Pitfalls and misconceptions in component-oriented approaches for real-time embedded systems: lessons learned and solutions

Marco Panunzio and Tullio Vardanega
University of Padova, Department of Pure and Applied Mathematics
via Trieste 63, 35121 Padova, Italy
{*panunzio, tullio.vardanega*}*@math.unipd.it*

***Abstract*—When the task befalls you to devise a novel component-oriented approach for real-time embedded systems you first dutifully undertake a thorough scrutiny of the import and pitfalls of previous attempts. This is what we have done for the last couple of years, in the context of an initiative promoted by the European Space Agency for the creation of a component model to serve the development of new-generation on-board software. In this paper we recapitulate the lessons we learned in that effort.**

I. Introduction

The development of real-time embedded systems has recently shown an evident surge of attention for approaches centered on Component-Based Software Engineering (CBSE) [1] and Model-Driven Engineering (MDE) [2].

The CBSE paradigm has potential for fostering better software design and for improving the software fitness for reuse. MDE makes it possible to raise the abstraction level of the development process and use automation capabilities to generate lower-level artifacts, such as analysis models, documentation and (part of) the implementation code. The marriage of the two arguably sounds like a jolly good news. Their joint application should in fact earn the development of real-time embedded systems the increased productivity and shorter time to market that are the promised land of software industry.

Unfortunately, such a brave undertaking is far from obvious in our target domain, and its relative successes are still too few to motivate massive adoption.

In the last few years, we were involved in an investigation promoted by the European Space Agency (ESA) aimed to the conception and realization of a software reference architecture for use in future on-board software. In previous work, cf. e.g., [3], we elaborated on the key ingredients of the approach we are developing in that context. One of those central ingredients, and one of special interest to this paper, is the *component model*, which permits to create the software design as an assembly of software components.

The essential elements of the above approach are also being investigated and assessed in the CHESS project (cf. the acknowledgments), an international R&D project targeting space, telecommunication, railway systems and investigating the applicability of its results also to the automotive domain.

Our investigation for the component model [4] largely benefited from the feedback obtained from those two distinct contexts of application. The richness of horizon also permitted an extensive scrutiny of previous initiatives with possibly similar characteristics and ambitions. In that evaluation we were able to discern some, if not all, of the most successful traits and the most critical pitfalls of previous approaches. In this paper we elaborate on these findings and illustrate where our approach differs in positive from its predecessors.

Two concepts are central to component-oriented development: the *component*; and, separately, the *component model*.

There are various interpretations of these concepts; cf. e.g. [1] or [5]. The acceptation given to those concepts, and their realization in practice, largely influence the goodness of the approach, the fulfillment of all the application needs of interest, and ultimately, the opportunities for software reuse.

The distinguishing feature of our approach consists in permeating the whole development with the principle of separation of concerns [6]. We are in fact convinced that components shall encompass *exclusively* functional concerns and be void of any extra-functional concern.

The component is a unit of encapsulation, which exposes in an interface a set of cohesive functional services offered to the system and declares the functional services it requires (from other components or the environment) to operate correctly. The component explicitly *declares* any extra-functional need or requirements (e.g periodic execution with a certain period, needed resources) in terms of annotations to its functional interface. However, the source code of the component comprises exclusively sequential code; the extra-functional annotations on the component specification are used by the design environment to automatically generate software entities that *realize* those extra-functional needs. The nature of components enables their reuse under different extra-functional requirements, hence it increases their reuse potential.

The component model is the framework that defines: the form and attributes of components; the rules for their composition and packaging; the specification of extra-functional annotations and the deterministic rules for the automated generation of the implementation entities that realize the declared extra-functional annotations; the architectural description, which we argue should be made in terms of design views, each being a partial representation of the system focused on limited concerns and tailored for use by a specialized development actor.

The remainder of the paper is organized as follow: in section II we enumerate the most important lessons we learned on the definition of a component-oriented approach targeting real-time embedded systems; in section III we recapitulate the key elements of our component model by tracing our design choices to the lesson learned; finally in section IV we draw some conclusions.

II. Lessons learned

A. On the overall design process

1) The component model defines a design flow: The notional purpose of a component model is to permit the creation of a whole software system as a composition of software components. In actual fact, however, the component model does more than that. Not only does it prescribe the syntactic rules to create design entities and the way to relate them to one another, but – whether intentionally or implicitly, in any case inevitably – it also establishes a defined design flow.

The design flow comprises a series of steps that must be followed to create components, assemble them and ultimately produce the software system. It may also determine a set of precedence relations between those steps.

For example, a component model may require that components are enclosed in platform-dependent wrappers in order to execute them on a specific target platform. In that case it follows that the creation of at least some basic representation of the hardware topology and allocation of components to target processing units are preconditions to code generation.

This implies that the developers of the component model shall pay careful attention to ensuring that the design flow promoted by their approach is compatible with the development process in use with the concerned industrial domain.

2) Do not force a development direction: Software design proceeds in one of three possible directions: top-down; bottom-up; some combination of them. A general evaluation of the virtues and limitations of those directions is not very useful, as the goodness of one's choice is highly dependent on the specific context of application. We can however reason on when those directions are more likely to be chosen: top-down development is likely to occur when components are specified for the first time (as top-down reasoning helps to master the design complexity); in contrast, bottom-up development is likely to prevail when reuse of components enters the picture.

Similarly to the design flow, a component model may well promote a given development direction, perhaps even actively. Problems arise however when active promotion of one direction causes active opposition to another.

An example may help illustrate the point: some research in connectors theory (e.g., "exogenous composition" for "encapsulated components" [7]) very strongly promotes bottom-up development. That can be acceptable for some application domains or some specific development cases, but it can cause serious inconveniences if applied to other domains where the development practice has historically and conveniently settled to top down.

B. On component definition

1) Lack of abstraction: One of the most interesting and qualifying aspects of a component model is the level of abstraction of its design entities. It is important to understand that the user space of the component model is above all the realm of the *software architect*. The primary goal of the software architect is to direct the software design in accord with the methodological principles of the chosen software architecture. Hence the component model shall provide a view of the software (and partly of the system) at a level of abstraction that corresponds with that goal. This implies that the level of *specification* required to the component model is most definitely not the *implementation level*: the component model shall not pollute the design view with implementation artifacts like threads, semaphores, and such like.

Detailed design, low-level details and implementation concerns (like allocation to threads, selection of task priorities, etc.) are by all means outside of the *primary concerns* of the component model. That does not mean of course that the component model shall simply ignore those aspects. We contend instead that the implicit design flow of the component model shall ensure that all these concerns appear later in the development, when they become the right subject of attention. Furthermore those concerns should be addressed through the component model and maintained in a syntactic structure that is either directly amenable to static analysis or otherwise easily transformable in the input formalism for the chosen analysis tool.

Finally, the implementation aspects addressed through the component model shall be faithfully (i.e., in a semantically consistent manner) reflected in the implementation: this is so much easier if the design environment caters for them by way of automated code generation.

2) Stateless components: Some component models allow the creation of exclusively stateless components (for example, the TASTE toolchain [8] developed at ESA).

That component model in fact does not permit to specify at design level the attributes (i.e., typed parameters) of the component, which collectively form the *state* of the component. The state of the component is thus relegated to the algorithmic code of the component implementation. In that manner however the component model is unable to represent that information in the design specification.

This is a very constraining choice, as it avails no means to: (i) control component configuration directly in the design space; (ii) understand which provided services of the component access which part of the state (i.e., the typed parameters, through getter and setter operations), so as to identify which subsets of the state should be protected under mutual exclusion, if any.

The difference between managing issue (ii) from within the design space and doing so only at implementation level is crucial. In the former case it is in fact possible to generate from the design specification an analysis/implementation view which correctly describes all the synchronization mechanisms that need to be employed (semaphores, protected objects, etc.). Hence a valid input for schedulability analysis can be derived from it. In the latter case, instead, the component can only be viewed as a single monolithic state. This pushes the use of mutual exclusion on the complete execution of all the services provided by the component. The net consequence of which is poor responsiveness at system level owing to the excessive blocking time induced by the blind recourse of mutual exclusion protocols.

3) Software interfaces vs. typed ports: Different approaches exist to expose the provided and required services of a component. The two predominant solutions are (software) interfaces and typed ports. In the former case the component provides (respectively requires) one or more interfaces. Each interface gathers a set of functionally cohesive operations that can be used to access the component. An interface can be specified using the syntax of a programming language, or (as in the majority of cases) with a dedicated Interface Definition Language (IDL). Examples of the IDL solution can be found in Koala [9], Robocop [10] or LightweightCCM (a conformance point of the CORBA Component Model) [11]. In essence, the component model provides a concrete realization of all the provided interfaces. In the alternate approach, a component exposes just a set of typed ports, as done in, e.g., ProCom [12], BIP [13] and TASTE [8].

Although syntactic means to group ports in some composite entity may exist, the radical difference between the two approaches stands: (software) interfaces exist independently of the component; more precisely, their creation precedes the specification of components. Ports do not.

The approach with software interfaces possibly incurs more design complexity, but it owes its greater advantages to its better fit for object-orientation. In fact, software interfaces can be extended (using some interface refinement mechanism) to create new interfaces that are subtypes of a base interface. This is very useful to manage the evolution of a system by reducing the effort to substitute or update components (a required interface can be bound to a provided interface of another component on the condition that it is a valid subtype) and to update the component bindings both in the design specification and in the implementation.

C. On the design language

1) The design language is not the component model: A fundamental distinction must be drawn between what is the *component model* and what is the *design language* that is used to create components.

The design of the component model (component features like ports, attributes, component bindings, deployment and extra-functional concerns, etc.) precedes and is disjoint from the actual language that is later used to specify the components in the user space.

When the time comes to define the design language for the component, there always is at least one default solution: to create a domain-specific language that exactly mirrors the definition of the component model.

Alternatively, the developer of the component model may try to express it using a standard modeling language for real-time embedded systems. Example languages include the UML MARTE profile [14] or AADL [15].

Choosing a standard language as the specification language of the component model can earn strategic advantages: the standard status of the language can encourage tool vendors to invest in the creation of design environments for it. In that case, the component model would leverage the tooling, thereby dispensing with the burden of developing an ad-hoc design environment from scratch. Furthermore, it is quite likely that the developers of tools for specialized analysis (e.g., schedulability analysis, fault-tree analysis, etc.) may converge over time to a common reference language that permits them to feed their tools with the information extracted directly from standard representations of the system model at hand. In this case, the component model developers are no longer required to create syntactic and semantic mappings to the input formalisms of each of the needed or desired analysis tools.

Of course, the component model developers must be able to fully express all the features and traits of their component model with the standard language, whether directly or using its extension mechanisms, as for example with UML stereotypes or AADL property sets.

When trying to reconcile with the pre-existing structure of the modeling language, two situations may occur. In one, the chosen language is able to fully express the component model, because its expressive power covers a superset of the needed features. In the other instead, the chosen language is not expressive enough and thus the mapping fails. The situation is depicted in figure 1.

In any case, the designer of the component model should tailor the chosen design language so that it only expresses what is necessary for the component model and avoids the confusion and overhead stemming from the expressive power in excess.

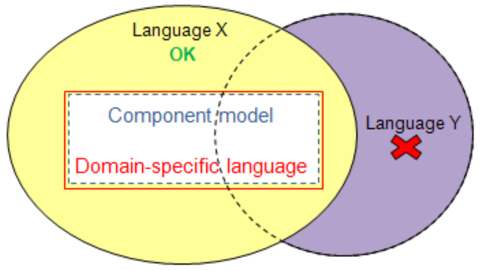

Figure 1. A component model has always a corresponding domain-specific language able to fully express it. Language X can express a superset of the necessary concepts, and it is able to fully express the component model. That is not true, for language Y, which lacks the expressive power for all the concepts of interest.

D. On extra-functional concerns and analysis

1) First design a component model and only later think about extra-functional concerns and analysis: The component model is not completely independent of extra-functional and analysis concerns [3].

Designing a component-oriented approach without considering those aspects from the outset can lead to several inconsistencies, infeasible implementation and costly corrections to the approach, as patently shown by the development of the AUTOSAR initiative [16].

The key argument here is that a component model for real-time systems makes sense and serves its purpose solely if it always leads to an analyzable implementation. It therefore is the body of the chosen analysis theory that dictates the spectrum of the admissible implementations.

Two implications stem from this observation. Firstly, when some theories assume questionable simplifications to ease the mathematical solution of the problem, they narrow the admissible form of the implementation to the extent that it may be considered poor and uninteresting. An example of those simplifications is the infamous assumption of task independence, which defeats even the simplest producer-consumer collaboration pattern.

Secondly, the analysis space coincides with the level of the implementation. For this reason, the component model shall ensure that analysis concerns are accurately reflected at the higher level of component abstraction. It shall then be possible to syntactically express (or derive) at component level all the information to create input for the analysis.

2) Plain schedulability analysis is not enough: Real-time software is subject to verification to ascertain that it fulfills all the timing requirements. To fulfill that obligation, some form of schedulability analysis is performed on some description of the software implementation.

In recent years, a number of independent projects demonstrated that it is feasible to generate the input for schedulability analysis from a component description. Notable examples include: SaveCCM [17], which is able to translate the design specification into an equivalent representation expressed

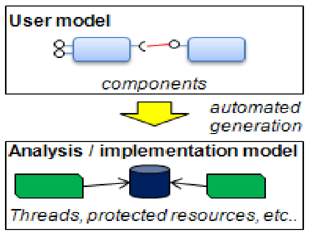
User model
components
automated generation
Analysis / implementation model
Threads, protected resources, etc..

Figure 2. The user model and the analysis model correspond to different abstraction levels. The approach shall be able to generate a meaningful analysis model from the information specified at the user level.

as timed automata with tasks, and perform response time analysis on it; AdaCCM [18], which permits to decorate the component description with metadata that are used to create the input for the MAST analysis tool [19], which implements analysis equations of the response time analysis strand, up to offset-based analysis with precedence constraints [20].

This is indeed an interesting achievement, which demonstrates that it is possible to feed schedulability analysis with information specified or derived from the abstraction level of the user space (where component are described), as opposed to the abstraction level of the analysis (which applies to implementation entities like tasks and the like) [21].

If the overall approach warrants that the architectural and semantic assumptions of the analysis are conveyed to and preserved in the implementation, and the execution platform guarantees the preservation of the resulting properties, then the design specification can factually be considered as a faithful representation of the software at run time.

To close the loop, it is however necessary to back propagate to the user space (i.e., as attributes that decorate the design specification) the results of the analysis, in a simplified form of model-based round-trip analysis. In this way, the designer can directly appreciate the results of the analysis as a complementary information of the component specification. Iterations of the analysis then become just a matter of changing attribute setting in components; which can be repeated until the designer is satisfied by the predictions on the timing behavior of the system under development.

Unfortunately, while this shows the feasibility of a value-added contribution to the goodness of the development process, it still does not earn us as much advantage with regard to the quality of the *results* of the analysis.

In point of fact, simple (or worse, simple-minded) support for schedulability analysis is not enough for real-scale systems. The problem used to be with the pessimism incurred by the *analysis equations*, but this is no longer so thanks to the constant advancement of the theory. The problem rather consists in the generation of the worst-case scenario, single or multiple that they may be.

The adopted analysis theory specifies the conditions that determine the theoretical worst-case scenario for the system and the rules to statically determine it, using the information given as input for the analysis.

However, as it is well known, the worst-case scenario may never occur during system execution, and so for two reasons. First because the system may not incur at the same moment all the adverse conditions that determine the worst-case scenario (e.g., all the tasks executing for their worst-case execution time); secondly, because perhaps the overall system cannot incur the theoretical worst-case scenario for some logical conditions (for example, two sporadic tasks that cannot be activated at the same time or as frequently as their minimum inter-arrival time, due to some logical condition that it was not possible to extract statically from the design). The former situation is not interesting, because the analyzed system can still reach the worst-case condition. The latter instead is caused by our inability to supply the analysis with sufficiently accurate information to understand that it has to prune the worst-case scenario from conditions that can be excluded a priori.

The situation is even worse when we want to develop a system with several multi-moded applications (each designed in the form of a component or an assembly of components): each application will behave differently with regard to functional needs, timing behavior and resource needs, in accord with the current mode of operation. In this case, with a plain application of schedulability analysis, the worst-case scenario for the analysis would be generated with the joint occurrence of the worst-case scenario of each application. It is quite plausible that the system cannot even forcedly reach that condition, which is just generating so much pessimism that the analysis may incorrectly report the system as infeasible.

To remedy this situation, we must strive to support scenario-based analysis. The designer thus shall: (i) have the means to relax the formulation of the worst-case scenario, whenever they know that the theoretical worst-case scenario of the system can never be reached; (ii) be able to command the calculation of the worst-case scenario as the composition of the right set of local analysis scenarios.

Prescription (i) is similar to what is commonly applied in static timing analysis: a tool may not be able to statically determine the maximum number of iterations of one loop, and the user manually annotates the correct loop bound, which is then used to perform the analysis. For prescription (ii), the designer (if interested in reducing the pessimism of the analysis) shall be able to associate a local analysis scenario to at least every operational mode of an application, and be guided in the composition of multiple local scenarios to create the real worst-case scenario for the system.

It is clear that both prescriptions shift the responsibility of formulating the worst-case scenario from the theory to the designer. In particular, with prescription (i) we may incur *unsafe* analysis scenarios. with prescription (ii) conversely, if the designer omits the specification of a local analysis scenario, the global worst-case scenario may be built off an *incomplete* set of local analysis scenarios. The generation of a global analysis scenario from a set of locally defined scenarios (even if unrelated to operational modes) is investigated for example in the Robocop component model [10].

In spite of those concerns however, scenario-based analysis may be a worth direction to pursue in those domains where predictability and efficient use of the available resources matter.

3) Separation between the specification of functional/ algorithmic concerns and extra-functional concerns: We maintain that a component model for real-time embedded systems ought to enforce strict separation of concerns between the functional/algorithmic concerns and all extra-functional concerns [4]. Concurrency and real-time concerns should be dealt with by component wrappers generated around the component. Interaction concerns are dealt with by *connectors*.

Separation of concerns shall be enforced: (i) in the design process through the use of "design views", in line with what is advocated by ISO 42010/IEEE 1471 [22]; (ii) at syntactic level in the specification language for the component; (iii) in the implementation, by careful allocation of the extra-functional concerns exclusively to the automatically generated code for the component wrappers and connectors.

Components that encompass only sequential code can be immediately reused under a variety of extra-functional requirements: it is sufficient to declare in the design environment the new extra-functional attributes, and new, appropriate component wrappers and connectors will be generated.

4) Make explicit all extra-functional constraints: Even if we are able to develop components that solely comprise pure sequential code, it may happen that the source code implicitly carries some extra-functional concerns in the form of constraints on its execution.

For example, the code of a control law may have been developed for execution with a defined frequency (e.g., 8Hz), because its designer qualified its robustness for that specific rate only. This kind of extra-functional constraints cannot be inferred from the source code.

It is then important to make these constraints explicit in the form of annotations on the component specification. The component model shall then offer: (i) the means to syntactically specify those constraints to augment description of the component implementation; (ii) means to check that when when extra-functional attributes for a component are specified, they do not violate the extra-functional constraints that apply to the component implementation.

E. On the applicability of the component model

1) Misunderstanding the concept of "domain-specific" component model: The creation of a component model generically targeting real-time embedded systems is not sufficient. That component model would in fact qualify as ”domain neutral”. ”Domain specific” in this context denotes a component model specialized for a given real-time embedded application domain (e.g. space, telecommunications, civil avionics, etc.). Support for ”domain-specific” concerns entails the support for all the concerns carried by that specific application domain. Without that support, the adoption of the component model as industrial baseline may become unattractive, as the extent of modification required to fully adapt the domain-neutral part to domain-specific needs may be significant, and may even risk to undermine fully or in part the guarantees of the original approach.

In essence, the ideal situation would be the development of a component model shared between different application domains, yet capable of expressing domain-specific aspects. The domain-neutral part of the approach would be used by all the domains. Each domain would then activate only the domain-specific part of interest (which includes language extensions, additional design views, specialized analysis and code generation) to complement the domain-neutral part.

The consequences of this vision are challenging at different levels: (i) from the methodological point of view, the solution to the domain-specific concerns shall not invalidate the core principles underlying the domain-neutral part; (ii) from the language point of view, the design language shall permit the modular specification of domain-specific concerns, loosely coupled with the domain-neutral part; (iii) from the tooling and design point of view, a domain-specific concern shall be presented in a separate design view, which is activated only when needed; (iv) from the analysis point of view, the narrowing of the target to a specific application domain makes it possible to specialize the analysis model with a fine-grained description of the relevant aspects of the target platform (for example, the precise delegation chain for remote message passing of the distribution middleware of choice) and to adopt refined analysis equations (for example, response time analysis specialized for the adoption of the Ravenscar Computational Model [23]).

III. A COMPONENT MODEL FOR ON-BOARD SOFTWARE APPLICATIONS

After discussing, with the benefit of the hindsight, the pitfalls of our predecessors, let us now outline the key aspects of the component model for on-board software that we are currently developing. In the description we will especially highlight the aspects that were influenced by the lessons learned discussed earlier in this paper.

The component model enforces separation of concerns by carefully allocating them to three distinct software entities: (i) the component; (ii) the container; (iii) the connector.

Components are software units that comprise pure sequential code, and are candidates for software reuse. Containers are software wrappers, automatically generated by the design environment, which realize the declared extra-functional attributes and warrant the preservation of the analysis results. Connectors are responsible at implementation level for the interactions between components. The designer creates exclusively components in the design space. Containers and connectors are generated by the design environment.

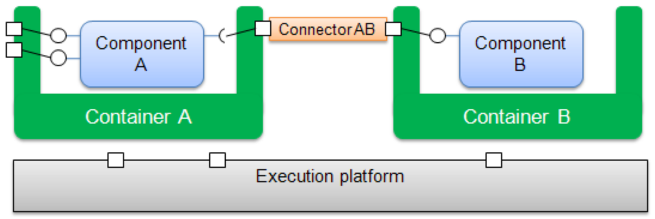

Figure 3. The figure depicts components, containers (each of which embeds a component), and connectors, which manage the mediated connection between containers. Containers and connectors rely on the services of the execution platform for the realization of their concerns.

In our approach, the designer creates component ”types”, which include the declaration of provided and required functional services typed with already defined software interfaces, as well as a set of typed attributes with getter and setter operations generated according to the visibility of the attribute (read-write, read-only, private). Component types are created in isolation.

From component ”types”, the designer creates component ”implementations” that are a concrete realization of a type in a programming language of choice. The component implementation is the subcontracting unit of the approach. A software integrator can establish on it a set of technical budgets (worst-case execution time of operations, memory footprint, etc.) and delegate its realization to a software supplier. Extra-functional constraints emerging from the source code are annotated on the implementation.

The designer continues by instantiating component ”instances”, which are the entities that are bound together to satisfy the functional needs, are subject to allocation to processing units and on which the designer declares the desired extra-functional attributes (e.g., ”cyclic execution” or ”sporadic execution” with the appropriate period/minimum inter-arrival time and deadline, synchronization requirements, end-to-end timing requirements, etc.).

The software model is then used to generate the appropriate input for the analysis of interest. The results of the analysis are directly reported back in the user space to confirm or refute the feasibility of the design.

Finally, containers and connectors are automatically generated to match the extra-functional attributes specified at instance level.

Figure 4 represents the design flow induced by our component model, depicted in terms of design steps and precedence relations. For example: our components expose interfaces to relate with other components: the creation of software interfaces to reference from components is a precedence constraint for the creation of component ”provided

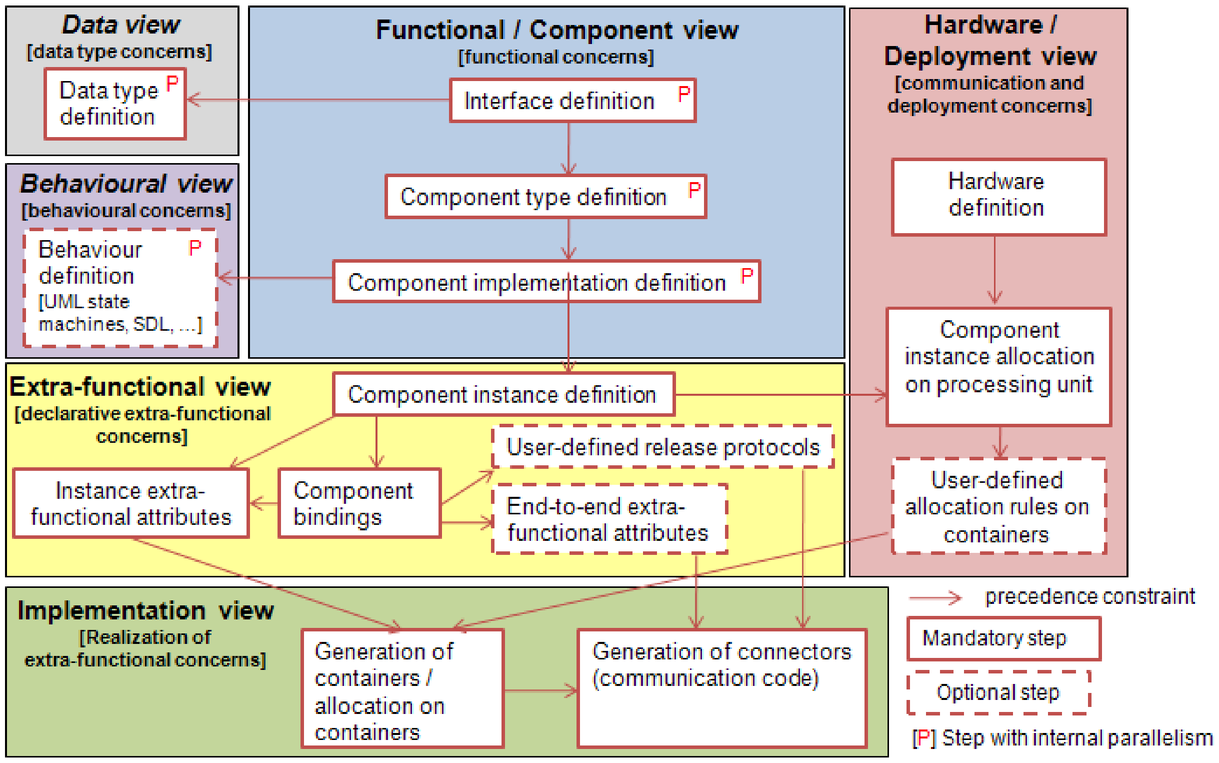
Data view
[data type concerns]
Data type definition P
Functional / Component view
[functional concerns]
Interface definition P
Component type definition P
Component implementation definition P
Hardware / Deployment view
[communication and deployment concerns]
Hardware definition
Behavioural view
[behavioural concerns]
Behaviour definition [UML state machines, SDL, ...] P
Extra-functional view
[declarative extra-functional concerns]
Component instance definition
Component instance allocation on processing unit
User-defined release protocols
Instance extra-functional attributes
Component bindings
End-to-end extra-functional attributes
User-defined allocation rules on containers
Implementation view
[Realization of extra-functional concerns]
Generation of containers / allocation on containers
Generation of connectors (communication code)
precedence constraint
Mandatory step
Optional step
[P] Step with internal parallelism

Figure 4. The design flow of our component model and the design views used to enforce separation of concerns.

interfaces". The figure also depicts the various design views with which we organize the specification of the system.

Activities of different views can proceed independently (they can be parallelized and assigned to different actors), provided that their precedence constraints are respected. For example, "Hardware definition" can proceed independently of any concern of the functional, data or behavioral view; the "Behavior definition" of a single component only when its component implementation has been defined (it is not needed to define *all* component implementations). Finally, several steps (identified with a 'P' in Fig. 4) can be internally parallelized as it is possible to define their design entities in isolation. For example, it is possible to define component types in isolation; or there is no dependency between the behavior definition of two distinct component implementations.

To confirm our claim about the separation between the component model and the specification language for components, one of the authors of the paper is creating a prototype implementation of the component model described in [4], using a domain specific language, defined as an *ecore* domain-specific metamodel (DSM). The domain-neutral part of the component model is being implemented instead in the CHESS project as an extension of the UML MARTE profile.

We are currently investigating the extension of the DSM to include domain-specific concerns relevant for the space domain, e.g., the support for PUS services [24], which specify the transmission and on-board execution of operation requests issued by the ground segment. Example of those services include: the monitoring of on-board parameters, specification of the reaction to events raised on board, the dump or upload of memory regions, the frequency of the generation and contents of the telemetry generated on board and the rules to downlink it to a ground receiver, etc.

In the CHESS project we will instead investigate how to include in the component model a selection of the domain-specific concerns of the target domains according to the criteria we discussed in section II-E.

In both projects we are investigating the application of scenario-based schedulability analysis.

Finally, the biggest challenge ahead of us is the inclusion in the approach of hierarchical components, consistently with the outlined goals of the two projects, reconciling them with the current structure of the component model and avoiding the pitfalls discussed in section II.

IV. Conclusions

A number of proposals have been made in recent years for the adoption of component-oriented approaches in the development of real-time embedded systems. We consider this to be good news, because that evolution is bound to raise the level of abstraction with which systems can be constructed while offering, in principle, sufficient control of the performance-critical trade-offs of the system design and implementation. The net result of that would be greater economy, faster development and more effective consolidation of best practices.

The success of most of those proposals however was doomed by a number of common pitfalls and misconceptions that undermine their goodness of fit.

We had the luxury of hindsight in capturing (some of) those defects, for the task we were commissioned entailed a thorough review of the state of the art and enabled us to interact with authors and users of relevant proposals.

In this paper we discussed the most common pitfalls and misconceptions that we encountered in our review, and we contrast with them an outline of the proposal that we are currently working on.

ACKNOWLEDGMENTS

This work was supported by the Networking/Partnering Initiative of ESA/ESTEC and by the CHESS project ("Composition with Guarantees for High-integrity Embedded Software Components Assembly", ARTEMIS Joint Undertaking grant nr. 216682).

REFERENCES

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

[2] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[3] M. Panunzio and T. Vardanega, "On Component-Based Development and High-Integrity Real-Time Systems," in *Proc. of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[4] ——, "A Component Model for On-board Software Applications," in *Proc. of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, 2010, pp. 57–64.

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

[6] E. Dijkstra, "On the role of scientific thought," in *Selected writings on Computing: A Personal Perspective*, E. W. Dijkstra, Ed. Springer-Verlag New York, Inc., 1982, pp. 60–66.

[7] K.-K. Lau and M. Ornaghi, "Control Encapsulation: A Calculus for Exogenous Composition of Software Components," in *Proc. of the 12th International Symposium on Component-Based Software Engineering*, 2009, pp. 121–139.

[8] E. Conquet, M. Perrotin, P. Dissaux, T. Tsiodras, and J. Hugues, "The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software," in *Proceedings of Embedded Real Time Software and Systems (ERTS)*, 2010.

[9] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *IEEE Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[10] E. Bondarev, P. H. N. de With, and M. Chaudron, "Predicting Real-Time Properties of Component-Based Applications," in *Proc. of the 10th Int. Conf. on Real-Time and Embedded Computing Systems and Applications*, 2004, pp. 60–78.

[11] Object Management Group, "CORBA Component Model, v4.0," April 2006, http://www.omg.org/technology/documents/formal/components.htm.

[12] T. Bures, J. Carlson, I. Crnkovic, S. Sentilles, and A. Vulgarakis, "Progress Component Model Reference Manual - version 0.5," Mälardalen University, Tech. Rep., April 2008. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=1467

[13] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," in *Proc. of the 4th IEEE Int. Conference on Software Engineering and Formal Methods*, 2006, pp. 3–12.

[14] Object Management Group, *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)*, 2009, version 1.0 http://www.omg.org/spec/MARTE/1.0/.

[15] SAE International, "Architecture Analysis and Design Language (AADL)," http://www.aadl.info.

[16] R. Racu, A. Hamann, R. Ernst, and K. Richter, "Automotive Software Integration," in *Proceedings of the 44th annual conference on Design Automation*, 2007.

[17] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The SAVE Approach to Component-based Development of Vehicular Systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 655–667, 2007.

[18] P. López Martínez, J. M. Drake, P. Pacheco, and J. L. Medina, "Ada-CCM: Component-based Technology for Distributed Real-Time Systems," in *Proc. of the 11th International Symposium on Component-Based Software Engineering*, 2008.

[19] Universidad de Cantabria, "MAST: Modeling and Analysis Suite and Tools," http://mast.unican.es.

[20] J. C. Palencia and M. González Harbour, "Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems," in *Proc. of the 20th IEEE Real-Time Systems Symposium*, 1999.

[21] 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.

[22] ISO/IEC/(IEEE), "Systems and Software engineering - Recomended practice for architectural description of software-intensive systems," ISO/IEC 42010 (IEEE 1471-2000), 2007.

[23] T. Vardanega, J. Zamorano, and J. A. de la Puente, "On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels," *Real-Time Systems*, vol. 29, pp. 59–89, 2005.

[24] European Cooperation for Space Standardization, "Space Engineering - Ground systems and operations - Telemetry and telecommand packet utilization," 2003, ECSS-E-70-41A.