Efficient Software Component Reuse in Safety-Critical Systems – An Empirical Study

Rikard Land[1], Mikael Åkerholm[1], Jan Carlson[2]

[1] CrossControl, Västerås, Sweden
[2] School of Innovation, Design, and Engineering, Mälardalen University, Västerås, Sweden
rikard.land@crosscontrol.com, mikael.akerholm@crosscontrol.com, jan.carlson@mdh.se

Abstract. The development of software components to be reused in safety-critical systems involves a number of challenges. These are related to both the goals of using the component in several systems, with different definitions of system-specific hazards, and on the high demands of today's safety standards, which assume a top-down system and software development process. A large part of the safety-related activities is therefore left for integrator, and there is a risk that a pre-existing component will neither be feasible nor more efficient to use than internal development of the same functionality. In this paper we address five important challenges, based on an empirical study consisting of interviews with experts in the field, and a case study. The result is twelve concrete practices found to improve the overall efficiency of such component development, and their subsequent reuse. These are related to the component architecture and configuration interface, component and system testing and verification, and the information to be provided with the component.

1 Introduction

Safety-critical systems are systems which may, should they fail, harm people and/or the environment – such as vehicles, power plants, and machines. To develop such systems, one must demonstrate that potential hazards have been analyzed, and that all prescribed activities listed in an applicable safety standard have been performed. There are generally applicable safety standards, such as the IEC-61508, and domain-specific standards, such as IEC-61511, ISO-15998, ISO-26262, RTCA DO-178B/C, EN50126/8/9, ISO-13849, and IEC-62061. In the daily development work, achieving a sufficient level of safety boils down to adhering to the relevant standard(s).

These standards are based on an assumed top-down approach to system construction. Each system must be analyzed for its specific hazards and risks in its specific environment, and the system requirements must be traced throughout the development to design decisions, to implementation units, to test cases, and to final validation. The standards' general approach to the inclusion of pre-existing software components in a system is to present them as being an integrated part of the development project, and let them undergo the same close scrutiny as newly developed software for the specific system (which is inefficient). The standards in general provide very little guidance for potential developers of software components, intended for reuse in several safety-critical systems – with the main exceptions of the recently issued ISO-26262 and the advisory circular AC20-148 complementing RTCA DO178B.

For a reusable component to be included in a safety-critical system, the component developer needs to not only comply with the relevant standard throughout the life cycle, but also ensure that the integrator saves effort by reusing the component. In safety-critical systems, the actual implementation is just a small part of the "component" being reused and savings are lost if the integrator has to re-perform much or all of the safety-related work (e.g. verification, traceability, adaption of documentation).

This paper takes an overall view and intends to identify the most important challenges, as perceived by practitioners, and provide some guidance on how to address these challenges. Five specific challenges are (Åkerholm & Land, 2009):

- **Component interface.** The challenge is to define a well-specified interface (in a wide sense, including e.g. configuration parameters, restrictions on tools, assumptions on usage etc.) which does not unnecessarily restrict the integrator.
- **Component abstraction.** The challenge is to create a component which is general enough to provide the expected functionality in many different systems, while addressing e.g. traceability and deactivation of unused code.
- **Activities left for the integrator.** Many analyses and verification activities will necessarily be left for the integrator, and the challenge is to make this easy.
- **System level traceability.** Each system requirement has to be traced throughout all relevant project artifacts such as documents, design models, source code, and test cases; a challenge is to define a "traceability interface" so that component design decisions and assumptions can easily be linked to system hazards and contexts.
- **Certified or certifiable.** The challenge is to make the strategic decision whether to aim at certifying a component, or to develop it according to a standard and provide all relevant information with the component, packaged in a format so that the integrator easily can certify the system including the component.

This paper presents an empirical study, consisting of interviews and a participatory case study, resulting in twelve practices that address these challenges.

The research method is further described in section 2, and section 3 describes related work. Section 4 is organized per the challenges listed above and presents identified practices the component developer should perform. Section 5 concludes the paper.

2 Research Method

The purpose of the study is to collect valuable experience, but the extent to which the suggested practices improve efficiency is not independently validated. First four open-ended interviews were performed (see section 2.1). Secondly, as action research we used an industrial project (see section 2.2), applying some of the findings from the interviews. All observations were compiled (qualitatively), and the synthesized result is presented here, with the source of each observation indicated in the text.

2.1 First Phase: Interview Study

The four interviewees are listed in Table 1. We used AC20-148 as a template to construct the interview questions, added further open-ended discussion topics. The interviews lasted approximately two hours, with more than one author participating. The interviewees approved the interview notes after minor clarifications, additions, and corrections. The interview data is not intended for statistical analysis; the purpose was to collect valuable experiences.

Table 1. The interviewees and their background and experience.

Interviewee #	Background and experience
1	Experience as developer as well as independent assessor from a number of projects, according to e.g. IEC61508.
2	Experience as independent assessor from a number of projects in various domains, in particular railways (standards IEC-50126/8/9. Experience from development of safety-certified operating system.
3	Technical expert; the company develops a software component for avionics applications, approved under DO-178B / AC20-148.
4	Safety expert; the company develops a HW/SW platform, certified to several standards (IEC61508, IEC61511, ISO13849, and IEC62061).

2.2 Second Phase: Industrial Project

The development project was action research in the sense that it was from the start explicitly set up as a case study for our research, where we intended to implement some of the findings of the interviews. A reusable component was developed, implementing mechanisms to handle all data communication failures as specified in IEC 61784-3. The component was developed according to SIL3 of IEC61508. Some further technical details are described under each topic heading in section 4.

The authors were heavily involved throughout the project, as project manager, designer, reviewer, and verifier, together with other staff as well. This gives us first-hand insight into the project, but is also a potential source of bias. During the project, observations and ideas were recorded in a research log, which was studied at the end together with other project documentation. AC20-148 was used as a template for (part of) the safety manual of the component, describing e.g. activities left for the integrator. A limitation is that the component has not yet been included in a certified system.

3 Related Work

From the area of component-based software engineering, it is known that predicting or asserting system properties from component properties is difficult in general (see e.g. (Hissam, Moreno, Stafford, & Wallnau, 2003) (Larsson, 2004)), and particularly difficult for safety (Voas, 2001), partly because the safety argument is not embedded in the component itself but in the surrounding documentation and the rigor of the activities during its development. Among the few attempts to describe reuse of software in safety-critical software from a practical, industrial point of view, we most notably find descriptions of components pre-certified according to the AC20-148 (Lougee, 2004) (Khanna & DeWalt:, 2005) (Wlad, 2006), which describe some of the potential benefits of reusing pre-certified software, rather than provide guidance on how to develop a reusable software component efficiently as we do in this paper.

Common challenges of software reuse (Karlsson, 1995) also hold true for reuse of safety-critical software components; for example, there are various methods and practices addressing the need of designing a system with potential components in mind (Land, Blankers, Chaudron, & Crnković, 2008) (Land, Sundmark, Lüders, Krasteva, & Causevic, 2009). In general, there is more data and experiences on development *with* reusable components than development *of* reusable components (Land, Sundmark, Lüders, Krasteva, & Causevic, 2009), while the present study takes a broad perspective and includes both.

Literature on modularized safety argumentation provide several promising research directions, such as how to extend e.g. fault tree analysis (Lu & Lutz, 2002) and state-based modeling (Liu, Dehlinger, & Lutz, 2005) to cover product lines, that should in principle work also for composition of component models. A bottom-up, component-based process is described in (Conmy & Bate, 2010), where internal faults in an FPGA (e.g. bit flips) are traced to its output and potential system hazards. Such analyses should be possible to apply to components being developed for reuse, leading to a description at the component interface level, e.g. of the component's behavior in the presence. In the direction of modularized safety arguments, there are initiatives related to GSN (Goal Structuring Notation) (Despotou & Kelly, 2008) and safety contracts (Bate, Hawkins, & McDermid, 2003).

4 Twelve Practices that Address the Challenges

This section contains the observations made in the study, based both on the interviews and the development project, formulated as concrete practices the component developer should perform. The section is organized according to the five challenges listed in (Åkerholm & Land, 2009) and in the introduction of the present paper.

4.1 Addressing Challenge #1: Component Interface

The component's interface in a wide sense must be fully specified, including not only input and output parameters but also configuration parameters, restrictions on tools, the requirements on memory, execution time and other resources, and communication mechanisms (see e.g. AC20-148) (Åkerholm & Land, 2009).

Identification of Documentation Interface. A large amount of documentation related to the reusable component must be integrated into the integrator's life cycle data; the AC20-148 lists e.g. plans, limitations, compliance statement, and software approval approach. To make this as straightforward as possible, interviewees #2 and #3 give the advice to both component developers and integrators to follow the relevant safety standard as closely as possible with regards to e.g. terminology and required documents. According to the experience of interviewee #2, companies unnecessarily create a problem when using an internal project terminology and then providing a mapping to the standard. Interviewee #4 on the other hand, describes such a mapping from the platform's terminology to that of the standards it is certified against; however, since the same assessor (i.e. the same individual person at the certification authority) is appointed for all standards, this poses no major obstacles.

Still, the safety standards assume that the documentation structure is a result from a top-down system construction, and a component will need to specify for which part of this structure it provides (some of) the required documentation, and how it should be integrated into the system's documentation structure. When we followed the structure outlined in (Åkerholm & Land, 2009) in our project, we observed that the documentation interface is highly dependent on the technical content, due to the fact that design decisions on one level are treated as requirements on the level below. When defining a component for reuse, there are some specific challenges involved: the perhaps not obvious distinction between the architecture and requirements of the component, and it was realized in the project that the documentation needs to distinguish these more clearly than we did at the outset. Hazard and risk analysis for the component need to be performed backwards and documented as a chain of assumptions rather than as a chain of consequences; this needs to be documented very clearly to make the hazard analysis and safety argumentation of the system as straightforward as possible. Further research is needed to provide more detailed suggestions on how to structure the component documentation in order to provide an efficient base for integration.

Practice I: Follow the requirements of the standard(s) on documentation structure and terminology as closely as possible. Two important parts of a component's documentation interface are the component requirements and the component hazard and risk analysis, which should aim for easy integration into the system's design and hazard/risk analysis.

Identification of Configuration Interface. A reusable component should have a modular design and configuration possibilities, so that "hot spots" where future anticipated changes are identified and isolated (Interviewee #3; see also e.g. (Lougee, 2004)). Knowledge of the specific differences between customers and systems is required; interviewee #3 describes that their operating system has support for different CPUs, its ability to let the integrator add specific initialization code, and its support for statically modifying the memory map. With configurability come requirements on verification and testing of a specific configuration of a component in a specific system (interviewees #1 and #3). In our industrial project, we clearly separated the user configurable data from other data in the system, by setting up a file structure where only two files with a strict format are user modifiable. We used mechanisms provided by the source code language to both provide an easy-to-use configuration interface and the possibility of being able to statically include this data into the program with appropriate static checks (see also section 4.3 for construction of adaptable test suites).

Interviewee #3 describes that with configuration variables which are read from non-volatile memory during startup, the integrator needs to show that the parameters cannot change between startups. See section 4.2 on deactivation of dead code.

Practice II: Create a modular design where known points of variability can be easily expressed as configuration settings which are clearly separated and easy to understand for the user.

4.2 Addressing Challenge #2: Component Abstraction

Components suitable for reuse, in particular for safety-critical systems, need to address well-defined, commonly occurring design problems or commonly needed services (Khanna & DeWalt:, 2005). The product of interviewees #3 and #4 are indeed "platforms", in the sense that their components provide basic services on top of which applications are built. As such, the services they provide are of a general nature, such as partitioning of memory, which are not directly connected to a system's functional requirements. In the industrial project case, our main functional requirements come from the IEC 61784-3 standard on data communication in industrial networks, which defines all conceivable communication errors that need to be addressed, and which will be the same in many different systems, independently of the safety-critical functions they perform. All these components, as well as the published examples (Khanna & DeWalt:, 2005) (Wlad, 2006) of components constructed according to AC20-148, provide general functionality needed to address needs at the design level.

Practice III: Define the component functionality as well-defined abstractions solving commonly recurring problems on the system design level, rather than on the system requirements level.

Deactivation and Removal of Unused Code. Some of the features of a reusable component may not be used. In safety-critical systems, there is a very important difference between "dead code", i.e. unreachable statements left by mistake, and "deactivated code", i.e. program code deactivated with a hardware switch, configuration parameter in the program, or a runtime parameter (see e.g. RTCA DO-178B). Although it is preferable to identify unexecuted code and remove it altogether from the executable, the interviewees refer to the standards which do not prohibit deactivated code per se (e.g. RTCA DO-178B). In such cases, however, the interviewees stress that an argument must be provided showing that the code will not cause harm even if executed, and this must be supported by careful testing, including fault injection tests. Also, one must reason about possible side effects such as I/O operations and writes to shared variables or permanent storage in deactivated code (interviewee #2). The integrator also needs to provide an argument that the parameters cannot change between startups (interviewee #3; see also section 4.1); such argumentation is avoided if the code is statically excluded (interviewee #3 and case study). In our project, the source code used only by either senders or recipients are protected with macro definitions.

Practice IV: For deactivated code, base the safety argumentation on the avoidance of hazards, and be particularly observant on code with side effects. Whenever possible, replace runtime mechanisms for deactivating code with static mechanisms to remove the code completely from the executable. (Related to Practice II.)

Definition of High-Level Design/Architecture. Although some of the design of a reusable component is hidden from the integrator, and should remain so, the definition of a reusable component's high-level design is also, to a large extent, a definition of the architecture of an assumed system: interaction paradigms (i.e., messages, functions, etc.), execution models (i.e., passive libraries, active tasks, etc.), expected interaction patterns, semantics of the source code functions, etc. Standardization of these aspects have led to the definition of formalisms such as AADL (As-2 Embedded Computing Systems Committee, 2009), EAST-ADL, AUTOSAR, SysML and MARTE[1]. Interviewee #1 stresses that the architecture of the component reflects what the component developer believes to be useful for the integrator. This is strongly supported by our experiences from the industrial project, where we investigated four conceivable execution models on the receiving side of the communication:

- **Time-triggered.** Execution of a task is started periodically, which retrieves all newly arrived messages and processes them; this is suitable approach for a node with a real-time operating system.
- **Event-triggered (using hardware interrupts).** Execution of code is trigged by hardware interrupts, which are either "a message has arrived" or a timeout. This is suitable for an otherwise interrupt-driven system.
- **Event-triggered (infinite loop with blocking wait).** The code hangs on a "wait for message" function call, which returns when a message has arrived or when a timeout limit has been reached. This blocking approach may be suitable when communication with other tasks is limited.
- **Continuous polling.** The application implements an infinite loop, that first reads data (if any) from the bus and then handles it, in one single thread without any delays or interrupts. This "busy waiting" approach is suitable for a node which have no other tasks to do, or where those tasks can also be performed in the same loop.

Our component is a passive library component to be called by the application to process messages rather than an abstraction layer.

Practice V: Define the execution models, interaction paradigms, etc., of the component, to support the assumed architecture(s) of many potential target systems.

To verify in the design phase that our designed API would support the four execution models, we a) wrote pseudocode for each of these (this later became part of the component's usage documentation), and b) let developers review the design given the question: "Could you create a good system design with this component?" Through this somewhat iterative analysis and design work, we were able to create an API, i.e. functions and rules for interaction, supporting all four execution models. However, due to the lack of firm boundaries in terms of requirements from a specific system, this activity required significantly more effort than expected.

Practice VI: Allocate a team to evaluate the feasibility and usefulness of your component at the early conceptual and architectural design stage. Allocate sufficient time in this phase for the necessary development and iterations of design proposals.

Structure of Component Design Artefacts. In our project, we first planned for one single software architecture document. We realized later in the project that the architecture of a reusable component is a mixture of both 1) inputs to the requirements (e.g., "the component shall support the following four execution models") and

[1] http://www.autosar.org/, http://www.sysml.org/, http://www.omgmarte.org/

2) implementation decisions made to fulfill the requirements (e.g. definition of data structures and functions, including traceability information to requirements). This caused an unnecessary circular dependency between requirements and architectural design documents. We therefore recommend that these two types of architectural information are kept distinct in separate documents, one being an input document to the requirements specification and one being a downstream document. However, this was perceived to be a clarity issue, not a real threat to safety or project efficiency.

Practice VII: Use separate documents for the external architecture (the assumed architecture of the system) and the component's internal architecture and design.

4.3 Addressing Challenge #3: Activities left for the integrator

There will remain a number of activities for the integrator, related to the context and environment of the component in a specific system. The challenge for the component developer is to aid the integrator in these activities by providing the component with certain information and artefacts. In the studies, we identified what can be labeled "analysis interface", and adaptable test suites as two important means for this.

Identification of Analysis Interface. Data coupling analysis, control coupling analysis, and timing analysis are examples of activities that can only be performed by the integrator, when the complete software is available (AC20-148). However, some analyses may in principle be partially performed at the component level, or some useful data or evidence may be constructed at the component level. In spite of research on composing system properties from component properties (see e.g. (Hissam, Moreno, Stafford, & Wallnau, 2003) (Larsson, 2004) and the TIMMO project[2]), the challenge remains to identify such analysis interfaces, including assertions that need to be made by the component developer, properties that need to be specified, and how to use these automatically in a system-level analysis. In the study, interviewees #3 and #4 mentioned timing issues to be especially important. With a simple application design, and certain component information, it may be sufficient to perform timing measurements of the integrated system, given that the component developer makes assertions on the behavior of the component. The current state of practice includes, according to interviewees #3 and #4, component assertions that the function calls are non-blocking, or information that the component disables interrupts, which is valuable for the integrator's more detailed timing analysis. Also, a specification of input data which will cause the longest path through a function to be executed, and/or the path that includes the most time-consuming I/O operations, is useful for finding upper bounds on the timing within a specific system and on a specific hardware.

Practice VIII: Provide information on the component's (non-)blocking behavior, disabling and enabling of interrupts, and input data which is likely to cause the upper bounds on timing, to facilitate the integrator's system level analysis of e.g. timing.

Adaptable Test Suites. The component of our project is delivered with a module test suite which automatically adapts itself to the configuration. The component configuration is made through macro definitions and filling static data structures with

[2] http://www.timmo.org/

values, and the test suite is conditionally compiled based on the same macros, and uses the actual values of the data structures to identify e.g. boundary values to use in testing, and of course determine the expected correct output. The test suite includes all necessary fault injection in order to always achieve sufficient code coverage (for the SIL 3 according to IEC-61508).

The creation of a module test suite on this higher level of abstraction forced us to reason about many boundary values, possible overflow in computations, and similar border conditions. Also, it helped us identify user errors, such as what would happen if the component is configured with empty lists or inconsistent configuration parameters. In addition, the resulting number of actual tests executed on a single configuration is significantly higher than we would otherwise have created, which also increases our confidence in the component, although strictly the number of test cases can never in itself be an argument for testing being sufficient. Thus, as a side effect, this greatly helped us to design for testability, and to design good test suites.

The main purpose of providing adaptable test suites is that the integrator easily can perform module tests on the specific configuration used in the system. To verify the configuration mechanisms and the test suite itself, we created a number of configurations and re-executed the tests with very little effort (a matter of minutes). This increased our confidence, not only in the component itself but in that we are saving a significant amount of effort for integrators. (However the integrator must learn and understand how to run the test suite correctly for a specific configuration, and how to interpret the test output (including verification of the code coverage reached).) Another extra benefit is that some changes (e.g. addition of messages on the bus) can be made late in the development process and easily re-tested.

The test suite is written in ANSI-C and is therefore as portable as the component itself, but the fault injection mechanism and the code coverage analysis rely on external tools and therefore somewhat restrict the integrator's freedom. To account for this, we have designed the test suite and test environment so that adapting the suite to another tool set should not be too effort-consuming.

Practice IX: Deliver an adaptable test suite with the component, so that the integrator can (re-)perform configuration-specific testing with little effort.

4.4 Addressing Challenge #4: System level traceability

Demonstrating *traceability* means tracing each requirement to design items, implementation items, test cases, etc. This requires extra attention when a part of the system is developed by an external company prior to and/or independently of specific system requirements since the traceability chain goes across organization boundaries.

Identification of Traceability Interface. Some steps towards defining a traceability interface were identified in the study: if the component provides an abstraction with error handling (such as operating systems, communication layers, or platforms in some other sense), it may be sufficient to demonstrate that the component's functional interface solves some of the design goals of the system (e.g., that it handles certain types of communication failures with a certain level of integrity) without introducing new hazards, that the component is verified sufficiently (e.g. using code coverage metrics), and that it is used as intended and its safety manual has been followed (interviewee #4; our project). Interviewee #2 in particular stresses that the objective when arguing safety is to perform the argumentation in relation to the system hazards; if a fault analysis (e.g. a fault tree analysis) shows that a component does not contribute to a specific hazard, the tracing may stop there.

Practice X: Specify component requirements and functional interface, so that a detailed traceability analysis is not required when integrated into a system. This includes providing a safety manual with assumptions and rules for component usage.

Standardization of Traceability Tools. Often traceability is managed manually as tables in electronic documents, and even if a traceability tool is used, there are problems to share the same database, and it is also likely that the component developer uses a different tool than its customers (interviewee #1). This is a challenge for standardization and tool developers, rather than for component developers or integrators.

Meeting the Requirements on System Hazard and Risk Analysis. Normally, the system hazards are used, with their estimated frequency, consequence etc., to determine the SIL level (or similar; the standards have different classifications), which influences all downstream activities. When developing a component for reuse, the risk analysis is instead performed backwards: a target market is selected, and the component is developed according to common requirements and a SIL level which it is believed that integrators will require. It is only in a system context safe external behaviors in case of detected failures can be determined (e.g. to shut down the unit immediately, apply a brake, or notify the operator; it may or may not be safe and desirable to first wait for X more messages in case the communication recovers; etc.). It is always the responsibility of the system developer to focus the argumentation around the hazards and show that they cannot conceivably occur. "A general software component does not have safety properties, but a quality stamp. Only in a specific context do safety properties exist." (interviewee #2)

In the case study we performed some analysis based on assumed, realistic, values for usage and disturbances to demonstrate that an average system or application using our component also meets the hardware requirements at the target SIL level. (It may be noted that there is no major differences in the requirements on development of software between SIL2 and SIL3 according to IEC61508; the requirements on hardware however typically impose more expensive solutions including redundancy etc.)

Practice XI: Lacking a definition of system hazards, identify component error-handling, fault tolerance mechanisms, and behavior that are common for many systems, as independently as possible of the specific system hazards. (See also Practice III.)

4.5 Addressing Challenge #5: Certified or certifiable?

A developer of a component intended for reuse needs to make a decision between certifying the component, or developing the component according to a target standard and handing over all the safety-related documentation for the certification of each system. This decision is dependent on the situation of the component developer. This section lists the goals that were mentioned by the interviewees in the study, and describes some of their considerations in meeting these goals.

Goal 1: Saving Effort, Time and Money for the Integrator. Since component development is carried out according to the standard, and much of the required documentation and evidence is created, the integrator may potentially save the same effort (interviewee #1; see section 4.1). However, for interviewees #3 and #4, the effort savings for the integrator are not so significant. Interviewee #4 shared his experience of a component not developed according to the required safety standards, which brought a significant additional cost to construct the required evidence and documenting it. Interviewee #3 states that with AC20-148, the effort spent by the certification authority is decreased since only changes of the component have to be investigated.

Goal 2: Reducing Risk for the Integrator. Interviewees #1, #3, and #4, all state that with a pre-certified component (or a certifiable component, which has been used in another, certified, system), the confidence is high that the component will not cause any problems during system certification. Interviewee #3 specifically mentions that the customers using the component from his organization do it because the component is pre-certified according to AC20-148 and thus is a low-risk choice.

Practice XII: If the main goal is to present a component as risk-reducing, the component developer should consider certifying the component. If the main goal is to save efforts for the integrator, it may be sufficient to develop it according to a standard, and address effort savings in the ways outlined in this paper.

5 Conclusions and Future Work

Twelve practices for development of reusable software components for safety-critical systems were identified in an empirical study with interviews with industrial experts and an industrial case study. Being based on five previously identified challenges (Åkerholm & Land, 2009), they potentially represent important effort savings.

Further empirical studies, complemented by theoretical research, are needed, to further define many of the details relevant for a component interface, such as guaranteed behavior in the presence of (certain) faults, or a demonstration that "component-level hazards" have been appropriately analyzed and addressed.

Not to be underestimated is the potential gain in efficiency through standardization of platforms, tools, languages, etc. In the long term, safety standards also need to evolve to recognize the possibilities of reusable software components, while continuing to ensure systems' safety integrity. Our participation in the large European SafeCer[3] project provides an opportunity to study also other industrial cases in order to collect further good practices and to validate the conclusions brought forward in the present paper. Methods and notations to support modularized safety argumentation, such as those described in the related work section, will also be further developed, applied, and evaluated. Finally, the project aims at influencing future editions of safety standards to incorporate sound practices and methods that will make it easier and

[3] http://www.safecer.eu

more economical to build safety-critical systems from pre-existing components, while ensuring that they are still at least as safe as with the current standards.

5.1 Acknowledgements

We want to thank the interviewees for sharing their experiences with us. This study was partially funded by The Knowledge Foundation (KKS), CrossControl, ARTEMIS Joint Undertaking, and Vinnova.

6 References

1. Mikael Åkerholm and Rikard Land, "Towards Systematic Software Reuse in Certifiable Safety-Critical Systems", *in RESAFE - International Workshop on Software Reuse and Safety*, Falls Church, VA, 2009.
2. Scott A. Hissam, A. G. Moreno, Judith Stafford, and Kurt C. Wallnau, "Enabling Predictable Assembly", *Journal of Systems & Software*, vol. 65, no. 3, 2003.
3. Magnus Larsson, "Predicting Quality Attributes in Component-based Software Systems", Ph.D. Thesis, Mälardalen University, 2004.
4. Jeffrey Voas, "Why Is It So Hard to Predict Software System Trustworthiness from Software Component Trustworthiness?", in *20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, 2001.
5. Hoyt Lougee, "Reuse and DO-178B Certified Software: Beginning With Reuse Basics", *Crosstalk – the Journal of Defense Software Engineering*, December, 2004.
6. Varun Khanna and Mike DeWalt:, "Reusable Sw components (RSC) in real life", in *Software/CEH conference*, Norfolk, VA, 2005.
7. Joe Wlad, "Software Reuse in Safety-Critical Airborne Systems", in *25th Digital Avionics Systems Conference*, 2006.
8. Even-André Karlsson, *Software Reuse : A Holistic Approach*. ISBN 0 471 95819 0: John Wiley & Sons Ltd., 1995.
9. Rikard Land, Laurens Blankers, Michel Chaudron, and Ivica Crnković, "COTS Selection Best Practices in Literature and in Industry", in *Proceedings of 10th International Conference on Software Reuse (ICSR)*, Beijing, China, 2008.
10. Rikard Land, Daniel Sundmark, Frank Lüders, Iva Krasteva, and Adnan Causevic, "Reuse with Software Components – A Survey of Industrial State of Practice", in *11th International Conference on Software Reuse (ICSR)*, Falls Church, VA, USA, 2009.
11. Dingding Lu and Robyn R. Lutz, "Fault Contribution Trees for Product Families", in *13th International Symposium on Software Reliability Engineering (ISSRE'02)*, 2002.
12. Jing Liu, Josh Dehlinger, and Robyn Lutz, "Safety Analysis of Software Product Lines Using State-Based Modeling", in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005.
13. Philippa Conmy and Iain Bate, "Component-Based Safety Analysis of FPGAs", *IEEE Transactions on Industrial Informatics*, vol. 6, no. 2, 2010.
14. George Despotou and T. Kelly, "Investigating The Use Of Argument Modularity To Optimise Through-Life System Safety Assurance", in *3rd IET International Conference on System Safety (ICSS)*, Birmingham, 2008.
15. I Bate, R Hawkins, J McDermid, "A Contract-based Approach to Designing Safe Systems", in *8th Australian Workshop on Safety Critical Systems and Software (SCS'03)*, 2003.
16. As-2 Embedded Computing Systems Committee, "Architecture Analysis & Design Language (AADL)", Document Number AS5506, 2009.