Fostering Reuse within Safety-critical Component-based Systems through Fine-grained Contracts

Irfan Sljivo, Jan Carlson, Barbara Gallina, and Hans Hansson
Mälardalen Real-Time Research Centre, Mälardalen University,
Västerås, Sweden
{irfan.sljivo,jan.carlson,barbara.gallina,hans.hansson}@mdh.se

Abstract

Our aim is to develop a notion of safety contracts and related reasoning that supports the reuse of software components in and across safety-critical systems, including support for certification related activities such as using the contract reasoning in safety argumentation.

In this paper we introduce a formalism for specifying assumption/guarantee contracts for components developed out of context. We are utilising the concepts of weak and strong assumptions and guarantees to customise fine-grained contracts for addressing a broader component context and specification of properties for specific alternative contexts. These out of context contracts can be conveniently instantiated to a specific context, thereby providing support for component reuse.

1 Introduction

Most standards for certification of safety-critical systems are formulated from the perspective of developing a new system from scratch. However, software is typically not developed completely from scratch. Instead, as much as possible is reused from previous projects, in order to reduce development time and take advantage of existing knowledge. This reuse can be in the form of architectural patterns, platforms, or code implementing common functionalities. There are also many methods that facilitate software reuse by making it more structured and systematic. For example, component-based software engineering is a method according to which software is developed by composing pre-existing or newly developed components, i.e., independent units of software, with a well-defined interface capturing communication and dependencies towards the rest of the system.

Our work addresses safety and certification of component-based system, i.e., systems developed using component-based reasoning that need to be certified according to a specific safety standard. Examples of such systems are found in domains such as automotive, railway or avionics.

Certification is (except for rare cases) performed with respect to a specific system, describing 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. The basic idea behind our work 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 characterised by a similar chain of events, its fault-tolerant behaviour 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 informa-

tion, we propose a contract-based approach, since this 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.

The main contribution of this paper is that it provides an adaptation of the assumption/guarantee contracts introduced by [3] to conveniently support reuse of components between products. We are specifically proposing a fine-grained assume/guarantee contract formalism that supports contract specification for out-of-context component types. The proposed approach assumes broader contexts and eases reuse in different contexts by allowing for the specification of mandatory properties and additional properties for specific alternative contexts. The broader contexts enables capturing of certification-relevant information that are needed for specification of safety contracts.

The rest of the paper is organised as follows: In Section 2 we present a short overview of key notions and related work we build upon. In Section 3 we present the key features of our contracts and define the proposed format and operations that can be performed on the contracts. Conclusion and future work are presented in Section 4.

2 Background and related work

In this section we briefly present some background information concerning safety-critical systems, component-based software engineering and contracts.

Safety-Critical Systems (SCS) are systems that may result in harm or loss of human life when they fail to perform their function [6]. In some SCS domains, e.g. automotive, railway or avionics, the systems must be certified according to a set of domain specific safety standards. 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 [5] is often required to show that the system is acceptably safe to operate. Building a safety case is based on human reasoning and expert judgement and is mainly manual or semi-automatic work.

The use of off-the-shelf (OTS) items in SCS has been debated for many years [8]. Some of the OTS items, as recognised by the standards, are *commercial off the shelf* items and *software of unknown pedigree*. While the first are developed by the standards, the latter are not developed to bespoke standards. The safety standards are providing more and more support for the use of OTS items and usually require evidence both that the product is safe and that the process used for development is as rigorous as the one mandated by a specific standard. These strict requirements limit the number of OTS items that can be reused, because of lack of evidence to prove that they are acceptably safe.

Component-Based Software Engineering (CBSE) aims at enabling rapid composition of a system from independently developed components by specifying contractual obligations that components must satisfy in order to achieve the system predictability [1]. One of the major goals of CBSE is to facilitate reuse of pre-developed components in order to reduce both time and cost of system development. A *component* is an independent unit of software, with a well-defined interface capturing communication and dependencies towards the rest of the system. A *component model* defines how components should be implemented and specified, as well as how components can be composed to form a system.

Applying CBSE for development of SCS poses particular challenges [4]. From the perspective of safety and certification, addressing these challenges is even more important when trying to apply CBSE notions within SCS, because of their strict safety demands.

Since simple interfaces are not expressive enough to capture the contractual obligations that specify dependencies towards the rest of the system, a notion of *contract* is introduced as a means for specifying and capturing functional and extra-functional

properties. In its basic form, contracts were introduced within the Eiffel programming language as a set of pre and post conditions between a caller and a method, and class invariants [7].

This idea is further extended in form of assumption/guarantee contracts with basic form C=(A, G), where A and G are assertions (assumptions and guarantees); a component makes assumptions regarding its context, required to hold for the guarantees to be provided [2]. The contracts are built around the notion of rich components that include contracts on different aspects (e.g. functional, timing, safety) of the component.

An extended form of A/G contracts is provided in [3], where strong and weak types of assumptions and guarantees are defined. The extended contract is defined as C=(A, B; G, H) where A and G are strong, and B and H are weak assumptions and guarantees. While the strong assumptions must hold in order for the contract to be satisfied, the weak assumptions are not required to hold. If they hold, then the weak guarantees are offered, otherwise the weak guarantees are disregarded.

3 Customising weak and strong reasoning with contracts

In our work we address certification of SCS developed using CBSE. In particular, we envision a CBSE 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).

Component contracts will be specified with A/G reasoning in mind and will make a clear distinction between strong and weak assumptions and guarantees. In the traditional A/G contracts, assumptions are used to express constraints on the environment of the component, i.e., on other 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 a 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.

We plan to handle the broader context by customising strong and weak assumptions and guarantees within contracts to be able to specify a number of weak A/G pairs on top of common strong assumptions and guarantees. In such contracts, the common strong assumptions clearly capture constraints that all compatible environments must fulfill for the contract to be satisfied, while the weak A/G pairs support specifying different constraints that are more environment specific.

We envision the contracts to be associated with component types. When instantiating the component type to a specific component instance, the component instance contract inherits all the weak and strong assumptions and guarantees, but typically only a subset of weak assumptions will be satisfied, depending on the context for which the type is being instantiated. This means that only the subset of weak guarantees that are implied by the satisfied weak assumptions will be guaranteed. Besides the inherited contract, additional context-specific contracts can be defined for the instances, but these are not of interest from a reuse perspective. When a new component is developed for a particular context and there is a potential for future reuse, then those parts of the information that hold regardless of context, or where context dependencies can be clearly defined, should be "lifted" from the component instance to the component type.

Identification of assumptions is challenging work and requires a comprehensive approach for capturing and maintaining them [9]. The formalism of specifying contracts needs to support both formal approaches that allow automatic operations on contracts, such as checking of contract satisfaction and if refinement between contracts hold, and plain text

contracts relying on human reasoning, e.g. for aspects that are too complex to be fully formalised.

In summary, the key features of the proposed approach are the following:

- The formalism supports both formal contracts and contracts defined in plain text, and is consistent with traditional A/G contracts.
- Both component types (corresponding to components in isolation) and component instances (corresponding to components used in a particular system) are supported. Component instances inherit contracts from the corresponding component type.
- The formalism supports specification of several alternative contexts through strong assumption and guarantee, plus a number of weak A/G pairs, where each assumption holds in some contexts and the corresponding guarantee is only required to hold in these contexts.

3.1 Proposed contract format

The motivation for distinguishing between the strong and weak assumptions and guarantees is mainly methodological [3]. In order to support the broadening of the context and reuse between different contexts, 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 A/G 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 a contract format where strong assumptions and guarantees (A and G) are defined as common for all the weak assumption/guarantee pairs (B and H):

$$\langle A, G, \{\langle B_1, H_1\rangle, \ldots, \langle B_n, H_n\rangle\}\rangle$$

This corresponds to the following traditional A/G contract:

$$\langle A, (G \wedge (B_1 \rightarrow H_1) \wedge \ldots \wedge (B_n \rightarrow H_n))\rangle$$

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

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

If the assumption A holds then G follows (must hold), and for each weak A/G-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:

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

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

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

3.2 Contract operations

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 (out-of-context), 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 assumption 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 (out-of-context) 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.

4 Conclusion

We have presented our proposition for using assumption/guarantee contracts to capture reusable certification-relevant information, using a CBSE approach to certification of SCS. In particular, the focus was on enabling usage of contracts in a broader context and a possibility to specify additional properties for specific alternative contexts in which a component can be used under certain assumptions. We are using the extended contract form with distinction between weak and strong assumptions and guarantees. We use strong assumptions to clearly define the least compatible environment in which the component can be used. Moreover, we use weak assumption/guarantee pairs to specify additional properties for specific alternative contexts, which hold depending on the context in which they are used.

In our future work we plan to fully develop the theory presented in this paper. Further on, we will work on determining certification-relevant properties that can be reused using our approach and establishing closer relation between A/G contracts and safety argumentation.

Acknowledgements

Thanks to Patrick Graydon for fruitful discussions and inspiration.

This work was supported by the Swedish Foundation for Strategic Research (SSF) project Synopsis.

References

[1] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, and F. Long. Technical Concepts of Component-Based Software Engineering, Volume 2. Software Engineering Institute, Carnegie Mellon University, 2000.

[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 Concertation 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] I. Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

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

[6] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[7] B. Meyer. Applying 'Design by Contract'. *IEEE Computer*, 25(10):40–51, Oct. 1992.

[8] F. Redmill. The COTS Debate in Perspective. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security*, SAFECOMP '01, pages 119–129, London, UK, 2001. Springer-Verlag.

[9] 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.