become constraints on the system and their validity "a property" [of the system]. In fact they can be considered as the "box" (of output values/effects/behavior) within which the system/component corresponds to the stipulations. In order to ensure consistency between the analyzed model of the system and the system at run time, those properties have to be actively enforced or preserved during system execution.

Address heterogeneity of systems. It is not realistic to hypothesise the unification of the scattered range of technologies currently adopted by the various key players of the domain. Hence our approach should support interoperability at least between: (i) components written in Ada and C; (ii) the execution platforms in use by prime contractors.

5 Component model

Drawing from the earlier discussion we can assert that the success of the development methodology we are after requires, among others: (i) the adoption of rigorous separation of concerns, in particular between the functional and extra-functional dimensions; (ii) the verification of properties of components and of component assemblies.

Separation of concerns is enforced in CBSE by careful allocation of concerns to the three distinct software entities of the approach: the *component*, the *container* and the *connector*. This requirement imposes the restrictive interpretation of the entities that we discuss below.

5.1 Software entities

Component. The most common definition of component was probably by Szypersky in [15]. A more general definition appears in [5]: "A software component is a software building block that conforms to a component model. A Component Model defines standards for (i) properties that individual components must satisfy and (ii) methods, and possibly mechanisms, for composing components."

A component provides a set of functional services and exposes them through an interface called "provided interface". For this reason the component is also an encapsulation unit, as the only way to interact with it is through the services of the provided interface. The component may need some services from other components or the environment in general in order to successfully provide its own services. Those required services are gathered in the "required

interface" of the component. For this reason, the component is assembled with other components in order to completely satisfy its required interface. The component, in essence, stipulates a sort of contractual agreement by which, if all its functional needs (required services) are satisfied, as well as other possible assumptions on the environment or execution platform, then it is able to provide its functional services.

As a distinguishing feature of our approach, components are pure functional units: they only contain functional code that specifies the sequential behaviour of the component. Every other extra-functional concern is excluded from the component implementation and is addressed by the two other software entities, the container and the connector, or, via them, by the execution platform. Extra-functional attributes are specified as decoration of component interfaces with appropriate language-level constructs.

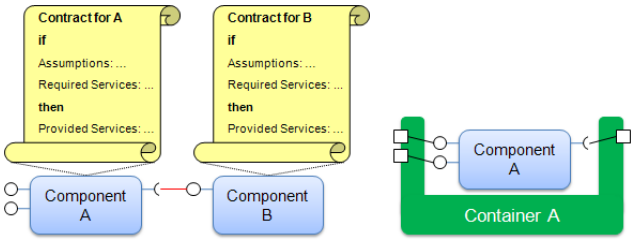

Figure 1. **Left**: Components with contractual interfaces and binding between them. **Right**: a container and its component.

Container. The container in Figure 1 can be regarded as a software wrapper placed around the component, which is responsible for satisfying the extra-functional requirements set on the component. In general, there can be no direct communication to a component: that is instead mediated by the enclosing container which acquires and exposes the provided and required interfaces of the enclosed component. The container also mediates the access of the component to executive services provided by the execution platform.

Connector. The connector [10] is the software entity responsible for the interaction between components, as mediated by containers. The role of the connector is to decouple interaction concerns from functional concerns. This means in practice that components do not directly handle the interaction with other components. In this way, the functional code of a component can be specified independently of: (1) the component(s) it will be later bound to; (2) the cardinality of the communication; and (3) the location of the other parties. This is necessary as components are designed in isolation and their binding with other components is a later concern, or may vary across reuse contexts.

The static nature of the target systems of the space domain reduces the variety of necessary connectors to a few basic kinds, which are required to perform function/procedure calls, remote message passing or data access (I/O operations on a file in the safeguard memory). This also means that we do not require an approach for the creation or composition of complex connectors [14]. We only need a small, closed language for capturing them all. More complex connector kinds may be required when higher guarantees are demanded for some remote communications, for example "exactly-once" semantics, to convert the underlying representation of data, or to define complex task release protocols.

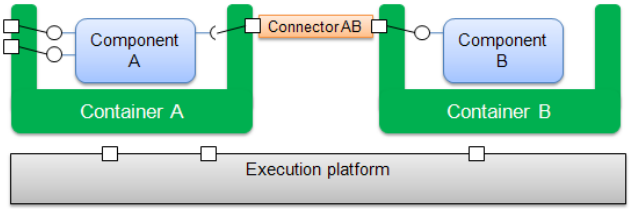

Figure 2. A connector that realizes the communication between two containers. The execution platform is also shown.

5.2 Execution platform

With the generic term "execution platform" we identify the middleware, the real-time operating system/kernel, communication drivers and the board support package for a given hardware platform. Components, containers and connectors use the various services of the execution platform according to the concerns they address.

In contrast to components that only include functional code and thus are independent of the execution platform, specific implementations are needed for containers and connectors per execution platform.

5.3 Design and implementation entities

In this section we elaborate on the design and implementation entities that support our CBSE approach: 1) data types and interfaces; 2) component types; 3) component implementations; 4) component instances; 5) component bindings; 6) description of the physical architecture; 7) containers and connectors.

Data types and interfaces. Data types and interfaces are the basic entities in our approach. An interface is a set of one or more operations, with a defined operation signature, determined by an operation name and an ordered set of parameters, each one with a direction (*in*, *in out*, *out*) and a parameter type chosen between the defined types.

Component type. The component type is the design entity that denotes a reusable software asset. A designer specifies a component type to provide a specification of the functional services of the component. The specification comprises the list of provided and required interfaces and zero or more attributes, which are typed members with a visibility modifier (read-only, read-write, private).

Component implementation. A component implementation is a concrete realization of a component type. A component type may in fact have distinct implementations, which may also differ for the implementation language. A component implementation implements all the functional services of its type, includes the functional (source) code of those services and the necessary packaging information.

Additionally it declares the platform constraints for the implementation (e.g., assumptions on the processing unit or execution platform), and some limited extra-functional constraints. The latter provision acknowledges that some implementation may set specific constraints to preserve operational correctness: an implementation of a control law may for example work correctly only if executed within a selected range of frequencies, e.g., [8Hz, 10Hz].

Component instance. Component instances are instantiated from a component implementation and deployed on a processing unit. At this level we decorate the services of the provided interface with: (i) timing properties, e.g., a periodic/sporadic release pattern and its period/minimum inter-arrival time; (ii) sizing properties, e.g., the memory footprint and stack size; and (iii) communication properties, e.g., a quota on data to be sent.

With this approach, component types and implementation preserve their pure functional nature and the extra-functional properties can be added later as a sort of superimposed layer.

Component bindings. Component bindings are set by the user at design time between a required interface and a provided interface of component instances. The binding is subject to static type matching between interfaces to ensure that the providing end fulfils the functional needs of the requiring end. Compatibility of interfaces in the extra-functional dimension is confirmed by static analysis; for example, schedulability analysis verifies the fitness of the timing properties of interfaces [1].

When bindings have been set, other extra-functional properties can be specified: (i) synchronization properties; (ii) queuing properties, e.g., queuing protocols and queue sizes; (iii) end-to-end timing properties, e.g., end-to-end deadlines on a call chains across components.

Physical architecture. The physical architecture provides a description of the system hardware. The following elements are described: (i) *processing units*, units that have general-purpose processing capability; (ii) *avionics equipment*, sensors, actuators / storage memories / remote terminals; (iii) the interconnections between the elements above, in terms of buses, point-to-point links, serial lines.

Once the physical architecture has been defined, the last step to perform in the design space is the allocation of component instances to processing units.

Containers and connectors. When a component instance is allocated to a processing unit, we can generate the container within which the component is deployed on top of the execution platform of the processing unit. If component bindings have been set, then it is possible to also create the connectors for them. The internal structure of containers depends on the extra-functional properties required for the components they may embed. In fact, for each allowable computational model we need a small taxonomy of them. Interestingly, this permits to afford the development of deterministic rules to automatically generate containers from the attributes set on the model.

For every pair (computational model, execution platform), we can factor out the whole set of allowable containers and connectors in a library of code archetypes, which vastly simplifies automatic code generation [2].

5.4 Preliminary assessment and future work

The component-based approach we described in this paper has just entered the realization phase. Hence we cannot yet evaluate empirically its effectiveness. However, we are fully confident in the feasibility of the approach, at least in as far as it builds on the results of the ASSERT project.

Table 3 shows how the requirements of table 2 are met by our component model.

ID	High-level requirement	Fulfilled by
HR-01	*Separation of concerns*	Careful allocation of concerns to components, containers, connectors
HR-02	*Reuse of software*	Components encompass only functional concerns
HR-03	*Reuse of V&V tests*	Under investigation
HR-04	*Adoption of CBSE*	Component model tailored for the space domain
HR-05	*Static analyzability*	Components encompass only functional concerns, extra-functional attributes drive automated generation of containers/connectors, computational model
HR-06	*Property preservation*	Archetypes for the generation of containers/connectors, conforming execution platform
HR-07	*Heterogeneous systems*	Data types, connectors, components independent from execution platform

Table 3. Summary of how high-level requirements are fulfilled by the proposed component model.

Besides technological and implementation concerns, we are currently investigating two grand challenges: (i) reuse of V&V tests; and (ii) support for hierarchical composition.

The qualification of a software artifact rests on the verification performed on it (by static analysis and test), which

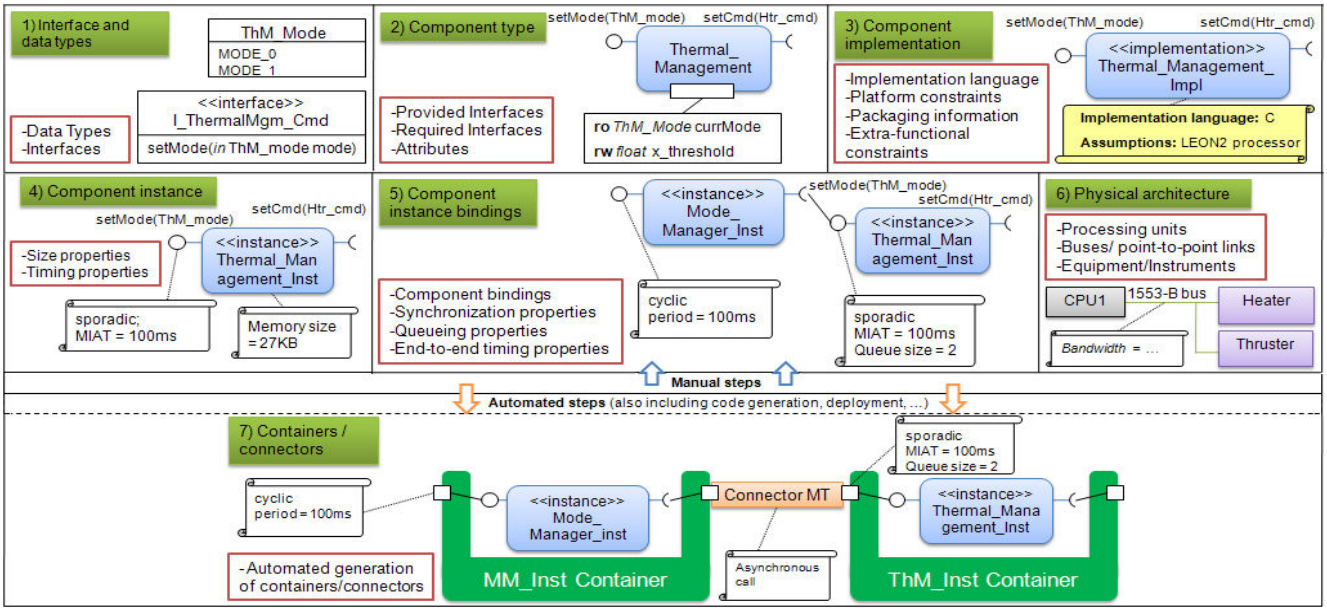

Figure 3. Summary of the design and implementation steps of our approach.

sanction that the software fulfills specific functional and extra-functional requirements. When the software asset is reused, it does in principle lose its qualification status for those requirements. We are investigating the methodological and technical ramifications of wanting to reuse functional tests, and the implications of that on the design process and the project management.

At the same time we are also investigating how to support hierarchical decomposition of components via the creation of child components with interface delegation from a parent component. In fact, this is especially attractive for the attainment of visibility control and mastering of the design complexity, more than for containment relationships between components per se.

Unfortunately, the extra-functional dimensions of the problem complicate the picture considerably. Decomposition along the functional (structural) dimension coupled with attribution of extra-functional properties to the interfaces of child components conflicts with the black-box nature required for the parent component. The solution to this problem shall preserve composability and compositionality, which are key to our component model.

6 Evaluation criteria

We now enumerate the quantitative and qualitative evaluation criteria against which we shall evaluate our proposal.

Overhead of the infrastructure. The cost in terms of memory footprint of the component model infrastructure (containers, connectors, distribution and communication transparency) affects the attractiveness of the approach when for use in systems with scarce memory resources. This criterion may also indirectly impact the engineering quality of the software product: to compensate for a high-overhead infrastructure, designers may tend to create bigger, but not necessarily functionally cohesive components.

The time overhead is less critical challenge: we value predictability over performance.

Containers and connectors. In our approach, containers and connectors are not represented in the (platform-independent) design space, as they are implementation entities that belong to a platform specific model. Alternative approaches may require their specification by the designer in the design space. In the former approach we also dispense with the problem of choosing the formalism and the diagram kind to represent containers and connectors.

Code generation. In our approach we aim at complete generation of code addressing extra-functional concerns. On the contrary, for what concerns functional code, we do not support in our infrastructure the generation of code, mainly due to the numerous formalisms and technologies necessary to cover all the domain needs.

Functional code. Component implementations shall include code that is void of extra-functional semantics. For code not produced with an editor that actively enforces that constraint, and thus extensively for legacy code, some sort of automated screening should be performed to assess the compliance of that code with the inclusion constraint. This provision is very important as the incorporation of some legacy is a must in all long-lived industrial domains.

Compatibility with IEEE 1471. IEEE Std-1471[9], a.k.a. ISO/IEC 42010:2007, describes recommended practices for the architectural description of software-intensive systems. A component-based approach addresses scores of aspects related to the creation and description of an architecture, in particular the various concerns of the approach, the viewpoints, and the supported views.

The architectural description language is another important topic; however, consideration of it was intentionally omitted in this paper for the sake of space and of priorities.

Analysis support. The capability of the development framework to support effective analysis techniques, by extraction or transformation of design information into the analysis formalism is central to high-integrity real-time domains and as such is addressed in most approaches.

Unfortunately however, insufficient attention is devoted to the return of the analysis results in the user model. What is wanted is the ability to feed complex analysis equations and back propagate results in the design space as attributes of the user-level model entities, with the highest degree of automation and the minimum effort for the user.

Complexity of use. The learning curve and complexity of use of the component model from the point of view of both a software integrator and a software supplier.

7 Conclusions

The results of the ASSERT project formed the starting point of our endeavor. ASSERT showed the feasibility of a development approach for high-integrity real-time systems centered on separation of concerns, correctness-by-construction, and property preservation.

In this paper we presented elements of a vast initiative by the European Space Agency to address the growing cost and time of development of satellite on-board software which is at odds with the required faster time-to-market and stable or decreasing budget allocated to software teams.

We presented the key industrial needs of the domain. We then discussed the high-level requirements that were elicited in response to them by an appointed expert group. Component-based software engineering was selected as the development paradigm. Therefore, we are enriching the consolidated results of ASSERT with a comprehensive notion of component model and strategies for reuse of components. The component model we described in this paper is centered on strict separation of functional and extra-functional concerns and rigorous allocation of them to components, containers and connectors. The solid and proven base of the approach gives us confidence on the soundness of our effort, which will be confirmed with a demonstrator and an industrial assessment of its actual reuse potential.

Acknowledgments. The views presented in this paper are the authors' only and do not necessarily engage those of the other members of the SAVOIR-FAIRE WG. This work was supported by the Networking/Partnering Initiative of ESA/ESTEC and by the CHESS project, ARTEMIS JU grant nr. 216682.

References

[1] M. Bordin, M. Panunzio, and T. Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[2] M. Bordin and T. Vardanega. Automated Model-Based Generation of Ravenscar-Compliant Source Code. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[3] M. Bordin and T. Vardanega. Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In *Reliable Software Technologies - Ada-Europe*, 2007.

[4] R. Chapman. Correctness by Construction: a Manifesto for High Integrity Software. In *ACM International Conference Proceeding Series; Vol. 162*, 2006.

[5] M. Chaudron and I. Crnkovic. *Component-based software engineering, chapter 18 in H. van Vliet, Software Engineering: Principles and Practice*. Wiley, 2008.

[6] European Cooperation for Space Standardization. Software general requirements - ECSS-E-ST-40C, 2009.

[7] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - A Component Model for Safety-Critical Real-Time Systems. In *Proceedings of the 30th Euromicro Conference*, pages 627–635, 2004.

[8] T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 40(10):32–40, 2007.

[9] IEEE. Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000, 2000.

[10] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proc. of the 22nd Int. Conf. on Software Engineering*, pages 178–187, 2000.

[11] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *7th Int. Symposium on Component Based Software Engineering*, pages 146–161, 2004.

[12] M. Panunzio and T. Vardanega. On Component-Based Development and High-Integrity Real-Time Systems. In *Proc of the 15th Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, 2009.

[13] Patricia López Martínez and José M. Drake and Pablo Pacheco and Julio L. Medina. Ada-CCM: Component-based Technology for Distributed Real-Time Systems. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, 2008.

[14] B. Spitznagel and D. Garlan. A Compositional Approach for Constructing Connectors. In *Working IEEE/IFIP Conference on Software Architecture*, pages 148–157, 2001.

[15] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd ed. Addison-Wesley Professional, Boston, 2002.

[16] K. C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.