International Workshop on Critical Software Component Reusability and Certification across Domains

Pisa, Italy, 18 June 2013
in conjunction with ICSR'13

Summary of proceedings

1 Introduction

The workshop stemmed from the following observations:

- Compositional reuse and certification become complicated when combined with safety
- Many working groups and research projects are investigating component-based software reuse for safety-critical applications
    - The implications of safety issues for the standard/certification processes.

The workshop aims to improve reusability of safety-critical components and to promote cross-domain solutions from single-domain approaches that satisfy cross-domain challenges.

The following topics were of special interest to the workshop:

- **Cross-domain reuse**
    - Where do we stand with it with respect to industrial practices or standards?
      Do standards support cross-domain reuse? How is this issue handled in practice?
    - Cross-domain reuse of components and of processes: how can we use components in applications complying with different safety standards – this is especially important for low level components (e.g. on communication layer) which are potentially used in different domains. This also relates to the definition of processes supporting different standards.
- **Enrichment of component specification**
    - Can it facilitate cross-domain reuse?
- **Component contracts**
    - Can this be a direction to push for?

1.1 Workshop materials

The abstracts and the presentation slides are available from the workshop web pages (http://www.intecs.it/CSC2013/).

2 Summary of the discussion

2.1 Reusing components across domains

Different standards require different approaches for software components and artefacts to be available, especially in the safety domain. One of the first problems encountered is that each context lives under its own specific safety standard: they are quite different to one another, and in some cases even in contrast, which causes undue difficulties in reasoning across domains.

Domain-neutral harmonization and factorization of domain-specific safety standards has recently become an active research area, currently addressed by some EU R&D project, but with no known deployment in industrial practice.

It may be argued that the current situation with regard to reuse practices in certification-susceptible domains is largely unsatisfactory for industry, as safety certification may easily entail an overload of cumbersome work: the ESA project IRIS, for instance, is facing the problem of applying different Standards (e.g. ECSS and DO-178B/C) to the same software to be certified.

2.2 Design component to be reused vs design by reuse

In this context, we distinguish the design of system that can be performed by reusing some existing components (design by reuse) and the design of components so that they can be reused in the design of different systems (design for reuse). There is a clear push by industry towards design by reuse in order to decrease the cost of development and possibly also of certification. However, design by reuse is not possible if components have not previous designed for reuse.

At present, designing components for reuse is not a frequently used industrial practice, despite the fact that some domain-specific standards (such as the ECSS for space) are supposed to take this aspect of development into account (requirements on "software intended for reuse" were already there in ECSS-Q-80B, 2003). It is a fact however that the development of new systems often has a product-line connotation, which causes the development team to identify parts (at all levels of abstraction, including requirements) that can be reused.

Furthermore, components are often used in different projects with little modifications. This causes several problems:

- if a component has not been developed with reuse in mind, its adaptation to a new context can be hard to achieve;
- we do not only want to reuse the components, but also related safety argumentation. Little modifications in the component however almost always cause a completely new safety argumentation

The difficulty with combining “design by reuse” with “design for reuse” stems from the difficulty with combine the top-down development style intrinsic to system design with the bottom-up development style typical of components design. Although some approaches such as platform-based design try to tackle this problem, more work has to be done to have an integrated top-down and bottom-up process in the industrial practice.

2.3 Component reuse vs the right side of the V process

Safety is a system property, therefore evaluating the safety of a reusable component on its own has little meaning: even if we describe our components with contracts, with extra functional properties etc., validation must be done on the system as a whole. Reuse makes sense if time and money can be saved through it. If all or most of the development activities have to be re-done anyhow, reuse is certainly less attractive.

In the case of a software component developed for intended reuse and then included in a software system submitted to certification, what can be saved with respect to to a component developed from scratch? What can we really reuse (in terms of artefacts and consequently spare in terms of time) when safety certification is concerned? It is likely that the certification authority will recognize the effort spent in the design, the results of the unit tests, and a few other items, however the most expensive development activities (analyses, verification, validation) could be required to be repeated in the target environment, because the behavior of that software component on the software system as a whole could be quite different, in terms of safety, from the one verified/validated in isolation or on a dummy system.

All this could lead to the conclusion that developing a SW component for reuse in safety-critical systems might not be cost-effective. And a cross-domain approach is likely to worsen the situation. The same considerations may apply even if safety certification is not involved, but just high dependability.

How can we measure or estimate the usefulness of reuse and how can we decide for which components reuse is economically beneficial? E.g. if we have a component which could potentially be reused with modifications, it has to be decided if it is useful to make modifications or if it would be better to develop a new component.

On the other side, there also need to be a way to decide if a component should be developed for reuse or just for a specific context. This is also a challenging problem, because it requires a good prediction of future needs.

Safety is a system property, is not linearly scalable. How do you scale? How do you decompose safety?

Decomposition is one important point, because different system views require different decompositions, which in turn results in different models which have to be kept consistent. This means that depending on the considered system aspect the decomposition will be different.

2.4 Certification and component reuse

In general, certification is applied to respect to a concrete specific system, assessing how the particular hazards of this system have been identified and how the system is constructed to reduce the risks or consequences of these hazards to acceptable levels. As a part of certification, a safety case in form of an explained and well-founded (i.e. valid evidence supporting the safety goals) structured argument [4] is often required to show that the system is acceptably safe to operate. Building a safety case is based on human reasoning and expert judgment and is mainly manual or semi-automatic work.

The basic idea behind the work [5] is that, although originally formulated in the context of a particular system, some parts of this reasoning would apply also in other settings. For example, in case the chain of events (fault, error propagation, failure, etc.) that leads to a hazard can unambiguously be identified and in case detection as well as recovery mechanisms have been designed and allocated onto a specific composite component, if that

component is reused within a system that is characterized by a similar chain of events, its fault-tolerant behavior is still valid in meeting the hazard avoidance goal.

In particular, when the system is developed using a component-based approach, it would be worth identifying parts of the reasoning that address only a particular component, and making no or few assumptions about the rest of the system. If such information can be associated with the component, and any assumptions be explicitly formulated, it could be reused whenever the component is reused in a new system. To capture this reusable information, we propose a contract-based approach, since it provides support for capturing not only the properties guaranteed by the component, but also under what context assumptions that these properties can in fact be guaranteed.

In addition to assumptions and guarantees of the component, captured by the contracts, we also want to reuse the argumentation why this information is trustworthy. For instance, if evidence is available (e.g. verification and validation results coming from testing or other verification activities) to support the claim that the above mentioned fault-tolerant component is actually meeting its hazard avoidance requirement, the fragment of argumentation used to show that a certain hazard has been avoided/mitigated can be reused.

2.4.1 Compositional certification

The aim of compositional certification is to reuse pre-existing set of certification data (partial or complete) of components (within a single domain or even cross-domain), and thus to reduce the time to build, assess and certify the systems. One possible approach is to use the Goal Structuring Notation (GSN; read more on it at: http://www.goalstructuringnotation.info/) with modular extensions to represent safety cases as a means to support compositional certification.

The challenges are how to capture the properties of each system component (and evidence of those properties) as a contract, and how to combine and match the contracts as part of a compositional certification approach.

2.5 Reusing in a model driven approach: benefits and limitations

Models and model-driven approaches seem to be an enabler for reuse in a safety-critical context. Models are essential artefacts as long as they are generative.

Models are rich yet abstract artefacts that could be usefully reused for analysis and code generation. Their reuse however is not an easy matter as semantically-relevant assumptions may be hidden “under” the model notations, and prove incompatible in different contexts.

Another very important issue is what model-based validation means, which obviously is only useful when the model represents all the “important” aspects of the reality of interest.

2.6 What can be reused?

Reuse in a safety-critical context has several aspects:

- Reuse of components in a single domain:
  Here, the main question is what the appropriate level of abstraction for reuse related to safety is?
  The general sentiment is that reusing large-grained component can be too complex because a large component provides a substantial functionality, which may not exactly match the requirements and

encompasses a large number of assumptions on its external environment, which may be difficult to accommodate. Furthermore, relying on a huge number of assumptions contains a lot of economical risk – parts which have been developed will maybe never be used. In that case, complex variability modeling and handling would have to be enacted to allow reuse in different contexts.
It is important to understand what the most effective granularity of components for use in safety domains is. And what type of abstraction and what type of composability can be applied.
Common approaches focused on component based design allow keeping functional issues separate from extra-functional ones; for instance real-time properties are particularly tied to the execution platform and are more subject to change considering the continuous upgrade of the computational power of the HW parts.

- Reuse of components across domains:
  Different safety standards require different development approaches and the availability of different artefacts. One of the first problems encountered is that each context lives under its own specific safety standard: they are quite different to one another, and in some cases even in contrast, which causes undue difficulties in reasoning across domains.
  Domain-neutral harmonization and factorization of domain-specific safety standards has recently become an active research area, currently addressed by some EU R&D project, but with no actual practice in industry.

- Reuse of processes:
  The requirements of different safety standards on the processes are slightly different. Therefore, the transfer of processes between different standards is hard to achieve. Different European R&D projects work on the harmonization of processes and on the investigation of commonalities between the various standards.

2.7 Configurable safety element

One idea was to use the concepts of variability management (product line) to support reuse of safety elements in one domain and across domains.

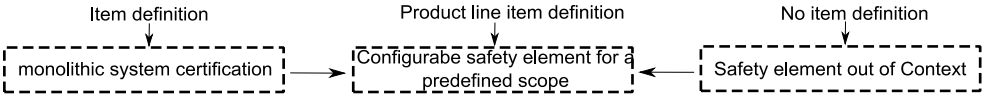
In order to improve reusability and thus increase flexibility in the development of safety-critical systems, ISO 26262 [Part 10] introduces the concept of Safety Elements out of Context (SEooC). This concept seems to be a good compromise between the need for component reuse and system-wide certification. The main idea is to make assumptions about safety requirements and the required ASIL. Therefore, the SEooC can just be verified according to these assumptions. A validation is only possible later in item development when the safety element is integrated in the system context. One main problem is the fact that assumed requirements often do not completely match the actual, ”in-context” requirements and thus, the safety component has to be adapted in order to be used.

In the context of a product line, the set of possible systems is more or less fixed enabling the specification of a set of possible, satisfiable safety requirements. The core idea is the definition of a set of safety requirements

in domain engineering which can be fulfilled by different configurations of this safety element. The safety requirements are provided as variants in a variation point, i.e. the variation point ”Safety requirements” lists all

supported safety requirements. In the same way, supported ASILs can be defined in a variation point ”ASIL(s)”. The implementation has to provide configuration (variability) mechanisms in the system safety design.

Configurable safety element therefore means that the element explicitly describes the provided variability. This means that a component is neither developed completely out of context nor for one specific context. This could also incorporate the definition of different standards as different contexts.



Regarding configurable safety elements the following research questions were raised:

- Is it possible to design configurable safety components for a predefined scope?
- Is it possible to fully validate configurable components with a predefined scope?
- Is it possible to support cross-domain reuse with this kind of configurable safety elements?

Configurable safety elements: variant configuration for product line could be extended for cross-domain. Variability factors should be captured. System to Software product line: SW components to be reused in different contexts. We lack representation for variability at model level.

Especially in the context of product lines there is a need for a safety process. Safety-critical product lines seem to be of interest in industry, but are still quite challenging.

3 Use of contracts to foster reuse

Both research and industry people agree that contracts are a useful means of expressing properties and requirements in view of reuse.

Again the level of abstraction/decomposition for contracts is an important issue: surely they should be used from the higher level of abstraction; at the lower levels they are more complex to be used, though could be particularly useful in terms of defining constraints on the context for component reuse. A tradeoff must be found so that introduction of contracts to define reusable components is worthwhile.

It is important to note the recent development in different projects and research groups of new methods, languages, and tools to support contract-based design at the system level. The contracts at software level are typically more complex due to the details of the functionalities to capture. On the other end, contracts at system level must be richer in terms of expressiveness because they must capture also extra-functional properties including real-time, performance, safety, and security aspects.

We can define contracts for different aspects (functional and non-functional) of a system and at different levels of abstraction (e.g., system level or atomic component level) [2] . Based on that we have different types of contracts e.g., safety, timing, functionality, type/value etc. Since not all aspects can be reasoned about on the same level, different contracts should be used at different levels of abstraction. For example, within object-oriented programming languages Ada 2012 and Eiffel, we can define contracts based on pre- and post-conditions constraining type and value of a method's inputs and outputs. While at this atomic level we can reason about type and value properties, in order to reason about timing or safety we need to move to higher levels of abstraction.

In [5] we envision a component-based software engineering approach that uses a rich component concept encompassing implementation, interface description, extra-functional properties, contracts, and argument fragments. We also assume clear separation between out-of-context component types and in-context component instances. Moreover, we assume a hierarchical component model, where a component type can either be primitive (directly implemented by code) or composite (implemented by interconnected subcomponents). By using rich component concept that encompasses assumption/guarantee contracts and argument fragments (used within safety argumentation), we are aiming at enabling reuse of argument fragments across contexts alongside components.

In this setting, contracts are associated to component types and specified with assumption/guarantee reasoning in mind, with a clear distinction between strong and weak assumptions and guarantees. In the traditional contracts, assumptions are used to express constraints on the environment of the component, i.e., on the components connected to its interface. We propose to broaden the scope of assumptions to include not only the component environment in terms of other connected components, but also usage context (e.g. frequency of service usage), hardware context (e.g. available memory), development context (e.g. compilers) and system context (e.g. system hazards). For example, within component instance contracts we can have timing contract specifying some low-level timing properties (e.g. worst case execution time) for which we assume timing properties of other components. For moving this information to a component type level (i.e.,

out-of-context) we need to broaden the scope of our assumptions to e.g. hardware platform and assume properties such as processor type or its frequency.

This broadening of the scope of assumptions implies that more assumptions are needed, meaning that the number of assumptions will increase. We plan to handle the broader context by customizing strong and weak assumptions and guarantees [3] within contracts to be able to specify a number of weak assumption/guarantee pairs on top of common strong assumptions and guarantees. This way we can separate those properties that must hold for all contexts/environments and properties that are context specific.

From reuse perspective, different aspects of components require different number of assumptions to be specified. More assumptions we make on the environment narrows down the number of environments in which we can reuse this information, i.e., which satisfy the assumptions. For example, if we take the timing contract specifying low-level timing property of a component, we need to capture all assumptions that influence the value of that property e.g., platform properties, hardware context, development context etc. Such property would only be valid in the exactly same environment, which is not likely to repeat itself, but this information on the behavior of the component is still beneficial in the process of selecting component.

3.1 Languages needed to express/verify contracts and their composition

Should contract be formal or informal? On the one hand, informal contracts are more readable and thus closer to the stakeholders' view. Also, regarding certification, it was argued that the core of the certification process is negotiation, which involves human interaction and judgments and cannot be automated. On the other hand, others argued that formal semantics on contracts are needed in order to derive what we have to guarantee on final components.

Formal approaches are currently available to allow contract formalization, formal verification of contract refinement and formal verification of contract against the implementation. FBK presented a concrete example of expressing contracts in a formal language that uses terms and syntactic constructs quite close to natural language: this language is used by their formal verification engine.

Use of strong and weak assumptions/guarantees to distinguish contracts that must always hold (out of context strong assume/guarantee) for reuse, from contracts that hold in particular reuse contexts only (in context weak assume/guarantee).

Contracts may be formal or informal. Identification of assumptions is a challenging work, which requires a comprehensive approach for capturing and maintaining them [1] . Besides that, not all properties and requirements can be captured by formal contracts; hence we need to enable the formalism to support informal contracts as well. While formal approaches allow automatic operations on contracts such as checking of contract satisfaction and if refinement between contracts hold, informal contracts rely on human reasoning, e.g., for aspects that are too complex to be fully formalized.

With this formalism we aim at supporting the broadening of the scope of assumptions as well as reuse between different contexts (of rich components that include argument fragments) and taking into account informal contracts with human reasoning. In order to support this, we are proposing contracts that specify all the properties that an environment must satisfy separately from the guarantees and assumptions that are required to hold only in some contexts. The latter is specified as a set of weak assumption/guarantee pairs to

preserve the connection between assumptions and guarantees, and to enable specification of additional properties for specific alternative contexts. For example, consider a component *Sender* that provides the operation *send* and requires some other component to provide an operation *encrypt*. A contract for *Sender* could specify that under the strong assumption that *encrypt* always terminates; the strong guarantee is that *send* always terminates. Upon these strong assumptions and guarantees we can define some more context specific (weak) assumption/guarantee pairs:

1) Under the assumptions that *encrypt* always terminates within 10ms when GCC compiler is used on ARM platform, in return, the guarantee is that *send* always terminates within 30ms, and
2) Under the assumption that GCC compiler is used on ARM platform, the guarantee is that *Sender* requires no more than 5KiB of memory.

This idea translates into contract format where strong assumptions and guarantees (A and G) are defined as common for all the weak assumption/guarantee pairs (B and H):

$$< A, G, \{< B_1, H_1 >, \dots, < B_n, H_n >\}$$

This corresponds to the following traditional assumption/guarantee contract:

$$< A, (G \ \wedge (B_1 \rightarrow H_1) \wedge \dots \wedge (B_n \rightarrow H_n)) >$$

where our strong assumptions are becoming the only contract assumptions and guarantees are composed of conjunction of strong guarantees and weak assumption/guarantee pairs. Logically, this has the meaning:

$$A \rightarrow (G \ \wedge (B_1 \rightarrow H_1) \wedge \dots \wedge (B_n \rightarrow H_n))$$

If the assumption A holds then G follows (must hold), and for each weak assumption/guarantee pair B_i/H_i, the guarantee H_i follows (must hold) only if B_i holds. Note in particular that if B_i does not hold, then the implication holds regardless of the truth value of H_i. This provides a mean to specify guarantees that are required to hold only in certain contexts (i.e., H_i must hold in all contexts satisfying B_i).

The above expression means that G follows from A, and H_1 follows from A and B_1, etc. Logically, this has the meaning:

$$(A \rightarrow G) \wedge \left((A \ \wedge B_1) \rightarrow H_1\right) \wedge \dots \wedge ((A \wedge B_n) \rightarrow H_n)$$

but with additional distinction that the strong assumption A must hold, which is equivalent to:

$$(A \wedge G) \wedge \left((A \ \wedge B_1) \rightarrow H_1\right) \wedge \dots \wedge ((A \wedge B_n) \rightarrow H_n)$$

These contracts are associated to component types (components out-of-context). For a primitive component type we can check that the implementation satisfies the contract by ensuring that the code will deliver guaranteed functionality in all contexts satisfying the assumptions, i.e., that the contract implications hold.

In case of a composite component type, we can check consistency of its contracts and the contracts of the subcomponents in two separate steps: (1) by checking that all strong assumptions in a subcomponent that are not satisfied by the composition with other subcomponents are ensured by the strong assumptions in the

composite component contract, and (2) by checking that the composite component contract follows from the subcomponent contracts and the interconnections.

Basic component instance (in-context) contracts are inherited from the corresponding component type contracts. Component instance contracts refine inherited basic contracts based on information about the particular context.

Conversely, when an in-context contract is developed first and an out-of-context contract needs to be derived, an in-context contract can be manually "lifted" into an out-of-context contract.

3.2 About assumptions

Assumptions introduce some kind of uncertainty. Means must exist to verify that everything has been taken into account. The need also arises for a method which supports the specification of assumptions. As it has been mentioned before, development based on assumptions comes with an economical risk. It is not clear whether the assumptions are correct, consequently it is not clear whether the component can be used as it is.

Another main issue are hidden assumptions: often not all assumptions on the environment are explicitly described. It is obviously critically important to find effective ways to elicit these implicit assumptions. One possible idea is the use of simulation.

3.3 Using contracts in a model driven approach

The main added value achieved by adopting contract-based reasoning is if we can preserve guarantees at implementation and run time, to accurately formalize and validate expected runtime behavior. Currently there are few languages that already provide support to express contracts (e.g. SPARK Ada).

Traceability must exist, to guarantee that the results of the analyses performed at model level still hold at the implementation level…

Run-time contracts preservation would be required

3.4 Economics/business aspects of reuse

Metrics have to be defined to evaluate reusability in terms of cost, validation time…

It is important to keep in mind problems related to IPR: a totally different kind of problems (economical and legal for instance) that can hinder reuse.

4 Summary of research questions

- Is it possible to design configurable safety components for a predefined scope?
- Is it possible to fully validate configurable components with a predefined scope?
- Is it possible to support cross-domain reuse with this kind of configurable safety elements?
- What can be reused and on which level of abstraction/granularity?
- How can we measure or estimate the usefulness of reuse and how can we decide for which components reuse is economically beneficial?
- Is there a need for a dedicated process for reuse in a safety-critical context?
- How to be sure that all the assumptions about the environment are taken into account?
- Elicitation / validation of assumptions about physical environment: can we propose and validate a methodology?
- How can we validate that a method to specify assumptions is valid? (NOT a posteriori)

Reference

[1] M. Spiegel, P. F. Reynolds, Jr., and D. C. Brogan. A case study of model context for simulation composability and reusability. In Proceedings of the Winter Simulation Conference, pages 437- 444. ACM, 2005.

[2] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In Proceedings of the Software Technology Concentration on Formal Methods for Components and Objects (FMCO'07), volume 5382. Springer, October 2007.

[3] A. Benveniste, J.B. Raclet, B. Caillaud, D. Nickovic, R. Passerone, A. Sangiovanni-Vincentelli, T. Henzinger, and K. Larsen. Contracts for the design of embedded systems, Part II: Theory. Submitted for publication , 2012.

[4] T. P. Kelly. Arguing Safety - A Systematic Approach to Managing Safety Cases. PhD thesis, University of York, York, UK, Sept. 1998.

[5] I. Sljivo, J. Carlson, B. Gallina, and H. Hansson. Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts. In proceedings of the International Workshop on Critical Software Component Reusability and Certification across Domains (CSC2013), June 2013.