




RMC Factory: A New Approach for Avionics Software Reuse

Laurent Dieudonné ¹, Andreas Bayha ², Benedikt Müller ³

Abstract: In contrast to many other industries, the reuse-rate of software in the avionics domain is very low. This situation originates mainly from the incomplete specifications for reuse in the avionics standards. In particular, these ones address almost exclusively the reuse of certified software *previously developed for a specific project* and consider only big monolithic executable components like complete applications or operating system modules. This paper describes a new approach enabling an efficient reuse of components from small up to big scale, *specifically developed for reuse* and conceived to fulfill the fundamentals DO-178C requirements while maximizing the reuse of certification artefacts. This presented approach combines methods, tool infrastructure and process extensions and we denote it as *Reusable Modular Component (RMC) Factory*. We also present an implementation of the tool infrastructure and discuss our lessons learned from this implementation and the first experiments.

Keywords: Avionics, component, reuse, RMC, software factory, certification, process, tool chain.

1 Introduction

In comparison with other industries, software reuse in the avionics remains still rather inefficient and reserved for very specific applications. This is probably due to the lack of certification guidance concerning the reuse of simple software components. The avionics standards addressing at most the reuse topic are summarized in 5.

On the one side, a safety critical software in the avionics does not consist only of source and/or object code, but encompasses all certification process artefacts specified by the avionics development standards like project plans, requirements, design, tests or traceability. Therefore, the most widespread “clone-and-own” strategy used in many

¹ Liebherr-Aerospace Lindenberg GmbH, Software Platforms, Pfaenderstrasse 50-52, Lindenberg, 88161, Germany, laurent.dieudonne@liebherr.com, <https://orcid.org/0000-0003-4699-6328>

² fortiss GmbH, Guerickestr. 25, Munich, 80805, Germany, bayha@fortiss.org, <https://orcid.org/0000-0002-1835-7943>

³ Liebherr-Aerospace Lindenberg GmbH, Software Platforms, Pfaenderstrasse 50-52, Lindenberg, 88161, Germany, benedikt.mueller@liebherr.com, <https://orcid.org/0000-0002-6293-498X>

domains is not efficient for the avionics: any change of the software intended to be reused will induce a cascade of artefact modifications and therefore the loss of the previously prepared certification artefacts, and the certification credits eventually gained from a certified project taken as basis for the reuse.

On the other side, the current avionics development standards address only the reuse of big-scale and monolithic components fulfilling a complete system function, specifically developed for and already certified within a previous project. Driven by the strong avionics development process objectives, an efficient reuse in this case can in practice only occur if the customer requirements and the target computer are identical or very similar to the ones of the base project. This is the case for the most known and documented avionics reuse case described in [DH09][DH10], where variants are derived from a base model of the NH90 helicopter.

While remaining fully compliant to the avionics standards, our approach avoids the classical “clone-and-own” strategy and the big-scale focus of the current avionics guidance in favor of a solution considering small independent software components specifically designed for reuse in a DO-178C [RT11a] context, and combinable into bigger ones without compromising their process artefacts. Moreover, these components do not belong to the first-using avionics project but are stored in a warehouse common to all projects. Additionally, our approach enables a collaborative and incremental development of certifications artefacts, enabling the breakdown of the development costs over the projects using the new component concurrently.

This paper is organized as follows: First, the concepts of the RMC factory approach are explained. Then, the current implementation of the concepts is presented, followed by our lessons learned. A summary of relevant publications is then given. The paper concludes with a summary and outlook.

2 Concepts

Our approach leads to paradigm changes in the software development by introducing several major concepts, with the aim of helping a transition to an efficient avionics software factory. In this section, we will introduce these major concepts.

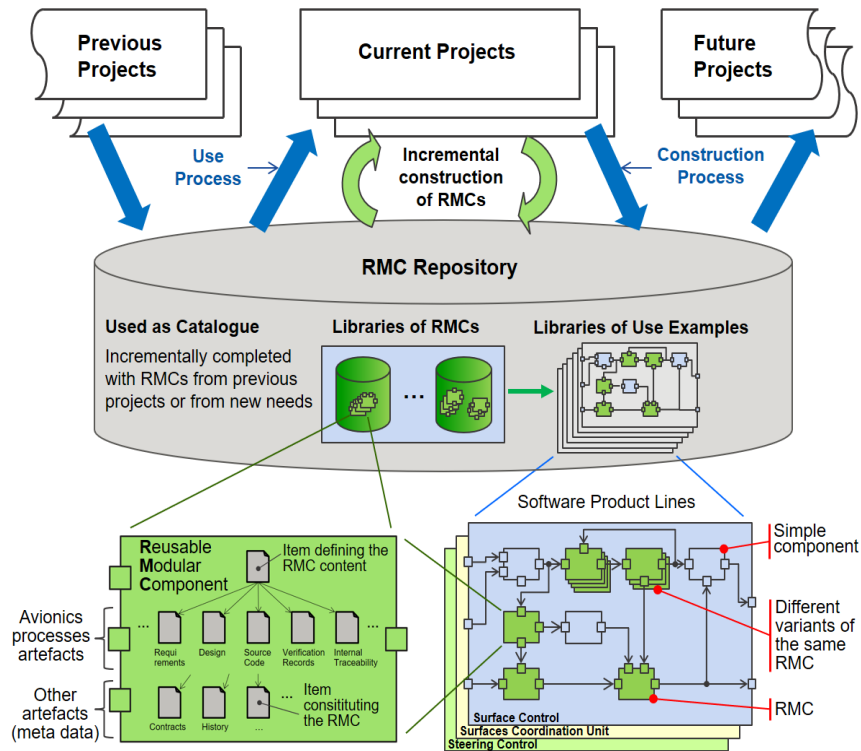


Fig. 1. Software factory concepts overview. Software components having mature process artefacts should be stored in a central warehouse to be immediately made available for other projects. Each project can use partly finished components realized by other projects and also act as contributor by completing the missing life-cycle data or by creating new components. Projects plans and used standards must be coordinated between contributing projects. Development processes at system and software levels should be extended with new activities guiding the construction and the reuse of components.

2.1 Catalogue of avionics-shaped software components

The first major concept is the design and centralized storage of components originally built for reuse in avionics projects. It is based on the compulsory use of a *central repository* storing *reusable pieces of software* accessible to all projects. These software pieces must be conceived as components fulfilling a standard definition, specified via a complex template – a metamodel, defining information placeholders, ontology and semantic –, and called *Reusable Modular Component* (RMC). These RMCs are organized in libraries of small components and presented like in a catalogue with usage examples in application skeletons. Each RMC is a model, instantiated from the RMC metamodel and filled with the information relevant for this particular RMC (name,

description, functions, integration aspects, etc.). The RMC metamodel has been realized in Eclipse/EMF.

The RMC metamodel is conceived to construct each RMC like a micro-project, meaning that it owns or references almost all categories of process artefacts as they are required in a full avionics application (requirements, design, source code, internal traceability, etc. but also plans and standards) for the certification activities – see Fig. 2. Additional information (meta data) is also included in the RMC in order to manage it. This contains many structured and typed information categories, like functional descriptions and features, interface specification, integration characteristics a.o. for compatibility checks, user manual, quality and maturity information including change history and open problem reports, platforms compatibility and resources consumption, etc. Each particular artefact that constitutes a RMC is identified as separate item and stored as such in a configuration management system which ensures versioning and problem reporting. One specific item – the “head” of the RMC – contains the list of all related items constituting the RMC, independently of whether they represent process artefacts or not. Moreover, again similarly to a complete avionics application, each RMC life-cycle data can be developed incrementally, so their current content depends on the needs of past and present consumer projects.

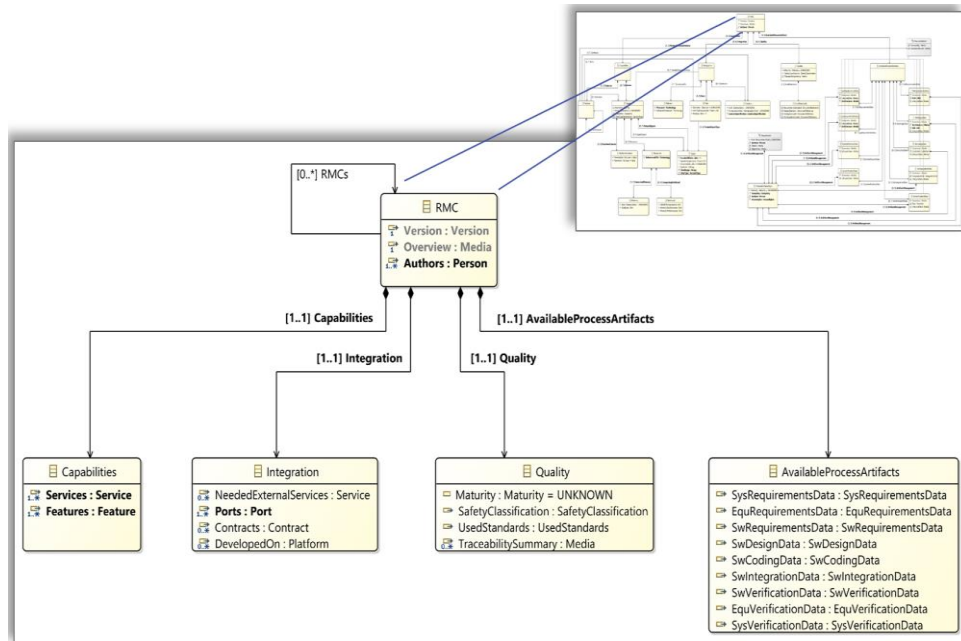


Fig. 2. RMC Metamodel Overview. A RMC is a *digital object* described by a formalized data sheet – a metamodel instance. An extract of the first level of the RMC metamodel is shown.

Last but not least, each project should also publish use examples of the RMCs it is consuming, in form of architecture skeletons identifying and referring exactly the used RMCs. In case several use examples (also from other projects) are very similar, they should be regrouped in a generic application architecture, containing different variants of RMCs or different combinations of them, and being reasonably configurable. The latter can be transformed in Software Product Lines (SPL) to enable high-level reuse of composite components, like described in section 2.3.

2.2 Development processes

One other major concept of our approach is the *integration of process activities specific for reuse* in the company development processes in order to stimulate the reuse of components *and* the construction of new ones during projects realizations (Fig. 1). On the one hand, each project team should systematically try to use existing RMCs to design its software. On the other hand, for new needs, project teams should design strategical software parts as RMCs to let other projects benefit from these assets in the future. This principle induces a major culture change: not only thinking about its own project, but also about future ones. Such a change should involve not only the software teams, but also the system engineering, the project management, the quality management and the company internal organization.

Process extensions needed for proper RMC use: Before starting to develop software, each project team has to search in the RMC catalog if one or a set of RMCs could fulfil some of the expected functional features. If yes, a more detailed validation, a *usage domain analysis*, concerning a.o. safety, timing, interface and resource requirements is done to confirm the adequacy of these RMCs with the project intending to consume them. If no, then a new software part will be developed from scratch. The choice of reusing RMCs is not without consequences. It may strongly influence the software design of the consuming project (not only the architecture) while at the same time the project requirements must be exactly fulfilled. At the planning level, the development standards applicable to the consuming project, as well as the own project plans (in particular the PSAC, the Verification Plan and the Configuration Management Plan) must be adapted to enable the reuse of existing software. At the realization level, all the own artefacts of a reused RMC shall stay unchanged in the consuming project. To guarantee this, in our solution the artefacts being integrated in a consuming project are copied (by the RMC-tooling) as *read-only* elements in the project file structure, inside a dedicated directory containing only the artefacts of this RMC. An integrity service of the RMC-tooling, based on checksums, ensures that the RMC artefacts are not changed by the consumer project. This integrity check must be performed during the verification process to prove that the reused RMC remained unchanged.

Process extensions needed for proper RMC construction: In our vision, the production of RMCs is not done by “the library guys”, but is the natural result of projects development and based on concrete needs. If a required project feature not available as RMC is evaluated as strategical (e.g. due to development costs, recurrent needs,

complexity, etc.), its corresponding new software piece must be developed as RMC. This new RMC is prepared incrementally, applying the project development schedule, plans and standards and simultaneously the development constraints for reuse. Thus, by using the RMC factory infrastructure and by integrating directly the RMC production activities in the project plans, RMCs are harvested from the projects being developed. A major success key to build a RMC for being reused often is the limitation of its scope to a single purpose understood as such in its intended usage domain: the RMC purpose description must be sufficient to let the targeted user understand *what* it does without to know *how* it is realized.

2.3 Variability as reuse lever

To promote the reuse of software parts, these must first be able to provide a simple solution to identified needs. One of the common issues for reuse is the adaptability of existing components to a new context. This is the main reason of the success of clone-and-own approaches in many other industries: the reused software is adapted for each new use case. Like explained earlier, this approach is too expensive in the avionics domain, hence we developed a better solution based on variability management.

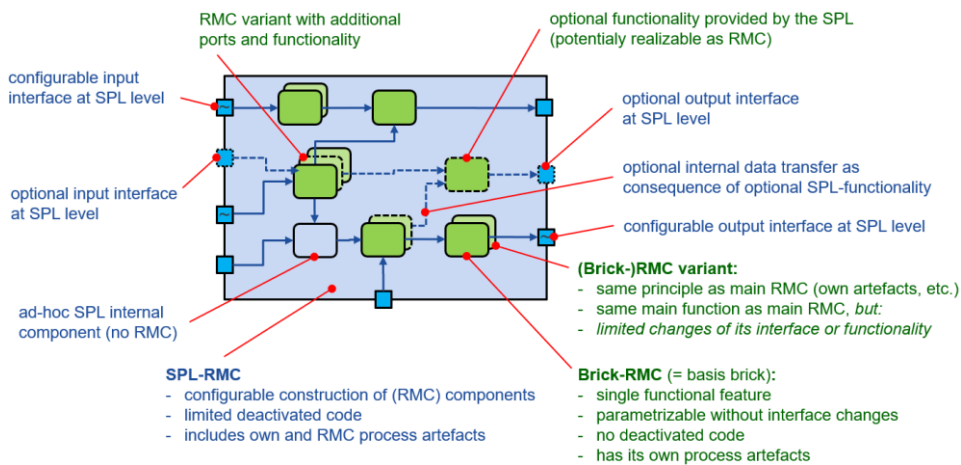


Fig. 3. Brick-RMC and SPL-RMC levels: dedicated RMC instances useful to maximize reuse with variability.

However, providing variability inside the reusable software to augment its adaptability may lead to problems inherent to the avionics domain. It can result either in runtime configuration and dynamic binding, or in embedding of alternative features within the application software. The first option is highly disliked because software control and data coupling must be completely verified, which it is very complex and time-expensive during runtime. The second option generates usually much deactivated code and other

unused process artefact parts, since usually only a small part of the alternative features is used per product. This is not forbidden, but makes the justifications for the certification much more complex.

To limit the deactivated parts while maximizing the adaptability of RMCs, our approach defines two categories of RMCs, providing two kinds of variability that preserve already realized certification-ready artefacts. This increases the attractiveness of RMCs reuse. Fig. 3 illustrates this principle. Both RMC categories are instances of the unique RMC metamodel described in 2.1.

Brick-RMC: defines a small RMC implementing basic functional features without internal variability. Its process artefacts are organized to remain unchanged for any integration case. However, our concept and tooling enable the management of *variants* of Brick-RMCs. A variant is also a full Brick-RMC, duplicating common properties from its basis Brick-RMC, and versioned specifically as derivative of the origin RMC. It provides differences principally concerning the interface without changing the main functional features. This method looks like “clone and own”, but the difference is that the process artefacts remaining unchanged keep their development maturity attributes and therefore do not need to be prepared again for certification. To maximize this effect, the RMC internal architecture should be designed by isolating parts prone to change from the origin in small software pieces. By this way, the Brick-RMC principle provides a limited but real variability at this level: case after case, variants are built and new consumer projects can find components better suited to their needs, avoiding heavy adaptation of their own architecture. The reuse of such RMC with fix defined features is thus easy and the time saved can be significant. Examples for Brick-RMCs from varying scopes but not configurable could be a signal monitor, a utility function like a CRC calculation, a voting algorithm, a statistic utilities set, a parameterizable filter, a specific control loop algorithm, etc.

SPL-RMC: The reuse of single Brick-RMCs is efficient, but the integration of many of them to build a full application may be complex and hold back project teams. Therefore our concept defines also Software Product Lines (SPL) of RMCs strongly inspired by [PBL05], providing high-level functionalities with variable rich features and interfaces, like software applications that need to be used in different contexts. The variability is implemented by differently combined Brick-RMCs and by interfaces that are organized in options. A SPL-RMC uses Brick-RMCs but may also use ad-hoc software not defined as RMCs. Only the Brick-RMCs and their concerned process artefacts statically selected for reuse during the SPL-RMC configuration are embedded in the project. Examples for SPL-RMCs from varying scopes, configurable at design and integration levels, could be an actuator control application, a scalable landing gear steering application, a RTOS kernel or library, a generic error logging module, etc.

2.4 Certification driven deployment of life-cycle data

Our reuse approach at *component* level is compatible with the current certification authorities publications [RT11a][RT11b][RT05] which regulate the reuse of *whole applications*. The separation of RMC life-cycle data at three levels (see Fig. 4) enables a high flexibility to integrate the RMCs in different projects. Yet, justification documents may be needed to demonstrate the equivalence of applicable plans and standards between the using projects and the RMC repository.

At lowest level, a Brick-RMC contains only basic life-cycle data that are known inside its scope like software requirements and design, source and object code, verification artefacts, traceability and problem reports. The second level is covered by the application software. As instance of a SPL-RMC or as simple aggregation of Brick-RMCs, this application software owns full certification liaison data and its own ad-hoc software artefacts, but only references the used Brick-RMCs and their life-cycle data. For the third level, the RMC general plans (software development, verification, configuration management, etc.) and standards (requirements, design and coding) are managed at library-level in the repository. An illustration is given in Fig. 4.

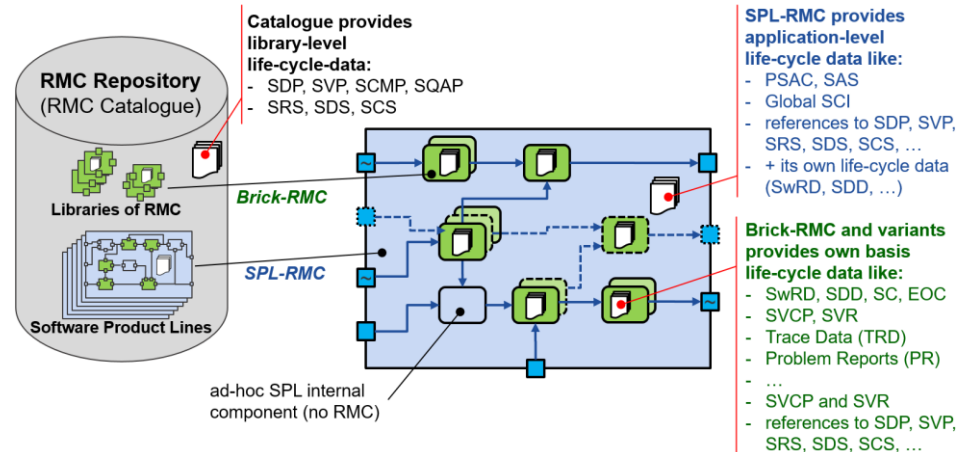


Fig. 4. To simplify certification, the life-cycle data (aka process data or process artefacts) is physically deployed at three levels: Brick-RMC (functional development artefacts), SPL-RMC (project integration and certification liaison ones), and Repository (common and generic ones).

Thanks to this approach, certification credits gained from previously certified projects using RMCs can be claimed by the new using projects if the RMC repository plans and standards are compatible with the ones of the new projects. This applies as well for the reuse of only a part of the RMC life-cycle data. The RMC repository concept and tooling enable the coexistence of different versions of plans and standards referenced by different RMCs.

2.5 Maximizing the reuse effect

Thanks to our concept and tooling that enables life-cycle data separation in the RMC repository, both the use and construction processes can be combined over several projects that are developed simultaneously. We named this a *Collaborative and Incremental Development*. That way, not only the development costs of the same RMC may be shared immediately between the concerned projects but it can also compensate personal bottlenecks in such tense situations with parallel projects. Fig. 5 shows a possible scenario up to the certification of a project including RMCs.

The development of RMCs generates more costs than one-shot, dedicated components. Even if the return on invest can already be notable from the first reuse case, companies must partly change their organization to integrate this reuse principle in an efficient way. In particular, as project teams must stay focused on the schedule-driven realization of their product features, a second team (“the library guys”) should undertake a part of additional efforts by driving reusable design of new RMCs and enforcing their efficient integration into projects.

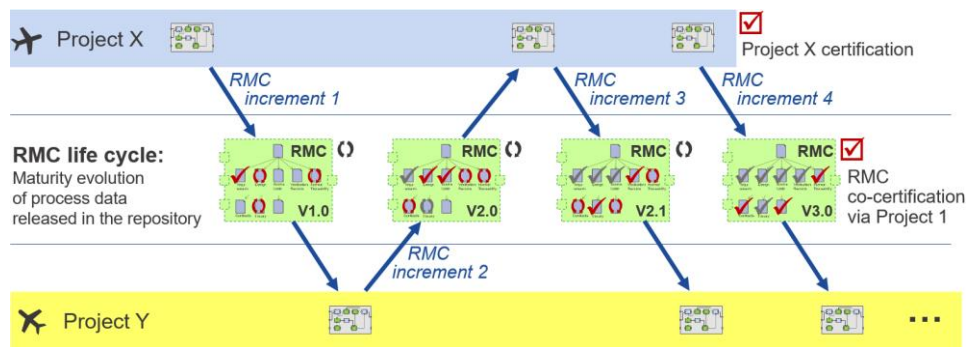


Fig. 5. RMC Collaborative and Incremental Development: RMCs having mature process artefacts but not necessarily complete are made available to other projects via the repository. Projects interested in RMCs which are still incomplete can contribute to their development by realizing the missing life-cycle data. Coordination between contributing projects about schedules, used standards and project plans is necessary.

2.6 Early validation of integration

As in all modular system architectures, all components specify certain assumptions on their system context, interfaces and similar. A standard example are components input signals, which expect certain physical measurement units, signal frequencies or value ranges. Whenever a RMC is reused, it is crucial to make sure that all these assumptions are fulfilled by the system it is inserted in. In addition, this system itself may have requirements toward the inserted components, which need to be fulfilled.

In order to capture these requirements for correct reuse, we enable every RMC to define

its own constraints for correct reuse in form of *contracts*. These contracts are different to the classical notion of contracts in formal analysis that specify behavior in form of pre and post conditions. Instead, we define invariants for the overall model structure – i.e. how a RMC is *correctly* integrated into a system. This includes the correctness of value ranges for interfaces and similar, but also certification information as required and assured DAL levels. Moreover, these contracts are not generated as object code, are not embedded on target and are not executed during runtime. They are checked during the integration of each RMC in each consuming project at design level. This solution is a great advantage for the certification and for the resource consumption of the target computer.

To enable automated checking of the contracts of all components, we introduced a dedicated *contract checker* service to our reuse environment. This contract checker can automatically check correct reuse, anytime a RMC is introduced into a system or the system around a RMC changes.

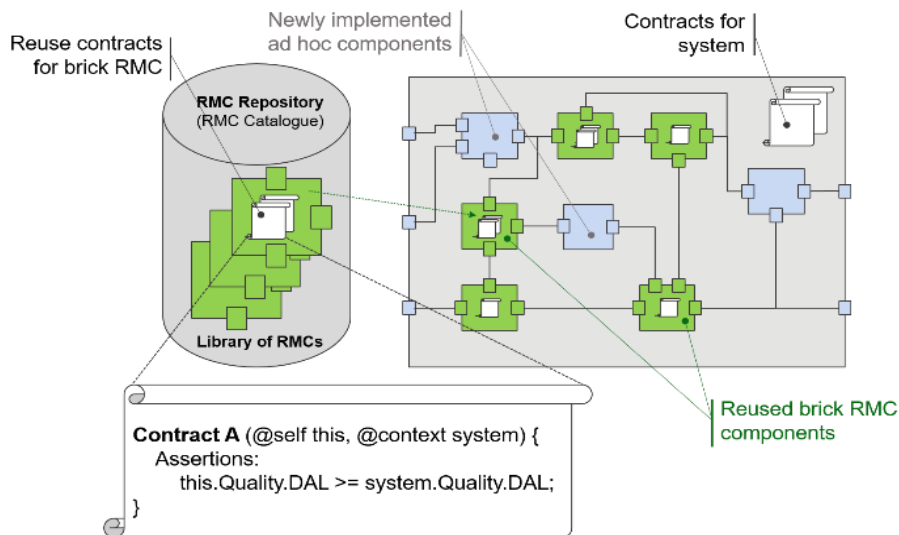


Fig. 6. Contracts specify how to correctly reuse a RMC component. These contracts need to specify the relation to future connected or contained components and the surrounding system.

A usual way in model-based systems engineering to implement such correctness checks is the usage of constraint languages and checkers. A prominent example hereby is the Object Constraint Language (OCL) [RG10] for which there are several implementations and of-the-shelf checkers available. Unfortunately, these existing approaches were not sufficient for our use case, as they are intended to deal with the correctness of individual models or artefacts. In our case, this would mean the internal structure of one single RMC component alone, as these are defined independent of an individual system context. Instead, a RMC is supposed to specify the prerequisites for correct reuse inside

the system around it – i.e. with other components it does not know about in advance. Fig. 6 illustrates an RMC with contracts that is reused in a system.

For this reason, we devised a constraint language which enables to specify contracts that also reason over components to which a RMC might be connected in future reuse scenarios. This is achieved by extending a OCL-like syntax for navigating between model elements and attributes defined in the RMC metamodel discussed in 2.1, and with the specific keywords *@self*, *@connected* or *@context* for referring to model elements of connected or surrounding components. With this, the contract checker can immediately check the assertions with these keywords, whenever this RMC is introduced into a system or connected to another component. Fig. 6 shows a simple example for such a contract.

3 Realization and integration in CASE-tools

To realize our concepts like described in chapter 2, a comprehensive and robust tool infrastructure has been implemented. We named it *RMC-Broker*. One of its major characteristics is the simplicity of integration of Computer Aided Software Engineering (CASE) tools. The RMC-Broker ensures a seamless workflow of RMCs between different CASE-tools. Its key characteristics hereby are: centralized storage; easy integration of tools taking part in the development process; multi-users and multi-tools; unified graphical user interface for all tools.

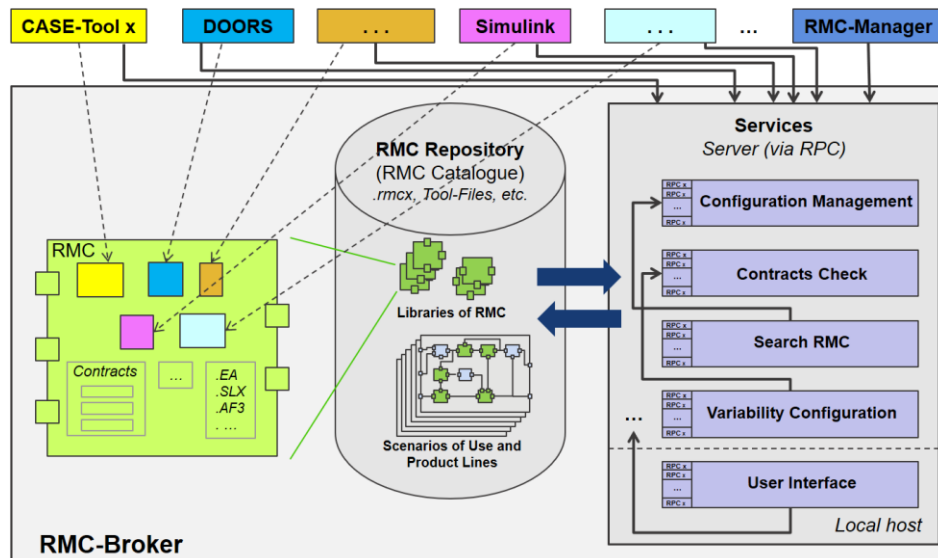


Fig. 7. The RMC-Broker facilitates the management of RMCs and simplifies the integration with CASE-tools thanks to common services.

Our approach based on the RMC-Broker establishes a modular tools platform, consisting of multiple, independent *services*, each providing a specific set of logically grouped functionality. Fig. 7 gives an overview about the RMC-Broker infrastructure. On the CASE-tool side, *connectors* provide an abstraction over the intricacies and details of communicating with these services. These connectors are themselves reused, allowing a rapid integration of new tools. Communication between tools / tool connectors and services is realized using a thin, well-defined remote procedure call interface. This connector principle is illustrated in Fig. 8. Two CASE-tools (Simulink and an UML tool) and the RMC-Manager have been integrated in the RMC-Broker for now (see 3.3).

3.1 Centralized repository

Instead of having tools directly exchanging with each other bilaterally, or reading and writing to some kind of database or repository, all required operations are centralized and coordinated by the Configuration Management Service (CMS). It controls accesses to RMCs and manages their variants and versions, as well as all their concerned process artefacts. The CMS is the sole module directly accessing the actual physical repository and therefore ensures data consistency for multi-user operations. It can also apply a set of checks and constraints before any repository operation. This service is absolutely mandatory for core operations of the RMC-Broker infrastructure.

3.2 CASE-tool integration / connection

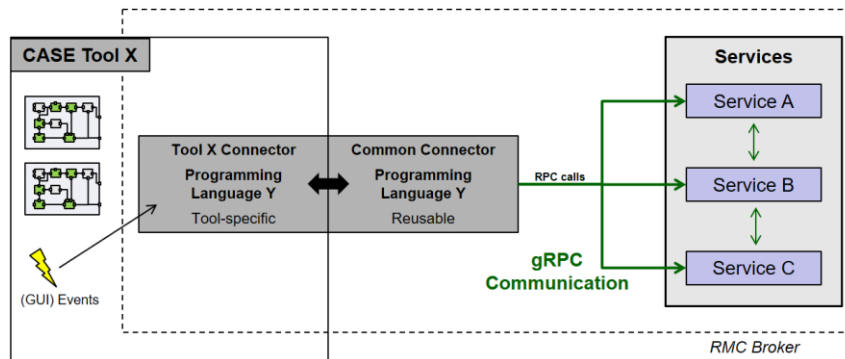


Fig. 8. Two-part CASE-tool Connector: Most of the connector remains reusable across multiple tools, including all the communication to RMC-Broker services. Only a small additional layer on top of this Common Connector has to be developed per tool, responsible for bridging between native CASE-tool events and mechanisms and the Common Connector.

Integration of CASE-tools into our RMC-Broker ecosystem is constrained mainly by two factors: the available means to extend a tool (add-on/plugin system, programming technology, etc.) and the available mechanisms for retrieving from, or contributing back

to the tools internal project data. Besides this, the logic to connect to the RMC-Broker is completely identical for all CASE tools.

To be able to reuse this communication logic, we have developed a two-piece connection to the tools, of which one contains most of the logic to accomplish RMC-Broker activities: the *Common Connector*. Being tool-agnostic and therefore reusable, it enables most CASE-tools to only require a very thin “glue layer” on top of it to be plugged into each their native extension mechanism. We call this thin layer *Tool Connector* - the second connector piece. The effort for developing this tool-specific connector depends largely on the capabilities of the CASE-tool in question. Basically, it comes down to handling tool user interface events and invoking Common Connector methods accordingly, as well as providing an interface for accessing tool project data. The concept of this two-piece tool connection is given in Fig. 8.

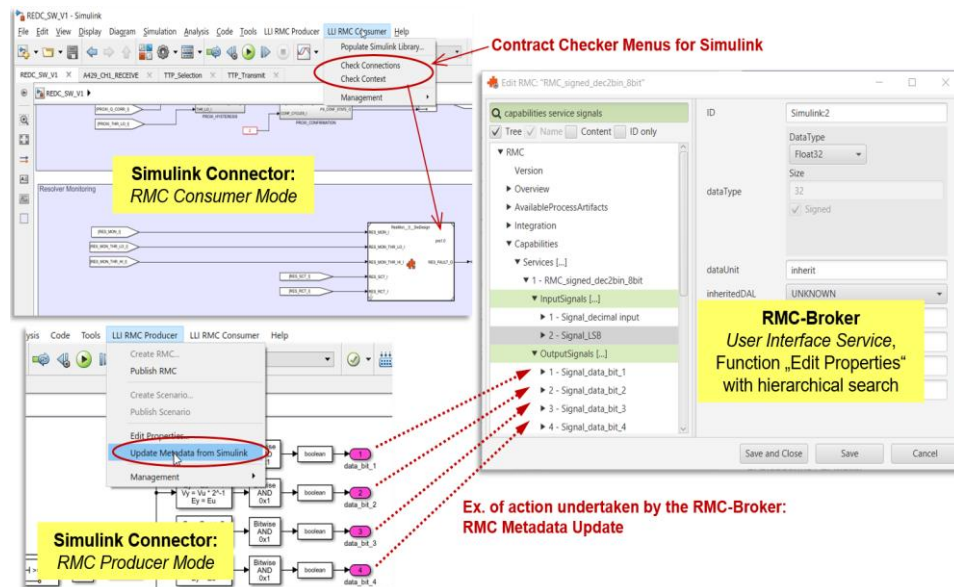


Fig. 9. Integration of MATLAB / Simulink with the RMC-Broker.

Each project model in any CASE-tool may contribute to different types of RMC process data. These model data can usually be fully exported only in files in proprietary file formats depending on the used CASE-tool (e.g. “.slx”-files for MATLAB/Simulink). These tool-specific files will simply be linked from RMC metadata and be stored alongside the other RMC process artefacts in the RMC repository. They are meant to represent the original source of the RMC information and will be used as such for work with the respective tools. To avoid having to enter all the RMC metadata manually, the connectors enable an automated extraction of important data from the CASE-tool project models, that are also relevant outside of the RMC itself, like interface properties. This provides other RMC-Broker modules with the basis necessary structured data to perform

their operations, one example being the Contract Checker service (see 2.6), which a. o. relies on data about the RMCs interface requirements. To accommodate this, we implemented this automatic data extraction for some tools in such a way, that it retrieves data from the project model and maps it to the RMC model structure. This operation is illustrated in Fig. 9: Ports from the Simulink model (lower left) are generated into the RMCs metadata (right). However, much data available in the CASE-tools project models does not correspond exactly to the RMC metadata structure. Some must be fetched from several tool-specific data sets, and transformed into RMC-compatible data kinds. Data transfer in the other direction is not trivial as well. Finally, not the whole RMC metadata can be found in the CASE-tools project models, also taken all together, and this missing data must be entered manually in the RMC via the RMC-Broker User Interface service.

3.3 Graphical User Interface

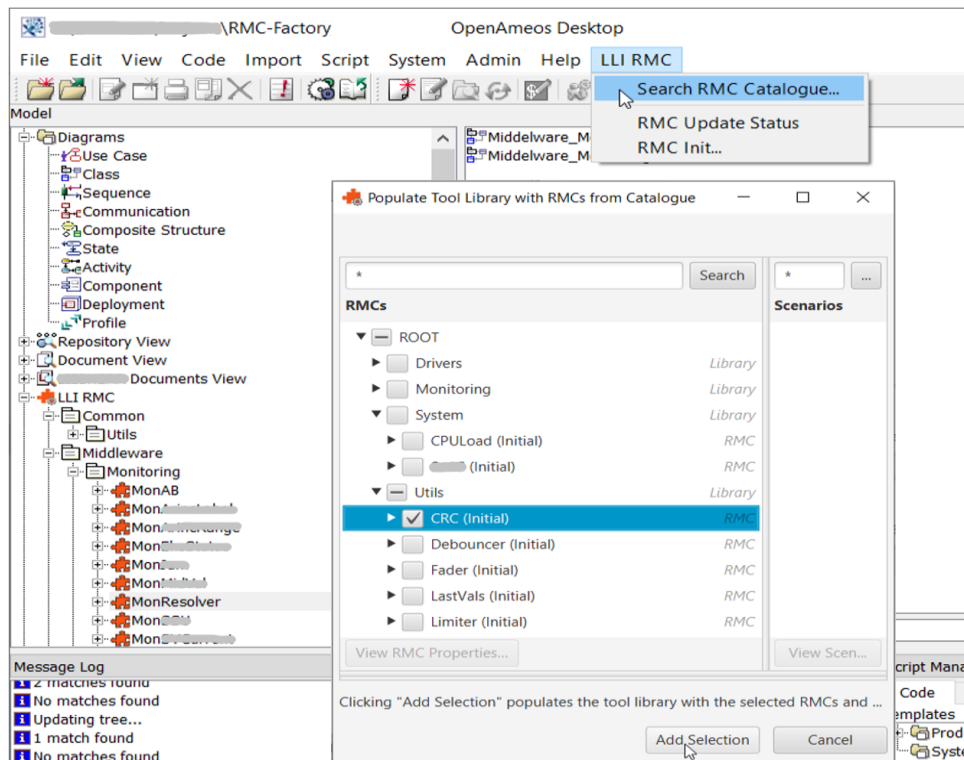


Fig. 10. Integration of the open-source UML CASE-tool AMEOS with the RMC-Broker. The dialog in the foreground is provided by the User Interface Service and therefore is the same in all tools. It lists all RMCs available in the central repository. Users can simply select RMCs and pull them into their local work environment with one click. The consuming project structure where the used RMCs are stored is independent of the RMC repository structure.

One of the most challenging aspects of such a multi-tool infrastructure is a unified graphical user interface across tools. Instead of using whatever a tool may offer to extend its user interface, we designed a *User Interface* service into our RMC-Broker architecture – leveraging the same underlying service technology as for example the Configuration Management service – while running on each users local machine. This not only makes it easy to reuse in multiple tools (the communication technology to services is already available), but also promotes a consistent and memorable user experience, no matter of the tool used. Important to note is the modular nature of this service architecture, i.e. a dedicated tool is not required to use this User Interface service. Examples of functionality provided by this service are presented in Figures Fig. 9 and Fig. 10, showing dialogs for editing all RMC properties (as defined in the RMC metamodel) and for consuming existing RMCs present in the central RMC repository, respectively.

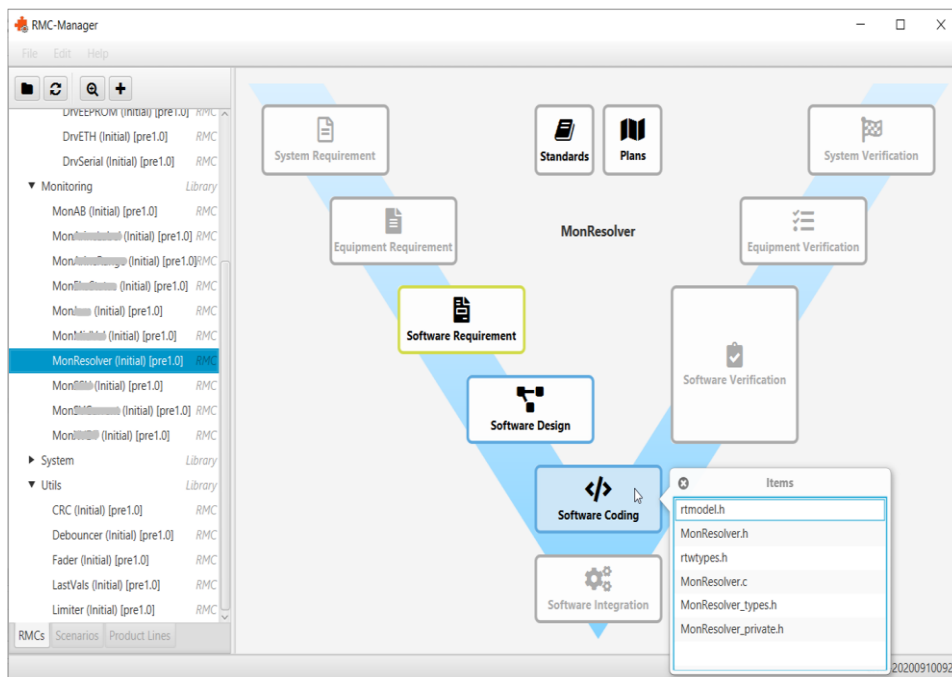


Fig. 11. RMC-Manager: GUI to manage all RMC process artefact categories independent to any CASE-tool.

Since users may not have an existing CASE-tool at hand at all times, we also developed a distinct RMC-Manager application, which provides standalone access to the RMC-Broker ecosystem. This application presents the user with a list of locally available components, as well as a V-model kind of overview (see Fig. 11), comprised of process artefact categories akin to DO-178C [RT11a]. RMC items (files) can be inserted per drag-and-drop in the related process artefact categories. The corresponding RMC data is

automatically completed. Most of the other data must be entered manually into the RMC, as no CASE-tool model is directly available as a data source. Maturity of the process artefact categories is indicated with different colors: none, in work, published, validated, and certified are the most important maturity levels.

4 Lessons learned

During the implementation of the RMC-Broker infrastructure, we constantly refined and adjusted our concepts based on the feedback we received from colleagues and future users. Moreover, not only the quality, but also the number of implemented features and of integrated CASE-tools within our RMC-factory concept foster the success of use. Hence, a major challenge for the implementation of such an infrastructure is the need to build an unobtrusive integration between our RMC management principle and many different kinds of CASE-tools. This drove us to develop the RMC-Broker principle which handles many aspects automatically, and which gives the impression that the graphical user interfaces for the RMC management are directly provided by the native tools because the user starts RMC-related actions from there.

Several RMC components have been developed and integrated in small case studies via the two CASE-tools and the RMC-Manager. The development of reusable software takes longer and costs more than software not designed for reuse, and the RMCs are no exception to this rule. Therefore, we focused especially on methods to promote reuse of existing reusable software, to take advantage as soon as possible, and to spread the development costs in the time and between several potential projects developed simultaneously. The most promising methods are the infrastructure concepts based on a catalogue of RMCs as the usage examples like the use scenarios and the SPL libraries. Furthermore the integration of specific process activities into the reuse global concept is vital to encourage project teams to search in the RMC-catalogue during their architecture design phase. The collaborative and incremental development principle is also significant to save time and costs.

The variability concept we presented here changed during the development and is radically different to the one we've started at the beginning. In the first place, it allowed much run-time flexibility in one unique kind of RMC. Widely used in many other domains, in the avionics this solution can lead to a very expensive verification – all flexibility combinations must be tested – as well as to complications during certification – difficulties to identify and verify deactivated code before runtime. We opted thus to have invariant RMCs – the Brick-RMCs – from which variants can be derived - which are invariants as well. From our opinion and confirmed in [Ri13] *Recommendation 9*, the reuse of small specialized, easy-to-understand bricks should happen much more often than for big components. Nevertheless, big components are necessary too, in order to match the system needs much faster – and also to convince the system-engineers to *re-use* instead to *re-create*. However, at this coarse-grained level, adaptability is compulsory – otherwise the reuse can only happen in exactly the same context (same

requirements, same hardware, etc.), what is very unlikely for complex functions at this level. Therefore, we conceived the SPL-RMC. As a consequence, the organization of life-cycle data, distributed between the Brick- and SPL-RMCs, became a little more complicated. But it was a necessary evil to combine certification and flexibility aspects.

After having changed our variability concept, we tried to implement a simple solution with existing SPL-tools. Unfortunately, we did not find any CASE-tool really fulfilling our requirements: most of them managed variability at source code level, were not easily customizable, or too strongly oriented to other industry domains. Several promising tools came from the open-source community, but were no longer maintained since many years. We found a limited but complete solution with the open-source tool FAMILÉ⁴ [Bu20] – though also no more supported – to reach our first investigation goals on the SPL method.

Last but not least, we noticed a potential upcoming issue when having many versatile RMCs to be integrated together. The numerous compatibility criteria between the components themselves and their integration context would generate an exponential complexity that the system and software integrators could not master completely before runtime. This convinces us to provide an early integration validation mechanism. For this reason we conceived and developed the contract-based integration concept. The intention is to give a first engineering confidence to the system engineers that the architecture is consistent – as another advantage to use the RMC factory. However, writing RMC-contracts is a complicated task and needs to be reserved to software and system experts to ensure the highest reliability. Even if the contracts should not be considered to prepare certification process data, they remain extremely useful as engineering validation to avoid possible corrective iterations appearing usually after regular integration tests.

5 Related work and publications

First, we need to mention the most relevant certification authority regulations addressing the reuse concern. The DO-178C [RT11a] and its clarification document DO-248C [RT11b] address the reuse aspect under the not reuse-specific topic “*Previously Developed Software*” (PDS), i.e. software already certified within a specific project. They consider it mainly as a *complete application software*, which must be upgraded or integrated in a new environment and therefore must probably be modified for different reasons – e.g. other requirements or other hardware. The DO-297 [RT05] contains guidance necessary to build and use IMA platforms and to design (system) architectures based on it. It focuses on system level (hardware and software) integration and mainly on certification processes. It slightly addresses software reuse, and exclusively as complete application. The AC20-148 [FA04] is a FAA guidance only, but covering all aspects of software reusability, in particular the specific development of components for

⁴ <http://www.ai1.uni-bayreuth.de/en/projects/FAMILÉ/index.html>

reuse and their integration in multiple projects. However, it addresses only big-scale components with all process artefacts like a complete avionics product, and is based on an older version of the DO-178.

Additionally to these official regulations, few other avionics publications nearly addressed our approach. The closest ones are summarized here. Leanna Rierson, FAA Designated Engineering Representative (DER) with Level A authority for both software and hardware, gives recommendations based on her real experience by dealing with software components in [Ri13]. Similarly as in the AC20-148 [FA04], she addresses the development and the reuse of high-level components, but only at the process point of view. Our approach has been inspired by several of these recommendations whose we have adapted for small components and SPL concept, for collaborative development, and for integration validation. F. Dordowsky and W. Hipp related their experience during the development of the helicopter NH90 in [DH09] and [DH10]. Due to its success, the NH90 had to be declined in a growing number of variants – up to 40 – for different countries and army types, during and after its ground development phase. To avoid resubmitting all software life cycle data for certification at each new system variant, the development team set ups a huge SPL program to certify new customers variants based on previous certified ones. Their approach covers variants of the same system with similar requirements and same ground hardware, but does not address software parts to be deployed on different hardware environments and in another requirement context.

Apart from avionics specific publications, we found many different concepts for reuse, some of which were relevant for our goals. In literature the approach we follow is denoted as *component-based software engineering (CBSE)*[Cr03][HC01]. The major concept hereby is to compose software of independent, reusable entities – i.e. components. This principle is not new and in the following we give an overview about existing related literature.

An important aspect in CBSE is how to design reusable components in such a way that they are as reusable as possible. Work on this can e.g. be found in the guidelines of Gill in [Gi03] or the reusability metrics by Sagar et al. in [SNS10]. We do not address these issues of how to define components with *good* reusability, but refer to the mentioned publications.

Other publications in the area of CBSE focus more on technical approaches – as we also do – but usually are limited to individual domains or artefacts: The case study [Sc05] by Schulte et al. presents an approach towards component-based reuse in avionics. However, they only consider design models and also do not take into account all artefacts that are necessary for certification. Similarly, Shani et al. present a solution for reusing models between different tools in [SB15], however they also do not use it for modular reuse of complete cross-domain components. In another direction as these works, Overhage in [Ov04] focuses more on aspects that are related to component interfaces and integration. Interestingly in comparison to our work is that they also propose a contract approach and also reason about necessary processes. However, their work targets a high level of abstraction and does not go into operationalization and detail

as we do.

The aspect which makes our work a contribution to this sketched state of the art on component-based engineering is the comprehensive approach in conjunction with a tooling concept. Hereby we not only offer the possibility to contribute to a reusable component from different tools, but also define the services and interfaces for a central repository to manage all RMCs.

As we aim on an integrated reuse solution for managing all required process artefacts, the integration between all editing CASE-tools for those artefacts is an important aspect. There are different approaches of how to integrate tools of which the approach of the *Open Services for Lifecycle Collaboration (OSLC)* gained a lot of attention lately [OS10]. Especially interesting about OSLC is that it is not only about a common data format (RDF in this case), but also specifies interfaces to interact within the tool integration environment. There also is work that applies OSLC to general systems engineering as by Saadatmand et al. [SB08] and even for the aerospace domain as by Vagliano et al. [Va17]. Yet, neither OSLC itself nor those mentioned works address the management and reuse of individual, small components with a central repository. As indicated, this however is one of our main focal points in the presented work. This fact in conjunction with the technical complexity of OSCL that is also shown by Leitner et al. in [LHM16] made us decide for a lightweight custom implementation instead of the OSLC interfaces approach.

6 Summary and outlook

After having established the concepts to improve the management of small and scalable reusable software components for avionics software, we have conceived and developed an ecosystem able to deal with small brick components and big flexible ones. Our approach combines methods, tool infrastructure and process extensions in order to promote an effective production and (re)use of components specifically designed at the beginning for being reused in the avionics domain. Our work has been driven by the design of a solution which facilitates the production and the reuse of certification process data, even if this one is only partly complete at the time of the first need by another projects. For now, the approach has been implemented in a tool infrastructure (the RMC-Broker) in which two CASE-tools have been integrated. Additionally, a standalone RMC-management application has been developed and integrated using the same infrastructure mechanisms. Several RMC components have been developed and integrated in small case studies via the two CASE-tools and the RMC-Manager.

One of our major next steps will be the integration of more CASE-tools to cover possibly all software process levels. A better long-term solution to manage SPLs is also planned. Other future works will be to develop more RMCs and to integrate them in a pilot project. More automatic filling of information into the RMC-models directly taken from the CASE-tools models is foreseen as well: this aims to accelerate the building of

the RMCs by reducing the manually transfer of data from CASE-tools models to the RMC models. This should also increase the acceptance of RMC consumers to also make their work available as RMCs. This future work will be supported among other by the research project IDEA.

Acknowledgment

This work is partly funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) through the projects ASSET 2 (grant no. FKZ 20Y1508E) and IDEA (grant no. FKZ 20Y1712G).

Bibliography

- [Bu20] Buchmann, Thomas, et al. "FAMILE: tool support for evolving model-driven product lines." Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications, CEUR WS. 2012.
- [Cr03] Crnković, Ivica. "Component-based software engineering-new challenges in software development." *Journal of computing and information technology* 11.3 (2003): 151-161.
- [DH09] Dordowsky, Frank, and Hipp, Walter: "Adopting software product line principles to manage software variants in a complex avionics system", 13th International Software Product Line Conference, 2009.
- [DH10] Dordowsky, Frank, and Hipp, Walter: "Implementing multi-variant avionics systems with software product lines", 36th European Rotorcraft Forum, Paris. CEAS, 2010.
- [FA04] Federal Aviation Administration, Advisory Circular AC20-148 "Reusable Software Components", December 7, 2004.
- [Gi03] Gill, Nasib S. "Reusability issues in component-based development." *ACM SIGSOFT Software Engineering Notes* 28.4 (2003): 4-4.
- [HC01] George T. Heineman and William T. Councill (Eds.). 2001. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [LHM16] Leitner, Andrea, Beate Herbst, and Roland Mathijssen. "Lessons learned from tool integration with oslc." *International Conference on Information and Software Technologies*. Springer, Cham, 2016.
- [OS10] Workgroup, OSLC Core Specification. "OSLC core specification version 2.0." *Open Services for Lifecycle Collaboration, Tech. Rep* (2010).
- [Ov04] Overhage, Sven. "UnSCom: a standardized framework for the specification of software components." *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*. Springer, Berlin, Heidelberg, 2004.

-
- [PBL05] Pohl, K., Böckle, G., and Linden, F. J.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
- [RG10] Richters, Mark, and Martin Gogolla. "On formalizing the UML object constraint language OCL." *International Conference on Conceptual Modeling*. Springer, Berlin, Heidelberg, 1998.
- [Ri13] Rierson, Leanna: "Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance", Chapter 24: Software Reuse, CRC Press, 2013.
- [RT05] RTCA Inc.: "Integrated Modular Avionics (IMA) – Development Guidance and Certification Considerations", SC-200, RTCA DO-297, November 8, 2005.
- [RT11a] RTCA DO-178C: RTCA Inc., "Software Considerations in Airborne Systems and Equipment Certification", SC-205, December 13, 2011.
- [RT11b] RTCA DO-248C: RTCA Inc., "Supporting Information for DO-178C and DO-278A", SC-205, RTCA DO-178C, December 13, 2011.
- [SB08] Saadatmand, Mehrdad, and Alessio Bucaioni. "OSLC Tool Integration and Systems Engineering--The Relationship between the Two Worlds." *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014.
- [SB15] Shani, Uri, and Henry Broodney. "Reuse in model-based systems engineering." *2015 Annual IEEE Systems Conference (SysCon) Proceedings*. IEEE, 2015.
- [Sc05] Schulte, Mark. "Model-based integration of reusable component-based avionics systems-a case study." *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. IEEE, 2005.
- [SNS10] Sagar, Shrdhha, N. W. Nerurkar, and Arun Sharma. "A soft computing based approach to estimate reusability of software components." *ACM SIGSOFT Software Engineering Notes* 35.4 (2010): 1-4.
- [Va17] Vagliano, Iacopo, et al. "Tool integration in the aerospace domain: A case study." *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. IEEE, 2017.