View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Journal of Systems Architecture 60 (2014) 770–781

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

An architectural approach with separation of concerns to address extra-functional requirements in the development of embedded real-time software systems

Marco Panunzio *, Tullio Vardanega

Department of Mathematics, University of Padova, via Trieste 63, 35121 Padova, Italy

ARTICLE INFO

Article history:
Received 16 February 2012
Received in revised form 22 March 2014
Accepted 9 June 2014
Available online 19 June 2014

Keywords:
Embedded real-time systems
Extra-functional properties
Software architecture
Component-based software engineering
Separation of concerns

ABSTRACT

A large proportion of the requirements on embedded real-time systems stems from the extra-functional dimensions of time and space determinism, dependability, safety and security, and it is addressed at the software level. The adoption of a sound software architecture provides crucial aid in conveniently apportioning the relevant development concerns. This paper takes a software-centered interpretation of the ISO 42010 notion of architecture, enhancing it with a component model that attributes separate concerns to distinct design views. The component boundary becomes the border between functional and extra-functional concerns. The latter are treated as decorations placed on the outside of components, satisfied by implementation artifacts separate from and composable with the implementation of the component internals. The approach was evaluated by industrial users from several domains, with remarkably positive results.

© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).

1. Introduction

Embedded real-time systems in general are characterized by two distinctive features: (1) they are resource-constrained, since most often only scarce processing power and memory space are available; and (2) the growing incidence of requirements that address concerns over and above system functionality, in the dimensions of time and space determinism, dependability, safety and, increasingly, security [1].

In this work we collectively refer to those requirements as *extra-functional*, to signify that, while they do not concur to the functional activity of the system, they crucially contribute to the ultimate quality of the system. We also use the term *property* for a feature of an implementation item that provably meets a requirement placed on the specification of that feature. We therefore have extra-functional requirements and extra-functional properties. In this paper we refer to the latter by the shorthand EFP.

Understanding, providing and asserting EFP has an increasingly large and costly effort footprint on the development process in a variety of application domains. The embedded real-time software systems industry therefore seeks ways to accommodate attention for EFP in their otherwise consolidated development practices without breaking the integrity of the overall process.

The central tenet of this work is that the adoption of a sound software architecture helps achieve a clear-cut and composable apportionment of development concerns. With that, EFP can be addressed aside from, yet in coordination with, the functional dimension, in keeping with the established principle of separation of concerns [2].

Interestingly, once the step is taken to put the software architecture at the center of the development strategy, product line concerns can be accommodated by elevating that notion to the stipulation of a *reference* software architecture. We draw from [3] that the reference software architecture is a common and agreed architectural framework capable of addressing all the relevant industrial needs, providing a recurrent solution to the development of a certain class of software systems. To that we add the capability of operating as a single and consistent basis for addressing EFP.

This paper reports on the lessons learned in pursuit of that vision in a large and encompassing research program that progressed along two parallel and complementary lines. One line of the research took place as part of an initiative launched by the European Space Agency (ESA) targeting the definition and realiza-

* Corresponding author. Tel.: +39 0498271359.
E-mail addresses: panunzio@math.unipd.it (M. Panunzio), tullio.vardanega@math.unipd.it (T. Vardanega).

http://dx.doi.org/10.1016/j.sysarc.2014.06.001
1383-7621/© 2014 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).

tion of a reference software architecture that should steer the development of on-board software for satellites across all of its software supply chain, using the component-based approach described in [4,5]. That initiative started from the capture of all product-line needs by the domain stakeholders and concluded with their validation mapping to the chosen reference software architecture. The other line of research occurred within the ARTEMIS JU CHESS project[1] ("Composition with Guarantees for High-integrity Embedded Software Components Assembly" 2009–2012), which aimed at the realization of a model-based component-oriented approach for the development of embedded real-time software systems for telecom, space, and railway applications [6]. All industrial parties from both actions subscribed from the outset to the principle of addressing EFP separately from the functional dimension. In doing so, they were witness to the unifying power of the software architecture concept and to the evidence that EFP were indeed addressed at a distinct level of abstraction as well as at a distinct step in the development process, in overlay to the functional specification of the software.

The remainder of the paper is organized as follows. Section 2 recalls the essential aspects captured in the concept of software architecture and argues why that concept is useful to address concerns in dimensions other than functional. That discussion proceeds into an original interpretation of the notion of reference software architecture and of its founding principles. Section 3 relates the work presented in this paper with state-of-the-art approaches that address similar goals. Section 4 describes the essential details of the proposed approach, with special focus on the way it assists the specification and assures the fulfilment of the extra-functional properties expected of the system. Section 5 presents an instantiation of the reference software architecture to a variety of industrial domains, and discusses how it meets the stakeholders' needs captured as part of the research effort. Section 6 discusses four extensive industrial case studies on which the approach presented in this paper was applied and successively evaluated. Section 7 draws some conclusions and outlines future work.

2. The role and potential of the software architecture

2.1. Common understanding

At odds with its intrinsic centrality, the software engineering practice harbors an exceedingly informal and liberal interpretation of the concept of software architecture. Most practitioners regard software architecture as a synonym to software design: reference [7] collects a large set of community definitions that portray the confusion. In actual fact, the software architecture is a much larger scope than that, with ample bearing on the principles that guide the design and evolution of the software system. The definition given in IEEE 1471, later promoted to ISO 42010 [8] clarifies, to our satisfaction, that an architecture is "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution". When applied to software systems, the definition of software architecture captures well the following concerns:

- *Software decomposition*: the organization of the software in terms of parts, so that every individual part has its own architectural cohesiveness (it addresses a single well-defined part of the problem), and the interactions between parts are minimized so as to reduce unnecessary dependencies (i.e., coupling), hence reducing incidental complexity of understanding, verification and validation, operation and maintenance;
- *Externally visible attributes of software "components"*: those attributes represent features or needs that are specific to individual components yet can influence other parts of the software or determine properties of the whole. The other attributes, if any, shall remain as (externally invisible) internal details and will not be used for the overall reasoning at the level of the software architecture;
- *Relationship between software "components"*: how components relate to one another in providing services and fulfilling needs;
- *Extra-functional concerns*: the abstraction level at which extra-functional concerns are to be addressed;
- *External interfaces*: the way the software interacts with the external environment (e.g., by commanding sensors or actuators, or serving external interrupts);
- *Principles for the development and evolution of the software*: the software design process that fixes the rules for development, maintenance and evolution, and dictates the supported form of software reuse;
- *The rules in place to warrant the consistent relationship between all of the above concerns*: a methodology capable of encompassing and consistently harmonizing all the aspects listed above; additionally, the methodology shall provide criteria to determine whether a software part can be included in the system, as it conforms to the principles sustained by the architecture, or it shall be rejected, as its inclusion would break system integrity.

A reference software architecture prescribes the concrete form of the software architectures that shape the specific systems for which it was originally developed. The reference software architecture can thus be regarded as a sort of "generic" software architecture that prescribes the founding principles, the underlying methodology and the architectural practices recognized by the domain stakeholders as the baseline solution to the construction of a certain class of software systems in that domain. Symmetrically, the software architecture of one particular system for a given domain can then be regarded as an "instantiation" to the specific system needs of the reference software architecture for that domain.

2.2. Narrowing to our context

The reference software architecture chosen for the ESA initiative – and later reflected in the whole span of investigation covered in this work – has a number of distinctive features, which are best introduced here before proceeding further.

First of all, the reference software architecture includes a companion component-oriented development methodology. Software systems in the industrial domains that adopt that methodology are therefore developed by defining (or reusing) components and by creating assemblies of them. The interpretation of components varies with the specific goals of the approach they serve. Numerous definitions for them indeed exist [9–11]. Suffice it to say for now that a component is the unit of design and of encapsulation of our approach: we return to this definition later in this paper.

Secondly, the reference software architecture was formulated so as to support the principle of separation of concerns. This is a long-known but much neglected practice first advocated by Dijkstra in [2], which strives to part the various aspects of software design and implementation so as to enable separate reasoning and focused specification for each of them. Separation of concerns is applied to the component model that rests at the very core of the reference software architecture. We consequently pursue it at a level of abstraction much higher than programming, as done

[1] http://www.chess-project.org/.

for example with aspect-oriented programming [12]. In our approach:

1. Components comprise functional code only, which is to be strictly sequential: extra-functional properties concerning tasking, synchronization and timing (and prospectively fault tolerance) are dealt with outside of the component by the component infrastructure.
2. The extra-functional requirements that the user wishes to set on components are declaratively specified by decoration of component interfaces with an ad hoc annotation language;
3. The realization of extra-functional properties is performed by a code generator, which uses a set of predefined and separately compilable code patterns to generate all of the component infrastructure code for components and their assemblies, and for the entities that realize extra-functional properties. The implementation of the functional code instead stays under the responsibility of the user.

This extent of separation has two principal benefits: (1) it increases the reuse potential of the software by enabling one and the same functional specification to be reused under different extra-functional requirements; (2) it facilitates the automated generation of vast amounts of complex and delicate infrastructural code addressing concurrency, real-time, communication and component interfacing needs in accord with well-defined styles and fully deterministic rules. Industrial experience shows that benefit (1) is only occasionally achieved, as common and stable specifications, effective component breakdown, clean interface design and consolidation are difficult goals to attain in practice for the functional part of the system, which is intrinsically variable owing to its product-specific nature. Benefit (2) instead becomes available much sooner and at a fraction of the cost of (1), with immediate and tangible benefits.

Finally, the reference software architecture shall help achieve composability and compositionality in a by-construction manner, as opposed to the by-correction style of common practice. With [11] and a little narrowing to the context of this work, we have that: composability is achieved when the properties (needs and obligations) of individual components are preserved on component composition and deployment on the target system; compositionality is achieved when the properties of the system as a whole can be derived (economically and conclusively) as a function of the properties of its constituting components. This work in fact seeks the higher-level goal of composition with guarantees, which with [6] we regard as a form of composability and compositionality assured by static analysis, guaranteed throughout implementation, and actively preserved at run time.

2.3. Our interpretation

In light of the preceding discussion, our interpretation of reference software architecture was centered on the following constituents [13], as also captured by Fig. 1:

1. A component model, to design the software as a composition of individually verifiable and reusable software units;
2. A computational model, to relate the design entities of the component model, their execution needs and their extra-functional properties for concurrency, time and space, to a framework of analysis techniques which ensures that the architectural description is statically analyzable in the dimensions of interest [14]; this provision is essential to achieving by-construction guarantees that the end-to-end behavior of component assemblies conforms with the underlying computational model (for its execution and communication semantics), and can thus be fully and soundly verified against the system requirements; the solution adopted in this work follows the principles enunciated in ([15]);
3. A programming model, which consists of a tailored subset of a programming language and a set of code archetypes, and is used to ensure that the implementation of the design entities conforms with the semantics, the assumptions and the constraints of the computational model;
4. A conforming execution platform, which is in charge of preserving at run time the system and software properties asserted by static analysis and is able to notify and react to possible violations of them.
5. A development process centered on Model Driven Engineering (MDE), more specifically in conformance with the Model Driven Architecture initiative by the OMG, much in the guise of the ideas originally presented in [16];
6. The provisions for domain-specific aspects, which complement the approach, yet are consistent with all its underlying principles.

Casting the above boundaries of action into a model-driven component-oriented development methodology, we can draw a direct parallel with the definition of software architecture given in Section 2.1:

- The component model addresses the software organization, the externally visible properties of software parts (i.e., components), the relationship between software components, the external interfaces of the software systems and the principles that govern software development and evolution;
- The computational model: (i) addresses the relationships between components, as their interaction shall conform exclusively to the communication semantics explicitly allowed by analysis theory in use; (ii) partly influences the principles on which the software is developed (as it restricts the feasible implementations to solely analyzable software systems); (iii) and defines how the EFP (restricted to timing, concurrency, communication and space in this work) shall be implemented in the system at run time;
- The programming model and the relevant execution platform fall within the development principles, may support the evolution of the software, and are related to the implementation of EFP;
- Model-driven engineering feeds the principles for the development and evolution of the software;
- Domain-specific concerns are mostly related to the external interfaces of the system (interactions with sensors, actuators, devices), the relationships between components and add domain-specific extra-functional concerns or may reinterpret how extra-functional properties shall be fulfilled in the software system.

We can therefore contend that, by our choice of essential constituents, we have provided an original interpretation of a reference software architecture that achieves the properties of interest to our industrial stakeholders.

3. Related work

There are considerable relations and subtle yet important differences between the premises we have laid down in the preceding sections and the relevant state of the art. The discussion of related work is best broken down in two parts: the state of the art on the combination of component orientation with model-driven engineering, and that which regards how extra-functional properties are addressed in software construction.

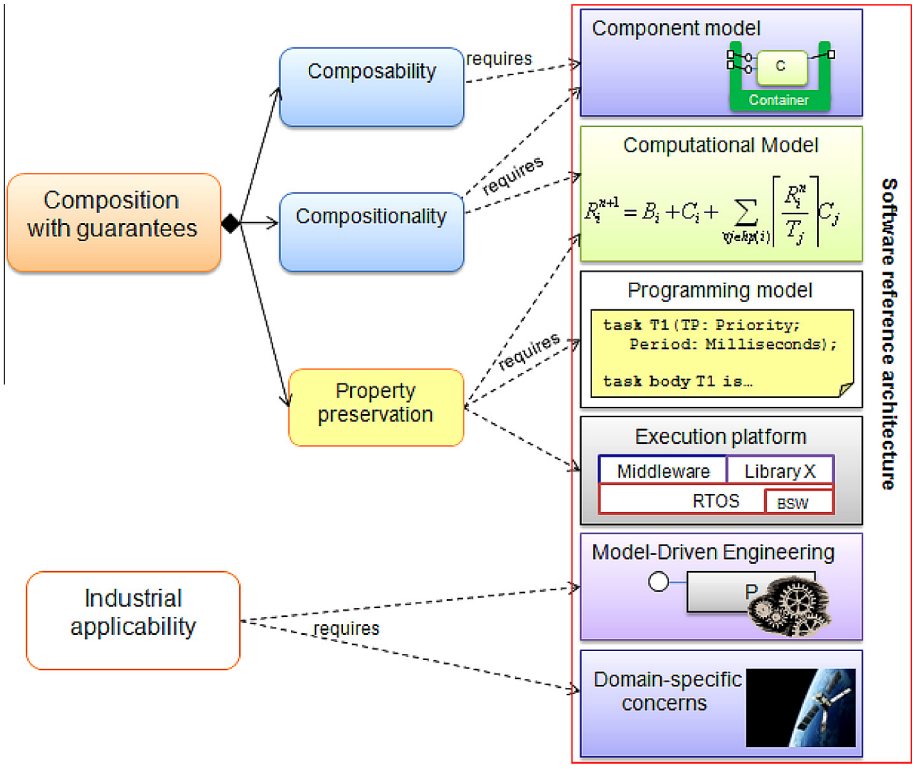


Fig. 1. The goals we set to achieve and the constituents of the reference software architecture.

3.1. Component-oriented model-driven development

We acknowledged earlier in this paper that there is large commonality between our work and the research efforts proceeding from [16]. We consider here those with the closest match.

CoSMIC (Component Synthesis using Model Integrated Computing) [17] addresses the grand challenge of using a modeling and generative programming approach to statically configure and fine tune component middleware tailored to meet quality of service (QoS) requirements for distributed real-time embedded (DRE) applications. As part of that work the authors identify multiple points of integration of model driven techniques with DRE component middleware frameworks, from: (1) configuring and deploying application services end-to-end; (2) composing components into component servers; and (3) configuring application component containers, which fall in the design activity; to synthesizing: (4) application component implementations, (5) dynamic QoS provisioning and adaptation logic, (6) middleware-specific configurations, and (7) middleware implementations, which collectively cover production. Experience shows however that some restrictions and constraints have to be enacted at those points of interaction to achieve "composition with guarantees" at bounded costs, by construction instead of by verification and correction. The cited authors have essentially the same intuition that we placed at the center of our approach, i.e., that model-driven tools can be applied to: (1) analyze relevant characteristics of system behavior, such as scalability, predictability, safety, and security, by using tool-specific model interpreters to feed the analysis tools as needed; (2) synthesize platform-specific code that is customized for particular component middleware and DRE application properties.

PICML (Platform-Independent Component Modeling Language), a central part of CoSMIC [17], is a domain-specific modeling language that enables developers to define component interfaces, QoS parameters and software building rules, and also generates descriptor files that facilitate system deployment. The authors aim to support a component-oriented approach to DRE systems that mitigates the complexity associated with the need to: (1) design consistent component interface definitions; (2) specify valid interactions and connections between components; (3) generate valid component deployment descriptors; (4) ensure that requirements of components are met by target nodes where components are deployed; and (5) guarantee that changes to a system do not leave it in an inconsistent state. PICML has close relations with our work, only we cast the various facets of the endeavor in a single, consistent and coherent software architecture concept, which provides the by-construction guarantees that we seek for our industrial users.

Balasubramanian et al. [18] note that while component middleware technologies raise the level of abstraction used for development, they also promote the decomposition of monolithic systems into assemblies of inter-connected individual components. Functional decomposition of DRE systems into component assemblies and monolithic components helps promote reuse across product lines [19]; however, it can also increase the number of components in the system, with consequences on size and performance. In the cited work the authors propose a "fusion" of analysis and synthesis solutions that help mitigate the problem and obtain a physical assembly, i.e., a component assembly ready for deployment on the physical platform, that has considerably reduced size and improved performance. Once again, this line of work has evident overlaps with ours, with the important difference that the cited works do not seem to pay much attention to achieving a clear-cut apportionment of concerns and responsibilities (and consequently of by-construction guarantees) to distinct elements of the software architecture that should underpin all development.

3.2. Addressing extra-functional properties in software construction

Interestingly, what properties are to be seen as composable and what as compositional is not arbitrary, but it rather depends on

how a component is defined and on the development methodology devised around that concept.

In recent years, a wealth of approaches studied ways to achieve composability for timing properties. Approaches with composable timing properties typically impose early (and often permanent) constraints on the software architecture. As a consequence, any extra-functional constraint imposed on the functionality provided by a component remains as a permanent characteristic of it and impairs the evolution or the reuse of that component. For example, in the GENESYS project based on a Time-Triggered Architecture [20], a component is a HW/SW subsystem that relates to other components via a "linking interface" (LIF). The LIF shall specify at what time instants communication messages are sent and received by the component, how the messages are ordered, and the rate of message arrival. Concurrency aspects (e.g., tasking) are included in the component specification and are necessary for the correct operation of the component function. Unfortunately, however, they do not emerge at the level of the LIF.

In Integrated Modular Avionics (IMA) – as represented by the ARINC 653 standard [21] (which is the predominant incarnation of Time and Space partitioning) – composability for the schedulability results of a task in a partition is achieved at the level of partition, but the scheduling of partitions is rigidly enforced by a cyclic executive, with all the long-known design vulnerabilities of that decision [22].

As a consequence of our choice of seeking separation of concerns, we treat timeliness properties (i.e., response time of operations and end-to-end delay of an activity chain) as compositionality concerns.

The following section expands on this matter and presents our approach to it.

4. Addressing extra-functional properties

As a distinctive character of this work, EFP are dealt with in accord with the principle of separation of concerns, enacted as follows:

- In the implementation space, with the careful allocation of different concerns to designated software entities: the component, the container and the connector, whose respective role is discussed later in this section;
- In the design space, with the use of design views.

4.1. Separation of concerns with different implementation entities

The concepts of component, connector and, less commonly, container, are well-known in the component-oriented community. In relation to them, we adopt a narrower definition that helps serve the principle of separation of concerns [5].

In this work, the component [9,10] is the unit of composition. The software system is built by creating an assembly of components, deployed on an execution platform which takes care of their correct execution. In actual fact, the definition by [9] also carries the requirement of "independent deployment of components", but the industrial needs considered in this work do not require it, as also do many other component-oriented approaches.

A component provides a set of functional services and exposes them through a "provided interface". All the services needed by the component from other components or the environment in general are declared in a "required interface". The component is assembled with other components so as to completely satisfy the functional needs of the required interfaces.

Components in this work are pure functional units; they only contain functional code that specifies sequential behavior: time, concurrency, synchronization, distribution and any other extra-functional concerns thus are not included in the component code. All extra-functional concerns are dealt with by the container and the connector, or, via them, by the execution platform; in all cases outside of the component itself.

The container is a software entity that can be regarded as a wrapper around the component and is responsible for the realization of all extra-functional properties that are specified for the component that it embeds. As a consequence, no direct communication to a component can take place, as all communication to them is mediated by the enclosing containers. The container also mediates the access of the component to the executive services it needs from the execution platform.

The connector [23] is the software entity responsible for the interaction between components, which actually is a mediated communication between containers. The role of the connector is to decouple interaction concerns from functional concerns. Components are consequently void of code that deals with interactions with other components. They communicate with endpoints that are set according to the deployment at system elaboration time.

In this way, the functional code of a component can be specified independently of: (1) the components 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 in different reuse contexts.

The application domains of interest (i.e., space, railways and telecommunication) do not require complex connectors, owing to the static nature of the target systems or of the operational modes that are considered for system modeling. Therefore, support for function/procedure calls, remote message passing through a communication middleware, data access (I/O operations on files in safeguard memory) and event and signals are sufficient to cover virtually all needs. This also means that we do not require an approach for the creation or composition of complex connectors [24]. More complex connector kinds are nevertheless necessary when stronger guarantees on remote communication are required (e.g., by introducing domain-specific communication protocols), for location and representation transparency in more heterogeneous systems.

Figs. 2 and 3 outline the envisaged development process.

The designer first creates or reuses a set of components, and assembles them to fulfill their functional needs. In parallel, the designer defines the hardware and the hardware topology of the system. Then they attach to component interfaces a set of extra-functional annotations.

We describe in detail the specification of extra-functional in the next section, as it is of fundamental importance to our approach. Finally, the designer specifies a set of deployment directives to allocate components to computational nodes. At that stage, the software systems description (the model) can be submitted to various extra-functional analysis; those currently of interest to our industrial stakeholders include: schedulability analysis, state-based dependability analysis; Failure Mode and Effect Analysis (FMEA) and Failure Mode, Effects and Criticality Analysis (FMECA).

EFP are only declared on components. An example of declaration syntax, for the concerns of tasking, concurrency and real-time is presented in [5]. The semantics of those declarations is informed by the chosen computational model, in our case the Ravenscar Computational Model (RCM) [25]. The RCM postulates a concurrent mode of execution that restricts the classic sporadic task model of real-time scheduling theory to a form directly amenable to static analysis in the time and space dimension and efficiently implementable by a small-footprint real-time kernel. By adopting the RCM, the system resulting from the assembly of components can be statically analyzed for its end-to-end response times,

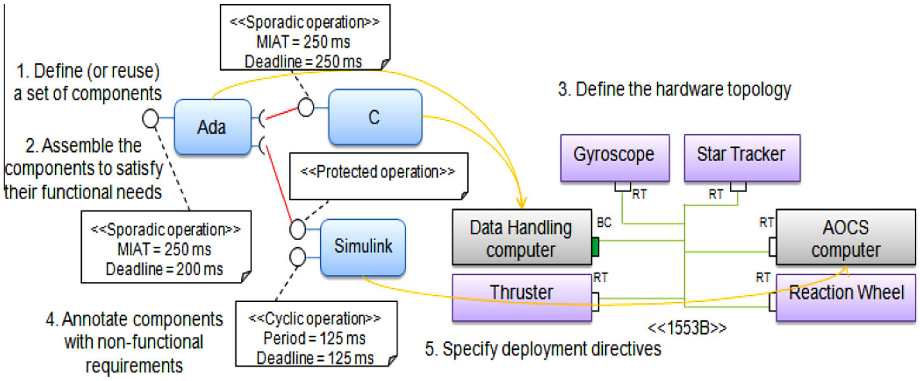


Fig. 2. A sketch of the actions that the user performs in the design space in the proposed approach.

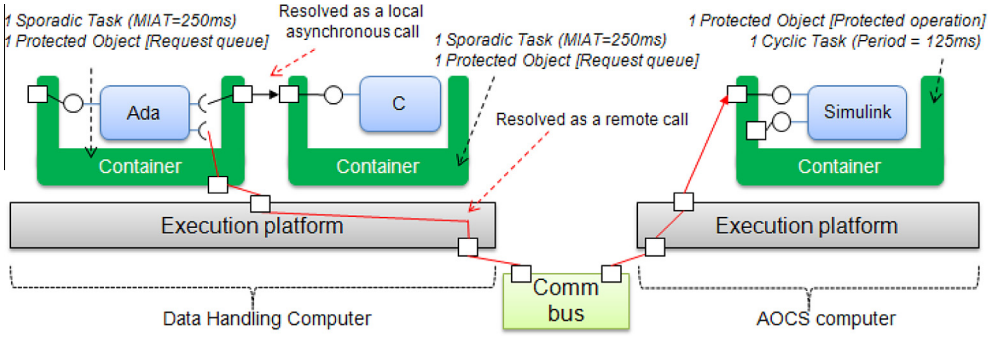


Fig. 3. An outline of the automated generation of implementation entities (containers and connectors).

including the overhead of containers and connectors, with response-time analysis [26].

EFP are realised by model-to-text transformations that generate containers and connectors according to a deterministic (hence provable) mapping of the extra-functional annotations set on components to code implementation artifacts [27].

Using the component bindings and the deployment information, the same pass of model-to-text transformation generates the connectors that realize the desired communication semantics between components, possibly relying on the service stack of the execution platform for transparent remote communication.

Fig. 2 depicts examples of EFP annotations. Owing to the model-to-text transformation rules in place (see [27] for a full description of them), the two sporadic operations declared on the two components implemented in Ada and in C shown in the example, require each the creation of a container which comprises: (1) a protected object (a queue protected with the Immediate Ceiling Protocol [28]) used to deposit incoming requests of execution; (2) a sporadic task released at every 250 ms. (so as to respect the minimum separation between subsequent executions, called Minimum Inter-arrival Time, MIAT for short), to fetch the first request in the queue and execute it, or to block on an empty queue. The operation marked as "protected" on the component implemented in Simulink, requires a protected object (or in alternative form, a semaphore) to provide execution of it in mutual exclusion. The protected object is generated in the container, which also hosts a cyclic task to periodically execute the operation marked as "cyclic". Fig. 3 depicts the containers produced from applying model-to-text transformation to the example.

Remarkably, the generated containers are invariant with respect to chosen deployment of components: it is connectors that take care of understanding it. In our example, the communication between the component implemented in Ada and the one implemented in C is resolved as a local asynchronous call (local, as the components are deployed on the same node; asynchronous, as the provided operation is sporadic, and therefore has its own thread of execution). The call from the Ada component to the Simulink component is mapped instead to a remote communication with message passing, as the destination component is deployed on a remote node.

The Ravenscar Computational Model restricts the nature and behavior of low-level implementation entities such as tasks, monitors, semaphores, underneath components. By using annotations on higher-abstraction entities such as components and interfaces, which are bound to the RCM semantics via model transformations, we are able to faithfully create lower-level analysis models on which accurate model-based schedulability analysis can be performed [29], while gaining the advantages of reasoning at the abstraction level of components in the user model.

4.2. Design views and specification of extra-functional concerns

According to ISO 42010 [8], the architectural description "is organized into one or more constituents called views", each of them being "a representation of a whole system from the perspective of a related set of concerns". Continuing to quote, each view is the expression of a specific viewpoint, that is "a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis".

Our work rests on these concepts, tailoring them to component-oriented development supported by Model-Driven Engineering tool environments.

In current approaches such as state-of-the-art SysML [30] technology, views and viewpoints can be defined by the user, and

consequently vary for distinct systems of the same class. This contrasts with the choice made in this work – arguably more consistent with the cited reference – whereby views are statically defined by the developers of the methodology, in agreement with the domain stakeholders.

If during the construction of the development approach, each view can be ratified as the expression of a single concern, then views become effective and powerful means to enforce separation of concerns in the specification of the system.

Design views also are the means to enforce a given flow of activities for software development: a development process envisions multiple design phases to be carried out – some in potential parallelism, others in some specific ordering – quite possibly under the responsibility of distinct development actors. If the passage from one design phase to the next crosses the boundary of responsibility between different actors with distinct concerns, then design views come as a very handy means to enforce this boundary.

For example, it may be prescribed that the specification of certain design entities or attributes in one view cannot be made without the prior definition of other entities in other views. The corresponding methodological framework should explicitly recognize this prescription and provide means to specify the desired design views accordingly. In MDE terms, the corresponding design views would be implemented by specifying visualization and modification rights for the metaclasses that refer to the entities in question, as defined by the metamodel for the chosen design language. Those rights would then be enforced in the various views to any concrete instance of that metaclass in the design model under development.

Technology solutions exist to evaluate constraints at model creation time, but they do not suffice to satisfy our needs. Not only in fact we require the capability of evaluating constraints related to entities in the model and their static relationships, but we also need to maintain a "stateful context" of when and by whom a model modification (e.g., creation of a new element, specification of an attribute) is performed: we capture those needs in the active design view. The present state of practice for MDE technology does not provide a complete solution to those needs: any current provision for design views is unable to work at the level of the metamodel – where this capability is best provided – and consequently resorts to augmenting the model editor in manners that are neither standard nor portable.

In this work the following design views were defined:

- *Data view*, for the description of data types and events (which are messages generated or received by a component following a publish/subscribe model);
- *Component view*, for the definition of interfaces, components and the bindings between components to fulfill their functional needs;
- *Hardware view*, for the specification of hardware and the network topology;
- *Deployment view*, for the specification of the allocation of components to nodes;
- *Extra-functional view*, where extra-functional annotations are attached to the functional description of components. This view may be further divided in sub-views in order to keep a cohesive control on the concerns of interest: in the reported work, the real-time view and the dependability view were defined as specific sub-views in the extra-functional dimension;
- *Dependability view*, a (sub-) view where specific extra-functional annotations enable various forms of model-based dependability analysis (i.e., error modeling, state-based analysis, FMEA, FMECA) [31] [32]. This view was implemented for industrial users in the CHESS project.
- *Behavioral view*, an optional view where the designer can specify in the model the functional code of component implementations using UML state machines and generate C++ code for it [33]. This view was implemented for industrial users in the CHESS project: in the absence of this view the designer can associate Ada or C/C++ source code written manually to a component implementation;
- *Space-specific view*, an optional view where the designer can specify the use of services related to spacecraft commandability and observability as defined in space-specific standards (e.g., [34]). This view was required for use by the European Space Agency and its industrial suppliers;
- *Railways-specific view*, an optional view where specific extra-functional decorations make it possible to address a number of railways-specific connection protocol concerns.

No telecom-specific view was deemed necessary in CHESS. The technical requirements of the domain were either already addressed by the component model or – when not addressed yet – carried sufficient interest for other industrial domains to be promoted to the domain-neutral part of the component model. An interesting example of a need promoted to core concern was for the decoration for "multiplicities" with component instances and ports. This was handy syntactic sugar to lessen the cluttering of the design space and the burden of specification for the user.

Another reason for not needing a telecom-specific view in CHESS reflects the specific case study chosen by the industrial partner (see Section 5), which solely exercised the domain-neutral part of the component model in conjunction with a reference execution platform for the domain.

All in all, the experience gained from the industrial case studies presented in Section 6 suggests that addressing domain-specific protocols at the component level is best done via dedicated, domain-specific views: their presence facilitates the specialized implementation of the relevant needs in connectors and the associated component infrastructure, and segregates their specification away from the core domain-neutral part of the component model.

Fig. 4 recapitulates the design views of our approach, and highlights the precedence constraints between them (i.e., the entities to be defined in a view depend on information from another view).

Some considerations are in order at this point.

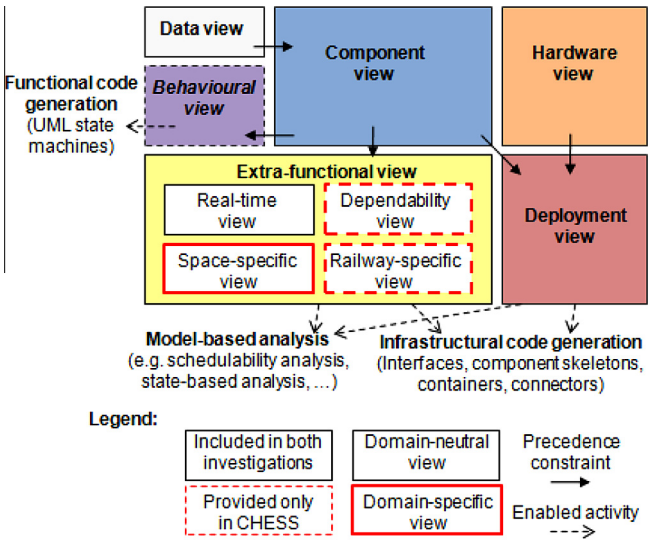


Fig. 4. The design views provided by our approach.

Firstly, the segregation of different concerns to different design views proved to be an effective means to enact a determined design flow across design activities.

Secondly, all domain-specific views were born in relation to extra-functional concerns. This is interesting in two ways: (1) it shows the goodness of fit of the core domain-neutral part of the component model, which was effectively shared by three different application domains; (2) it also shows that ease of activation of a domain-specific view by simply adding domain-specific annotations to the total catalog of extra-functional annotations.

Thirdly, separation of concerns as interpreted in this work enables the generation of functional code completely separate from the rest of the component infrastructural code (i.e., interfaces, component interfaces, containers, connectors). Because of this separation, functional code generation (or implementation) can be performed independently of any later deployment and extra-functional consideration. Moreover, the implementation of components can be directly submitted to functional verification and validation (V&V) (which encompasses unit testing and integration testing of component assemblies) as dictated by the applicable domain standards. The functional V&V results stay valid even if later the extra-functional annotations of components change, as the functional code does not suffer any impact. Only a few functional tests must be performed only after deployment information are specified: those related to the numerical precision of functional algorithms, as they necessarily require execution on target.

5. Instantiation to multiple industrial domains

The vision and the technology described so far in this paper were submitted to the evaluation of industrial users from three distinct domains: space (by two distinct research actions, in the contexts of the European Space Agency and of the CHESS projects), telecom and railways; the latter two also from CHESS. It was therefore necessary to submit the notion of reference software architecture and its constituents presented in Section 2.3 to the industrial users to learn from them which would stay common and which should require adaptations and changes.

Component Model. The same component model, defined in [4,5], was retained in all industrial domains, though with two different implementations.

The ESA side of the work created an incarnation of the component model centered on a domain-specific metamodel (nicknamed "SCM", short for Space Component Model) and developed an associated graphical editor based on the Obeo Designer framework.[2] Obeo Designer supports the static definition of viewpoints, which covered a sizeable part of the needs for design views. It was possible to specify per-view diagrams, graphical representation for design entities, automations and constraints.

In CHESS instead, industrial users required the use of open standards, which resulted in the adoption of UML and the MARTE profile (augmented with project-specific stereotypes) and in the use of the Papyrus[3] technology. CHESS-specific plug-ins were developed so as to provide automation capabilities, model validation checks, and support for the design views specified in the previous section.

Computational Model. All industrial users, except for telecom, converged on the adoption of the Ravenscar Computational Model (RCM) [25]. The two space case studies and the one for railways therefore explored the use of model-based schedulability analysis [29]. The support they were offered allowed the results of the analysis to be seamlessly back propagated to the design model in the form of decoration attributes of the design entities, which could be inspected without needing access to external tools.

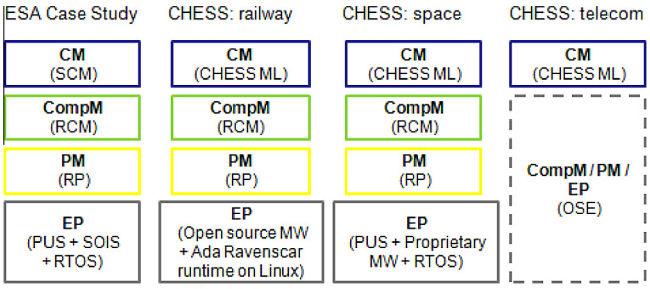


Fig. 5. What the case studies picked as the four constituents of the reference software architecture.

Programming Model. The two space case studies and the one for railways choose the Ravenscar Profile as programming model, which was rather natural after choosing RCM for a computational model. A set of RCM-conformant and property-preserving code archetypes programmed in Ada meet the requirements of the programming model [35,27]. They are used for the automated generation of the component infrastructural software that comprises: component skeletons, component interfaces, containers (including tasking and synchronization code) and connectors for communication concerns, by which we provide for enforcement of the extra-functional properties specified in the design model communication.

Execution platform. This is where the differences appeared among industrial domains.

In the European Space Agency case study, the execution platform comprised: (i) an implementation of the Packet Utilization Standard (PUS) services [34], which, in contrast to traditional code-centric approaches [36], are configured in the PUS view supported in this work, thus at the level of abstraction of the user model; (ii) the Spacecraft Onboard Interface Services[4] (SOIS), a set of standard services used for domain-specific message transmission on network/subnetworks, and communication with distribution transparency; (iii) an open-source operating system conforming with the RCM, for providing tasking, communication and synchronization primitives.

In the CHESS space case study, the target execution platform included: (i) a proprietary implementation of the PUS services; (ii) a proprietary middleware for distribution transparency and communication; (iii) a proprietary operating system with execution semantics conforming with the RCM.

The railways case study used a representative open-source middleware for distributed communication and Ada-level Ravenscar concurrency for execution on Linux.

Not conforming choices. The telecom case study made different choices for all aspects other than the component model and yet stayed broadly consistent with the notion of reference software architecture promoted in this work. The main interest of the industrial user was to try the component model for architectural modeling, the behavioral view for functional specification via state machines, and direct C++ code generation to OSE, a telecom-specific target operating system.[5] That choice covered the concerns addressed in the reference software architecture by the computational model and the programming model, which however were not of high interest of that industrial user.

Fig. 5 recapitulates the choices of each case study.

[2] http://www.obeodesigner.com.
[3] http://www.eclipse.org/papyrus/.
[4] http://public.ccsds.org/publications/SOIS.aspx.
[5] http://www.enea.com/software/solutions/rtos/ose/.

6. Case studies and industrial evaluation

Not all the case studies were end-to-end (from design to code generation). Their common goal was to validate the core approach, the common component model and its goodness of fit for the reference industrial process of interest. Additionally, each industrial user focused on aspects of the methodology of their specific interest.

The evaluation focused on assessing process-related results and product-related results; the industrial teams were also asked to evaluate the proposed approach according to some miscellaneous criteria.

Process-related results

- *PR-1*: Adoption of design views in the design process;
- *PR-2*: Adoption of a component-oriented process with a set of rigorous design steps;
- *PR-3*: Early schedulability analysis with back-propagation of results;
- *PR-4*: Complete generation of extra-functional code related to tasking, synchronization and time-related aspects;
- *PR-5*: Potential for model-based functional code generation;
- *PR-6*: Use of domain-specific views for separate specification of domain-specific concerns.

Product-related results

- *PD-1*: Use of containers for realization of extra-functional properties separately from component code;
- *PD-2*: Use of connectors for implementing communication needs and domain-specific properties;

Miscellaneous criteria

- *M-1*: Maturity of the methodology
- *M-2*: Increased productivity
- *M-3*: Learning curve

6.1. European Space Agency case study

This case study re-engineered a subset of the on-board software of a small yet representative Earth Observation satellite, operating in Low Earth Orbit and including an optical payload to capture images on demand.

The involved industrial user focused on evaluating whether: (1) the component model is able to express the needs for the development of on-board software for satellites; (2) the component model is able to accommodate space-specific needs (namely, the PUS services) in a manner that is consistent with the rest of the approach, and to provide the designer with specification means at the right level of abstraction in the space-specific view.

The case study was performed over 6 calendar months by a senior engineer without previous experience on model-driven, component-oriented methodologies of a small-size on-board software prime contractor, with support from a part-time consultant. The conclusions of the investigation were reviewed by two large software and system prime contractors of the space domain.

The experience provided some interesting feedback.

Firstly, fitting an existing, code-centered design baseline into the novel approach may not be straightforward: the strict adherence to separation of concerns at code level promoted with our method may entail considerable re-engineering for a code base that couples aspects that our approach views as orthogonal and separately addressed (i.e., functional code vs. tasking and synchronization code). Emancipation towards separation of concerns also requires careful attention in determining the right level of abstraction for expressing components in the new software architectural design: all too often, code-centered mentality tends to use components as tasks, which is a patent inversion of abstraction.

Secondly, an important amount of architectural reflection is needed to achieve a specification of components of the most convenient "size" (in quantity of provided services). Designing a great number of "small" components, although individually more easily amenable to reuse, may introduce a lot of complexity in the design due to the bindings of their instances. "Bigger" components are easier to manage as less effort for binding them is required to form component assemblies; yes, they are harder to reuse (the functional requirements of the new context of reuse shall be compatible for reuse to occur), yet the advantage of potential reuse is much more attractive (they represent a complete major function of the system). Curiously, these considerations mirror the difficulties encountered in all matters of software reuse in the last three decades, from object-oriented design and programming, to service orientation.

6.2. CHESS: Space case study

The space case study hosted by the CHESS project was based on Sentinel-3, an Earth observation mission within the Living Planet Program by the European Space Agency. The case study modeled a sizeable subset of the on-board subsystems of that satellite: AOCS (Attitude and Orbit Control System), EM (Equipment Management), PM (Platform Management, an abstraction layer between the SW applications and platform resources such as the 1553B command bus, the on-board time reference, etc.), TR (Thermal Regulation), SADM (Solar Array Drive Mechanism).

This case study focused on assessing whether: (1) the CHESS methodology is compatible with the current process and practices of the domain stakeholder; (2) the quality of the automated generation of code for containers and connectors, targeted towards a proprietary modeling infrastructure of the industrial user; (3) the use of timing-related extra-functional properties of the model and the support for model-based analysis and back-propagation to rapidly iterate the analysis.

The case study was performed by a software engineer with experience on component-oriented methodologies in two iterations: a shorter one for familiarization for the duration of 1 month, and a second one for actual experimental use for the span of 2 months.

6.3. CHESS: Telecom case study

The telecom use case is based on the "Connectivity Packet Platform" (CPP), a system to construct packet access nodes based on IP and ATM transport technologies. The system provides cluster functionalities, redundancy, and fault tolerance. It can be considered as a soft real-time system with a few components with stringent time requirements.

This case study focused on assessing: (1) the fitness of the component model for their development process; (2) the effectiveness of modeling functional code via state machines; (3) the quality and performance of functional code generation from state machines.

The case study was performed by two junior engineers with support by a senior engineer for the integration of their reference execution platform in the approach. It was performed along a time span of 9 months, in two iterations: one short, for familiarization, and one longer for actual evaluation work.

Table 1
I (Demonstrated and considered interesting and promising); Ad (Demonstrated and considered Adequate); NA (Not applicable); HM (High-maturity); m (moderate); l (low); H (high learning curve); M (moderate learning curve).

ID	Result/criteria short name	ESA case study	CHESS: telecom	CHESS: railways	CHESS: space
PR-1	Adoption of design views	I	I		I
PR-2	Component-oriented process	Ad	Ad	Ad	I
PR-3	Model-based schedulability analysis				I
PR-4	Automated generation of extra-functional code			I	I
PR-5	Model-based functional code generation		I		
PR-6	Use of domain-specific views	I	NA	I	I
PD-1	Containers for separate realization of extra-functional properties	I		I	I
PD-2	Connectors for communication needs and domain-specific properties			I	
M-1	Maturity of methodology	HM	HM	HM	HM
M-2	Increased productivity	m	m	l	l
M-3	Learning curve	H	H	H	M

6.4. CHESS: Railways case study

The railways case study was based on applications related to the European Rail Traffic Management Systems (ERTMS).[6] ERTMS comprises two main constituents: (i) the ETCS (European Train Control System), which is used to transmit information to the train driver (train speed, calculation of breaking curves) and permit to monitor the compliance of the driver with prescriptions; (ii) the GSM-R standard, to enable bi-directional wireless communication exchanges between the ground system and the train. The case study concerns a commercial solution for the monitoring and analysis of the strength of the up-link and down-link GSM-R signal in proximity of a high-speed/high-capacity railway line. It analyzes possible interferences on the signal and can discriminate if the interference originates from outside the train or on board; in the latter case the train driver is notified, so as to take appropriate actions.

The case study focused on modeling two subsystems: (a) an "analyzer", which performs the analysis of the GSM-R signal; (b) a "receiver", which receives the analyzed data on signal quality on board and performs the appropriate actions in response.

The "analyzer" is deployed on a laptop, the "receiver" on a dedicated board (simulated with a laptop in the case study).

The case study was small-sized, yet centered on several key features: (i) support for the creation of components written either in C or Ada; (ii) support for multi-node systems; (iii) support for a railways-specific communication protocol to regulate communication from (a) to (b).

Also this case study was based on the re-engineering of an existing code base, and was performed by a software engineer without previous experience in component-oriented methods, plus support from a lead engineer on dependability-related modeling. The case study was performed in two iterations for a total duration of 4 months.

6.5. Summary and evaluation

Table 1 recapitulates the key aspects of the proposed methodology and how they were evaluated by the feedback obtained from the four industrial case studies.

Process-related results

PR-1 is cited as an important factor in the telecom and space domain feedback of the CHESS project and in the ESA investigation.

PR-2 is cited by all case studies as adequate or interesting for the realization of their target system. It helps to split the intellectual work in manageable parts that can be realized (or further refined) independently by the designers.

PR-3 was successfully exercised in the space case study of CHESS, which highlighted its potential if it were introduced in the stakeholder's industrial process.

PR-4 was successfully achieved in the railways case study of CHESS. The code generation was also evaluated by the CHESS space case study, and deemed satisfactorily and promising.

PR-5 was demonstrated successfully on the CHESS telecom case study, where C++ code was generated for functional state machines associated to component implementations ([33]). The approach explored in the cited publication is very promising indeed, though it currently lacks integration with the techniques in place for the generation of extra-functional code (for containers, connectors, and component infrastructure). The difficulty with filling that gap is purely technical, with no bearing on fundamentals, contingent on the stabilization of the heterogeneous technologies used for the various steps of required transformations. Achieving this integration is part of our future work in a recently started follow-on project to CHESS.

PR-6 was realized for the space and railways domains. The three involved case studies confirmed the usefulness of the concept, which was considered interesting for specifying in a coherent manner their domain-specific needs directly in the user model, where the software is being specified.

Product-related results

PD-1 was demonstrated in the CHESS space and railways case studies and in the ESA investigation, and considered a key result.

PD-2 was demonstrated by implementing a basic communication protocol for the railways use case. The protocol can be parametrized by activating the railways view and applying railways-specific annotations to component bindings. Notably, no code of components and containers needed modification, as the only changes when the protocol is activated are carefully isolated in the code generated for the involved connectors. Furthermore, in the ESA case study, we realized a sizeable set of the PUS services, and demonstrated that it is possible to control and parametrize them directly at the abstraction level of the design space.

Miscellaneous evaluation criteria

For what concerns M-1, the maturity of the methodology is considered high by the feedback of all case studies.

For what concerns M-2, the productivity gains are still considered low to moderate. This feedback is mostly due to the prototypal nature of the two toolsets and to the lack of collaborative features for the modeling activity. We however are satisfied by this fair evaluation, as the feedback by industrial users highlights how this is exclusively a technological problem (and in fact the methodology itself was considered mature).

Finally, the learning curve of the approach (M-3) is considered from moderate to high. This highly depends on the previous

[6] http://www.ertms.net/.

exposure of the project partner to MDE and component-oriented approaches (the CHESS space partner is quite familiar with those paradigms, the space partner of the ESA investigation and the telecom and railways partners of CHESS were not). We maintain that most of the difficulty stems from the raised abstraction level proper of any model-driven engineering approach, which can be hard to master for an engineer exposed to it for the first time or after using UML and its derivatives only for documentation.

Furthermore, the difficulty of fitting a novel development approach into the pre-existent industrial process of a stakeholder – an aspect often ignored or downplayed in the scientific literature – shall not be underestimated. In particular, experimenting the novel approach in the place of a "traditional" approach (i.e., not already based on model-driven engineering and without separation of concerns) by re-engineering an existing code base. instead of starting a new architectural design from scratch, bears much hidden risk: the mindset of the actors stays more oriented to "twisting and bending" the novel approach to fit the existing design and code, rather than to applying it to ripe its full advantages.

This feedback also highlighted that an increasing effort shall be devoted to training and dissemination activities, especially in the form of tutorials or reference guides (for example, with the description of architectural patterns for solving recurrent design problems using the component model and the reference software architecture).

7. Conclusions and future work

This paper presented the results of two closely interconnected research efforts targeting embedded real-time systems: one effort was set to establish a reference software architecture for the on-board software of future satellite missions procured by the European Space Agency; the other defined a component-oriented methodology fit for use in the development of telecommunication, space, and railways software applications.

The industrial users in both efforts concurred from the start to the idea of separating out the treatment of extra-functional concerns from the rest of development issues. What was needed to that end was a systematic way to guarantee that the resulting products would meet the requirements by construction as opposed to by correction. The proposed solution centered on the formulation and the adoption of a reference software architecture based on four distinct yet closely related constituents: a component model, a computational model, a programming model and a execution platform. Specific instances were then proposed for each of those architectural constituents.

Separation of concerns was enacted at two levels of development.

At the design level, by segregating distinct concerns to separate design views, where design views are a manifest asset of the adopted component model. Each design view enforces specific creation, visualization and modification rights on design entities. Those rights are attributed according to the concern that the entity in question is an expression of. An entity relevant to a concern will allow creation and modification rights on itself in the design view attached to that concern: entities not relevant of that concern will instead be just read-only or even possibly plainly invisible in that particular design view. Extra-functional properties are defined as annotations attached to the outside of the component specification or to its exposed interfaces. The component boundary thus becomes the border between functional and extra-functional concerns.

At the implementation level, the realization of extra-functional concerns is delegated to two other software entities that pertain to the real-time architecture of the component framework: the container (which addresses tasking, synchronization and time-related concerns among others); and the connector (which addresses communication and interaction concerns). In the proposed approach, containers and connectors are automatically generated via model transformations, in a manner that conforms by construction to the programming model that implements the computational model chosen in the reference software architecture, and that preserves the desired execution semantics when on the target platform.

The industrial users involved in the research effort adopted a single component model and used all of its domain-neutral core features in their respective case studies. They were able to address a selection of their domain-specific needs via the addition of domain-specific views to the component model. This capability provided convincing evidence of the successful enactment of separation of concerns and of its ability to produce fully composable parts, all individually conforming to common design principles, and all consistently and verifiably contributing to end-to-end performance.

In the proposed approach the handling of extra-functional concerns stays strictly under the control by the design environment and it is performed via automated model transformation. This provision ensures the consistency of the approach and in particular – because of the congruent adoption of a computational model, a programming model and a conforming execution platform – warrants consistency between the software system as analyzed and the system implementation at run time.

The authors are now working on an extension of the component model that supports hierarchical components, with particular attention to ensuring that the specification and realization of extra-functional properties can work with top-down decomposition as well as bottom-up composition. Future plans also include further enhancement to the domain-specific extension capabilities of the component model.

Acknowledgments

This work was supported by the Networking/Partnering Initiative of the European Space Agency and by the CHESS project under ARTEMIS JU grant No. 216682.

References

[1] I. Crnkovic, M. Larsson, O. Preiss, Concerning predictability in dependable component-based systems: classification of quality attributes, in: Architecting Dependable Systems III, Vol. 3549 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 2004, pp. 257–278.

[2] E.W. Dijkstra, On the role of scientific thought, in: E.W. Dijkstra (Ed.), Selected Writings on Computing: A Personal Perspective, Springer-Verlag, New York Inc, 1982, ISBN 0-387-90652-5, pp. 60–66.

[3] P. Kruchten, The Rational Unified Process: An Introduction, third ed., Addison-Wesley Longman Publishing Co. Inc., 2003, ISBN 0-321-19770-4.

[4] M. Panunzio, T. Vardanega, A component model for on-board software applications, in: Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications, IEEE Computer Society, 2010, pp. 57–64.

[5] M. Panunzio, Definition, Realization and Evaluation of Software Reference Architecture for Use in Space Application, Ph.D. Thesis, University of Bologna, Italy, 2011, <http://www.informatica.unibo.it/it/ricerca/technical-report/2011/UBLCS-%2011-07>

[6] T. Vardanega, Property preservation and composition with guarantees: from ASSERT to CHESS, in: Proc. of the 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, IEEE Computer Society, 2009, pp. 125–132.

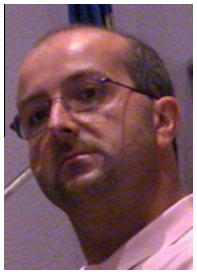
[7] Software Engineering Institute (Ed.), Defining Software Architecture – Modern, Classic, and Bibliographic Definitions, SEI – Carnegie Mellon <http://www.sei.cmu.edu/architecture/start/definitions.cfm>

[8] ISO/IEC, Systems and software engineering – Architecture description, ISO/IEC 42010:2011 (2011).

[9] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed., Addison-Wesley Professional, Boston, 2002, ISBN 0-201-74572-0.

[10] M. Chaudron, I. Crnkovic, Component-based software engineering, in: H. van Vliet (Ed.), Software Engineering: Principles and Practice, Wiley, 2008, ISBN 0-470-03146-8 (chapter 18).
[11] J. Sifakis, A framework for component-based construction extended abstract, in: Proceedings of the 3rd IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society, 2005, ISBN 0-7695-2435-4, pp. 293–300.
[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the 11th European Conference on Object-Oriented Programming, Vol. 1241 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 1997, ISBN 978-3-540-63089-0, pp. 220–242.
[13] M. Panunzio, T. Vardanega, On component-based development and high-integrity real-time systems, in: Proceedings of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, 2009, ISBN 978-0-7695-3787-0, pp. 79–84.
[14] T. Vardanega, Development of on-board embedded real-time systems: an engineering approach, Technical Report ESA STR-260, European Space Agency, ISBN 90-9092-334-2, 1999.
[15] M. Bordin, T. Vardanega, Correctness by construction for high-integrity real-time systems: a metamodel-driven approach, in: Proceedings of the 12th International Conference on Reliable Software Technologies – Ada-Europe, Vol. 4498 of Lecture Notes in Computer Science, Springer, Berlin Heidelberg, 2007, ISBN 978-3-540-73229-7, pp. 114–127.
[16] D.C. Schmidt, Model-driven engineering, IEEE Comput. 39 (2) (2006) 25–31.
[17] K. Balasubramanian, J. Balasubramanian, J. Parsons, A.S. Gokhale, D.C. Schmidt, A platform-independent component modeling language for distributed real-time and embedded systems, in: Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, 2005, ISBN 0-7695-2302-1, pp. 190–199.
[18] K. Balasubramanian, D.C. Schmidt, Physical assembly mapper: a model-driven optimization tool for QoS-enabled component middleware, in: Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, 2008, ISBN 978-0-7695-3146-5, pp. 123–134.
[19] R. van Ommering, Building product populations with software components, in: Proceedings of the 24th International Conference on Software Engineering, ACM Press, 2002, ISBN 1-58113-472-X, pp. 255–265.
[20] H. Kopetz, N. Suri, Compositional design of RT systems: a conceptual basis for specification of linking interfaces, in: Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society, 2003, ISBN 0-7695-1928-8, pp. 51–60.
[21] Aeronautical Radio, Incorporated, ARINC Specification 653–1: Avionics Application Software Standard Interface (2003).
[22] G. Douglass, Locke, software architecture for hard real-time applications: cyclic executives vs. fixed priority executives, Real-Time Syst. 4 (1) (1992) 37–53.
[23] N.R. Mehta, N. Medvidovic, S. Phadke, Towards a taxonomy of software connectors, in: Proceedings of the 22nd International Conference on Software Engineering, ACM, 2000, ISBN 1-58113-206-9, pp. 178–187.
[24] B. Spitznagel, D. Garlan, A compositional approach for constructing connectors, in: Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society, 2001, ISBN 0-7695-1360-3, pp. 148–157.
[25] A. Burns, B. Dobbing, T. Vardanega, Guide for the use of the Ada Ravenscar profile in high integrity systems, Technical Report YCS-2003-348, University of York, 2003.
[26] J.C. Palencia, M. González, Harbour, schedulability analysis for tasks with static and dynamic offsets, in: Proceedings of the 19th IEEE Real-Time Systems Symposium, IEEE Computer Society, 1998, ISBN 0-7803-5243-2, pp. 26–37.
[27] M. Panunzio, T. Vardanega, Ada Ravenscar code archetypes for component-oriented development, in: Proceedings of the 17th International Conference on Reliable Software Technologies – Ada-Europe, Vol. 7308 of Lecture Notes in Computer Science, Springer, 2012, ISBN 978-3-642-30597-9, pp. 1–17.
[28] J.B. Goodenough, L. Sha, The priority ceiling protocol: a method for minimizing the blocking of high priority Ada tasks, in: Proceedings of the 2nd International Workshop on Real-time Ada Issues, ACM, 1988, ISBN 0-89791-295-0, pp. 20–31.
[29] M. Bordin, M. Panunzio, T. Vardanega, Fitting schedulability analysis theory into model-driven engineering, in: Proceedings of the 20th Euromicro Conference on Real-Time Systems, IEEE Computer Society, 2008, ISBN 978-0-7695-3298-1, pp. 135–144.
[30] Object Management Group, SysML specification – version 1.3, <http://www.omg.org/spec/SysML/1.3/(2012)>
[31] L. Montecchi, P. Lollini, A. Bondavalli, Dependability concerns in model-driven engineering, in: Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, IEEE Computer Society, 2011, ISBN 978-0-7695-4377-2, pp. 254–263.
[32] L. Montecchi, P. Lollini, A. Bondavalli, Towards a MDE transformation workflow for dependability analysis, in: Proceedings of 16th IEEE International Conference on the Engineering of Complex Computer Systems, IEEE Computer Society, 2011, ISBN 978-0-7695-4381-9, pp. 157–166.
[33] F. Ciccozzi, A. Cicchetti, M. Krekola, M. Sjödin, Generation of correct-by-construction code from design models for embedded systems, in: Proceedings of the 6th IEEE International Symposium on Industrial Embedded Systems, IEEE, 2011, ISBN 978-1-61284-818-1, pp. 63–66.
[34] European Cooperation for Space Standardization (ECSS), Space Engineering – Ground systems and operations – Telemetry and telecommand packet utilization, ECSS-E-70-41A, 2003.
[35] E. Mezzetti, M. Panunzio, T. Vardanega, Preservation of timing properties with the Ada Ravenscar profile, in: Proceedings of the 15th International Conference on Reliable Software Technologies – Ada-Europe, Vol. 6106 of Lecture Notes in Computer Science, 2010, pp. 153–166, ISBN 978-3-642-13549-1.
[36] T. Vardanega, G. Caspersen, Engineering software reuse for on-board embedded real-time systems, Softw. – Pract. Exp. 32 (3) (2002) 233–264. ISSN 0038-064.

Marco Panunzio received the Laurea Specialistica (MSc) in Computer Science (full marks cum laude) from the University of Padova, Italy in 2006. He received the Ph.D. in Computer Science from the University of Bologna, Italy in 2011. During the Ph.D. and later as a post-doc research fellow at the University of Padova, Italy, he has been a visiting researcher at the European Space Research and Technology Centre (ESTEC) of the European Space Agency (ESA) in the scope of the Networking/ Partnering Initiative (NPI). Since May 2012, he joined Thales Alenia Space – France, where he works as R&D engineer in the area of on-board software development. His main research interests are: schedulability analysis of real-time systems, Model-Driven Engineering, Component-Based Software Engineering and software reference architectures.



Tullio Vardanega graduated with a degree in computer science at the University of Pisa, Italy, in 1986 and received the Ph.D. degree in computer science from the Technical University of Delft, The Netherlands, in 1998, while working at the European Space Research and Technology Centre (ESTEC) of the European Space Agency (ESA). At ESTEC, over the period 1991–2001, he held responsibilities for research and technology transfer projects as a lead person in the area of onboard embedded real-time software. In January 2002, he was appointed Lecturer in Computer Science, Faculty of Science, University of Padova, Italy, before becoming Associate Professor in October 2004. At Padova, he took on teaching and research responsibilities in the areas of high-integrity real-time systems, quality-of-service under real-time constraints and software engineering methods, including model-driven engineering, and processes for such environments. He has authored numerous papers and technical reports on these subjects. He runs a range of research projects in these areas on funding from international and national organizations.