Configuration-aware Contracts

Irfan Sljivo, Barbara Gallina, Jan Carlson, and Hans Hansson

Mälardalen Real-Time Research Centre, Mälardalen University,
Västerås, Sweden
{irfan.sljivo, barbara.gallina, jan.carlson, hans.hansson}@mdh.se

Abstract. Assumption/guarantee contracts represent the basis for independent development of reusable components and their safety assurance within contract-based design. In the context of safety-critical systems, their use for reuse of safety assurance efforts has encountered some challenges: the need for evidence supporting the confidence in the contracts; and the challenge of context, where contracts need to impose different requirements on different systems.
In this paper we propose the notion of configuration-aware contracts to address the challenge contract-based design faces with multiple contexts. Since reusable components are often developed with a set of configuration parameters that need to be configured in each context, we extend the notion of contract to distinguish between the configuration parameters and the other variables. Moreover, we define a multi-context reusable component based on the configuration-aware contracts. Finally, we demonstrate the usefulness of the multi-context components on a motivating case.

1 Introduction

Software intensive safety-critical systems are nowadays rarely developed from scratch or by a single company. Instead, parts of the system are usually either reused or developed independently of the system [11]. To move towards components with pedigree and thus to fully benefit from reuse within safety-critical systems, the integrator companies need to reuse not only the components themselves, but also the accompanying safety artefacts [7]. The difficulty with reusing safety artefacts is that they are often context-specific. To enable out-of-context development, the notion of Safety Element out-of-Context (SEooC) was introduced within ISO26262 [5] as well as a corresponding life-cycle. This SEooC-related life-cycle requires that a set of context assumptions is identified for the reusable component and validated when the component is reused in the context of a particular system. These assumptions represent a way for the supplier of the reusable component to impose certain requirements on the usage of the provided component, in order to guarantee the specified behaviour of the component.

The term context is usually described as *any information that can be used to characterise the situation of an entity* [4]. In terms of SEooC, in-context is defined as all the information about the particular system, while out-of-context means that very little or no information is known about the environment in which

the component will execute. Moving a SEooC to a particular context means that we are gradually increasing the knowledge about the environment in which the component will execute until we gain the full knowledge about the environment.

Component contracts in software engineering represent a way to support independent development of reusable components by specifying behaviours of the components in assumption/guarantee pairs [1]. The guarantees state the behaviours of the component, provided that the environment behaves according to the assumptions. By supporting such contracts with evidence and relating them to the safety requirements allocated to the component, they can be used to semi-automatically generate assurance case argument-fragments [10]. A characteristic of the assumptions and guarantees is that they represent properties of entire traces, unlike assertions in program analysis that constrain the state space of a program at a particular point. In the traditional assumption/guarantee contracts [1] this means that the assumptions cannot include detailed information about different contexts, which are often contradictory. The distinction on strong and weak contracts allows for capturing those context-specific properties within the weak contracts [9], while the strong contracts capture properties of behaviours in all the different contexts. The weak contract assumptions are not required to be satisfied, while the strong contract assumptions, just as the traditional ones, are required to be satisfied.

Contract assumptions allow the developer of the component to impose requirements on the environment in which the component is used. But currently this imposing of requirements can be done only for all contexts, and since not all requirements are safety relevant in every context, an approach is needed to facilitate imposing requirements on only some contexts where they are actually safety-relevant. For example, an integrator company asks one of its suppliers for a reusable component and provides the specification on how this component should behave. These specifications should be addressed by the guarantees of the contracts of the reusable component. The supplier develops the component and in that process identifies assumptions under which the component exhibits the guaranteed behaviours. For such a component to be used in a particular system, all of the assumptions stated in the component contracts need to be satisfied by the system. The contract assumptions thus represent a way in which the supplier can constrain the set of environments in which the component can be used. Such reusable components are often developed with a set of usage profiles in mind that are characterised by different configuration parameters of the component. The different configuration parameters imply different behaviour of the component, and may require different constraints on different environments to guarantee the same safety requirement. Since the assumptions need to be consistent, otherwise no environment could fulfil them, the suppliers of reusable components are forced to weaken the assumptions of the contracts for the sake of specifying behaviour of the component under different configuration parameters. One way of achieving this weakening is by specifying the context-specific information in the weak contracts. But weak contracts do not impose any constraints on the component environment, since its assumptions are not required to be satisfied.

In this work, we extend the notion of contract to handle the multi-context setting of reusable components by explicitly distinguishing between assumptions on configuration parameters and operational variables. We introduce the configuration-aware contracts and demonstrate how they can be used to achieve similar flexibility as the strong and weak contracts, while providing the possibility to the supplier to explicitly impose requirements on only some contexts. Finally, we demonstrate the usefulness and the applicability of the configuration-aware contracts on a motivating case where a safety-relevant component is developed for reuse within a family of wheel-loaders.

The rest of the paper is structured as follows: In Section 2 we recall essential background information. We present the configuration-aware contracts and the related reasoning in Section 3. In Section 4 we present the application of the proposed extensions on a motivating case. We present related work in Section 5. Finally, we bring conclusions and future work in Section 6.

2 Background

In this section we recall the essential information on contracts. Moreover, we introduce the motivating case we use for illustrative purposes as well as demonstration of usefulness of the configuration-aware contracts.

2.1 The Assumption/Guarantee Contracts

The component model of the assumption/guarantee contract theory is based on a set V of variables, where each **variable** represents some relevant information about a component (e.g., input and output ports) [1]. A **contract** C is defined over the set of variables V as a pair $C = (A, G)$ of assertions, called the assumptions (A), and the guarantees (G).

We denote the **strong contracts** with (A, G) and the **weak contracts** with (B, H) [9]. The strong assumptions (A) need to be met by the environment of the component, and in return the component provides the guarantees (G). In contrast, the weak guarantees (B) are not necessarily offered in every environment.Only when both the strong and the weak assumptions are met, the corresponding weak guarantees(H) are offered. Such strong and weak contracts can be represented as traditional contracts by transforming the weak contracts into implications stated in the guarantees. The resulting contract is in conjuncted form where only the strong assumptions remain as the contract assumptions, while the guarantees represent the conjuncted strong guarantees and the implications from weak contracts, i.e., (A, G) and (B, H) can be represented as a single traditional contract $(A, G \wedge (B \Rightarrow H))$.

Just as the contracts, the environment E and implementation I are also defined in terms of the assertions over the set of variables V. When dealing with distinct sets of variables an **environment** is defined as a tuple E=(V_E,P_E) and **implementation** is defined as I=(V_I,P_I), where V_E and V_I are the sets of variables of the environment and implementation respectively, and P_E and

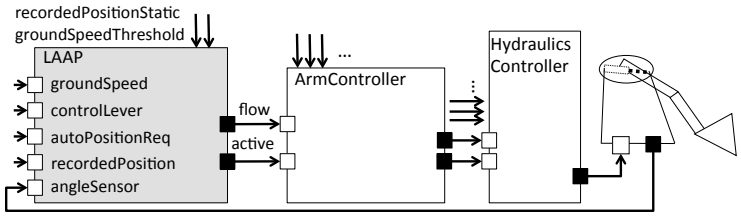

Fig. 1. The assumed structure of the lifting arm unit context

P_I are the sets of behaviours over the corresponding set of variables for the environments and implementations respectively. An implementation I is said to implement a contract $C = (A, G)$ if it provides the contract guarantees G, subject to the assumptions A, i.e., $P_I \cap A \subseteq G$. An environment E satisfies the contract C if it fulfils its assumptions, i.e., $P_E \subseteq A$. A **component** S is defined as a tuple $S = (V, C_S, I)$ where C_S is a set of contracts over the set of variables V, while I is a possibly empty set of implementations of the contracts [1].

2.2 Motivating Case

In this section we present the motivating case based on a real-world scenario where a single component Loading Arm Automatic Positioning (LAAP) is developed for reuse in different wheel-loaders.

Wheel-loaders are usually equipped with a loading arm, which can perform up and down movements. The software controller Loading Arm Controller Unit (LACU) handles both manual and automated arm movement. LACU calculates the arm movement commands based on the sensory data and user input, and then issues them to a hydraulic controller that moves the arm physically. The assumed structure of a representative LACU is shown in Fig. 1.

LACU is composed of two main subcomponents: the LAAP component that handles automatic arm positioning; and the Arm Controller component that issues the final command for both automatic and manual arm positioning. LAAP monitors the control lever that is used to lift/lower the arm manually and an automatic position request button that positions the arm in a predefined position. LAAP is activated by pressing the automatic positioning button, and it can be stopped by moving the control lever, as LAAP gets deactivated on detection of any movement of the control lever. When active, LAAP uses an arm angle sensor to determine the current arm position, while the target position is indicated by the recorded position port. The recorded position port is related to the *recordedPositionStatic* configuration parameter, which indicates whether the recorded position is predefined and constant, or if it can be set to a custom value. The ground speed port indicates the current ground speed of the vehicle and is used together with the *groundSpeedThreshold* configuration parameter such that the component deactivates if the current speed is greater than the specified threshold. Moreover, LAAP uses *maxGroundSpeed* that indicates the maximum ground speed of the vehicle to determine whether a faulty ground speed sensor can influence LAAP to move the arm when not supposed to.

3 The configuration-aware contracts

In this section we define the context and the configuration-aware contracts. Moreover, we define the reusable component for the multi-context based on the notion of the configuration-aware contracts. Finally, we discuss the differences between the configuration-aware contracts and the corresponding traditional contracts.

3.1 A component configuration context

As mentioned in Section 1, the context represents all information about a particular system. The fact that a component is developed out-of-context does not mean that no information about the system is known. Reusable components are usually developed with a set of usages in mind that are characterised by different configuration parameters of the component. We refer to the set of such parameters as the configuration context of the component. To define the configuration context, we first partition the set V of variables of the reusable component S: $V = V_{Sop} \uplus V_{Sconf}$; such that we distinguish between the operational (V_{Sop}) and the configuration (V_{Sconf}) variables. The distinction between the two types of variables is that the configuration variables, also referred to as parameters, can have only one value in a particular environment. The parameters allow component implementations to be prepared for use in different environments.

For each environment where the component may be used, there is a corresponding set of parameter values, which is why they can be viewed as constants when the component is used in a specific configuration context. We define a *configuration context* of a component as an assignment of a value to each variable from the set of configuration parameters V_{Sconf}. For example, *groundSpeedThreshold* and *recordedPositionStatic* in our motivating example are configuration variables that are constant in the context of a particular vehicle. Based on the values of these two parameters we can distinguish between different configuration contexts of the LAAP component. The recorded position can be fixed and the ground speed threshold set to zero, which represents the environment where likelihood of propagation of failures through LAAP is minimal. Another configuration context could be when the recorded position is dynamic and the ground speed threshold is at a higher vehicle speed e.g., 20 kilometres per hour. To reduce the likelihood of failures propagating through LAAP in environments that fit this context, requirements on the failure behaviours of the components providing recorded position and ground speed need to be imposed.

3.2 Configuration-aware contracts

As discussed in Section 1, using traditional contract assumptions to capture both the configuration and operational variables can lead to unwanted weakening of the assumptions. To overcome this problem we define the notion of *configuration-aware contract* to clearly distinguish between the assumptions on the configuration parameters and those over the operational variables. A *configuration-aware contract* is defined as a tuple $C = (A_c, A_o, G)$ where:

- A_c represent assumptions over the configuration variables, and is defined as an assertion over the set of configuration variables V_{Sconf};
- A_o represent assumptions over the operational variables, and is defined as an assertion over the set of operational variables V_{Sop};
- G represent the contract guarantees defined as an assertion over V.

The contract C states that the assertion A_o needs to be satisfied in all contexts satisfying A_c, and under these conditions G is guaranteed. While we model assertions over operational variables as sets of traces, assertions over the set of configuration variables can be simply modelled as sets of configuration contexts.

A correct implementation of a configuration-aware contract behaves according to the specified guarantees, provided that the corresponding assumptions on both types of variables are met. We define a configuration-aware *implementation* over the set of variables V as $I = (V, P_{Ic}, P_{Io})$ where:

- P_{Ic} represent a set of configuration contexts over the set of configuration variables V_{Sconf};
- P_{Io} represent an assertion over the set of operational variables V_{Sop}.

While an implementation considers the different values of the configuration parameters, an environment establishes a single configuration context, i.e., it considers only a single value for each configuration parameter. We define an environment over the set of variables V as $E = (V, p_{Ec}, P_{Eo})$ where:

- p_{Ec} represent the configuration context of the environment E over the set of configuration variables V_{Sconf};
- P_{Eo} represent an assertion over the set of operational variables V_{Sop}.

3.3 A multi-context component

As mentioned earlier, a reusable component can exhibit different behaviours in different configuration contexts. This makes the configuration context interesting for contract-based development because once a configuration context of a component is determined by a particular environment, then only behaviours of the component exhibited in that particular configuration need to be analysed. Hence, the configuration context information of an environment allow us to filter out the contracts relevant for the particular environment.

We define a multi-context component by considering its configuration contexts and the corresponding configuration-aware contracts. Formally, we define *multi-context component* as $S = (V, P_{Sc}, C_S, I_S)$ such that

- V is the set of variables composed of the sets of operational and configuration variables $V = V_{Sop} \uplus V_{Sconf}$;
- P_{Sc} is the set of configuration contexts over V_{Sconf};
- C_S is the set of configuration-aware contracts over V such that the set of configuration contexts of each of the contracts is a subset of P_{Sc};
- I_S is a possibly empty set of implementations over V.

As a multi-context component S is moved to the context of a particular system, it needs to be instantiated to an in-context component. We define an *in-context component* as a special case of a multi-context component where the set of configuration contexts P_{Sc} contains only one configuration context. Consequently, the set of contracts is reduced to only those matching the particular context. Given an environment $E_1 = (V, p_{E1c}, P_{E1})$ such that $p_{E1c} \in P_{Sc}$, we define an *instantiation* of an in-context component from the component S as $S_1 = (V, p_{S1c}, C_{S1}, I)$, where:

- $p_{S1c} = p_{E1c}$
- $C_{S1} = \{c \in C_S \mid c = (A_c, A_o, G) \wedge p_{S1c} \in A_c\}$

3.4 From configuration-aware contracts to traditional contracts

To transform a set of configuration-aware contracts to a traditional contract, we conjunct them such that the assumptions that need to hold for all configuration contexts are preserved, while other operational assumptions together with the assumptions on configuration variables are transferred to the traditional contract guarantees. The latter is done by implications in guarantees, expressing that if the transferred assumptions are satisfied they imply the corresponding configuration-aware contract guarantees. This is similar to how the strong and weak contracts are conjuncted into traditional contracts. This way the assumptions and guarantees transferred as implications behave as the weak contracts, while the assumptions and guarantees that hold in all configuration contexts behave as assumptions and guarantees of strong contracts.

The traditional conjuncted form does not distinguish between the different configuration contexts. While the two types of contracts are the same in terms of implementations, i.e., implementations of the conjuncted contract are the same as the implementations of the configuration-aware contracts, they differ in terms of environments. A correct environment of a contract in conjuncted form is every environment that satisfies only the overall contract assumptions, while the configuration-aware contracts of an in-context component offer the possibility for a more fine-grained constraining of the different environments without weakening the assumptions to only the assumptions that hold in all configuration contexts.

For example, if we consider *groundSpeedThreshold* configuration parameter in our motivating case from Section 2.2. LAAP is disabled when the ground speed value is greater than the threshold parameter. For contexts where the threshold value is lower than the maximum speed of the vehicle, the groundSpeed port failure can contribute to LAAP running when not supposed to, e.g., when groundSpeed is faulty and shows that the vehicle is moving slower than it actually is. For such contexts it is important to impose a requirement on the groundSpeed port value to be highly reliable. But when the threshold value is equal or greater than the maximum speed of the vehicle, then even if the groundSpeed port reports faulty value it cannot lead to LAAP running when not supposed to. Hence for the first set of contexts, it is important to impose the requirement on reliability of the groundSpeed port, while for the second case such requirement

is not necessary and in fact is too strong. For traditional contracts, this can be expressed only in the conjuncted form where no requirements on the environment would be made regarding the reliability of groundSpeed, or if such requirement would be made for all contexts, it would be too strong for certain cases.

The concept of configuration-aware contracts can be used to facilitate capturing optional behaviours in terms of weak contracts by defining a boolean parameter and using it only for the particular configuration-aware contract that is intended to be optional. This concept facilitates similar flexibility of the weak contracts, although it may result in a large number of configuration parameters for the different weak contracts.

4 Ilustrative case

In this section we demonstrate, using the motivating case introduced in Section 2.2, how capturing of the contracts as configuration-aware can influence the constraints imposed by the contract assumptions in a specific environment. Since the component is intended for a family of wheel-loaders, the requirement that needs to be satisfied in all the systems of the family is that the LAAP does not move the loading arm when not supposed to. Hence, the property that the LAAP outputs are not faulty needs to be satisfied in all the different systems. We model LAAP as a multi-context component and demonstrate how configuration-aware contracts can provide the mechanism for ensuring that the LAAP is not faulty in the different configuration contexts without making the assumptions too strong.

To compare the configuration-aware and traditional way of capturing the contracts, we consider how would capturing the same information using the two contract approaches influence the strength of the assumptions that a particular environment needs to fulfil. We first capture the configuration-aware contracts, shown in Table 1. The example shows five configuration-aware contracts where the first LAAP1 contract is valid for all the configuration contexts, since it checks whether the received values on the input ports are in the specified range. If not, then it disables the component outputs. The LAAP2-LAAP5 contracts are specific to the four different configuration contexts described in Section 2.2. The LAAP2 contract specifies the LAAP component behaviour when the maximum ground speed is greater than the ground speed threshold and the predefined arm position does not change. In this configuration context, for the component not to propagate any failures, the assumptions need to be made so that the ground speed value will not be faulty, while any faults of the recorded position do not influence the LAAP output. In contrast to this configuration context, when ground speed threshold is greater or equal to the maximum ground speed, as specified in the LAAP4 contract, the environment does not need to fulfil the requirement that the ground speed value is not faulty.

Since the traditional way of specifying contracts cannot capture the configuration variables in the contract assumptions, the assumptions need to be weakened to exclude the configuration variables. As described in Section 3.4, we transform the set of configuration-aware contracts to a traditional contract (Ta-

Table 1. A set of LAAP configuration-aware contracts

$\mathbf{A}_{c-LAAP1}$:	-;
$\mathbf{A}_{o-LAAP1}$:	*groundSpeed* within [0, 200] km/h AND *angleSensor* within [0,3] rad AND *controlLever* within ± 1 rad AND *recordedPosition* within [0,3] rad;
$\mathbf{G}_{LAAP1}$:	(*groundSpeed* not within [0, 200] km/h OR *angleSensor* not within [0,3] rad OR *controlLever* not 0 rad OR *recordedPosition* not within [0,3] rad;) **implies** (*Active* = false and *Flow* = 0);
$\mathbf{A}_{c-LAAP2}$:	*groundSpeedThreshold* < *maxGroundSpeed* AND *recordedPositionStatic*;
$\mathbf{A}_{o-LAAP2}$:	**not** *faultAngleSensor* AND **not** *faultAutoPositionReq* AND **not** *faultControlLever* AND **not** *faultGroundSpeed*;
$\mathbf{G}_{LAAP2}$:	**not** *faultFlow* AND **not** *faultActive*;
$\mathbf{A}_{c-LAAP3}$:	*groundSpeedThreshold* < *maxGroundSpeed* AND **not** *recordedPositionStatic*;
$\mathbf{A}_{o-LAAP3}$:	**not** *faultAngleSensor* AND **not** *faultRecordedPosition* AND **not** *faultAutoPositionReq* AND **not** *faultControlLever* AND **not** *faultGroundSpeed*;
$\mathbf{G}_{LAAP3}$:	**not** *faultFlow* AND **not** *faultActive*;
$\mathbf{A}_{c-LAAP4}$:	*groundSpeedThreshold* ⩾ *maxGroundSpeed* AND *recordedPositionStatic*;
$\mathbf{A}_{o-LAAP4}$:	**not** *faultAngleSensor* AND **not** *faultAutoPositionReq* AND **not** *faultControlLever*;
$\mathbf{G}_{LAAP4}$:	**not** *faultFlow* AND **not** *faultActive*;
$\mathbf{A}_{c-LAAP5}$:	*groundSpeedThreshold* ⩾ *maxGroundSpeed* AND **not** *recordedPositionStatic*;
$\mathbf{A}_{o-LAAP5}$:	**not** *faultAngleSensor* AND **not** *faultRecordedPosition* AND **not** *faultAutoPositionReq* AND **not** *faultControlLever* AND **not** *faultGroundSpeed*;
$\mathbf{G}_{LAAP5}$:	**not** *faultFlow* AND **not** *faultActive*;

Table 2. The corresponding LAAP traditional contract

$\mathbf{A}_{cf-LAAP}$:	$\mathbf{A}_{o-LAAP1}$ AND **not** *faultAngleSensor* AND **not** *faultAutoPositionReq* AND **not** *faultControlLever* ;
$\mathbf{G}_{cf-LAAP}$:	$\mathbf{G}_{LAAP1}$ AND (($\mathrm{A}_{c-LAAP1}$ AND **not** *faultGroundSpeed*) OR ($\mathrm{A}_{c-LAAP2}$ AND **not** *faultRecordedPosition* AND **not** *faultGroundSpeed*) OR ($\mathrm{A}_{c-LAAP3}$) OR ($\mathrm{A}_{c-LAAP4}$ AND **not** *faultRecordedPosition*)) **implies** (**not** *faultFlow* AND **not** *faultActive*));

ble 2) and also to the resulting in-context component overall contracts (Table 3). By comparing the assumptions of the traditional contract and the specific in-context overall contracts derived from the multi-context LAAP component, we notice that the assumptions for the different configuration contexts are in-general stronger than the assumptions of the traditional contract. This can for instance be seen on the in-context overall contract LAAP-C1 (Table 3), which besides the assumptions included in the traditional contract and the context-specific configuration parameters, also assumes that the ground speed value is not faulty. The strengthening of the in-context overall contract assumptions is done not only in terms of configuration parameters, but also in terms of assumptions over operational parameters. This way of deriving an in-context overall contract based on the configuration-aware contracts allows us to impose additional requirements in terms of assumptions that the corresponding environment of the particular in-context component needs to fulfil.

Table 3. The in-context LAAP overall contracts of the LAAP configuration contexts

$\mathbf{A}_{cf-LAAP-C1}$:	$\mathbf{A}_{o-LAAP1}$ AND **not** *faultAngleSensor* AND **not** *faultAutoPosition-Req* AND **not** *faultControlLever* AND **not** *faultGroundSpeed* AND $\mathrm{A}_{c-LAAP2}$;
$\mathbf{G}_{cf-LAAP-C1}$:	$\mathbf{G}_{LAAP1}$ AND **not** *faultFlow* AND **not** *faultActive*;
$\mathbf{A}_{cf-LAAP-C2}$:	$\mathbf{A}_{o-LAAP1}$ AND **not** *faultAngleSensor* AND **not** *faultAutoPosition-Req* AND **not** *faultControlLever* AND **not** *faultGroundSpeed* AND **not** *faultRecordedPosition* AND $\mathrm{A}_{c-LAAP3}$;
$\mathbf{G}_{cf-LAAP-C2}$:	$\mathbf{G}_{LAAP1}$ AND **not** *faultFlow* AND **not** *faultActive*;
$\mathbf{A}_{cf-LAAP-C3}$:	$\mathbf{A}_{o-LAAP1}$ AND **not** *faultAngleSensor* AND **not** *faultAutoPosition-Req* AND **not** *faultControlLever* AND $\mathrm{A}_{c-LAAP4}$;
$\mathbf{G}_{cf-LAAP-C3}$:	$\mathbf{G}_{LAAP1}$ AND **not** *faultFlow* AND **not** *faultActive*;
$\mathbf{A}_{cf-LAAP-C4}$:	$\mathbf{A}_{o-LAAP1}$ AND **not** *faultAngleSensor* AND **not** *faultAutoPosition-Req* AND **not** *faultControlLever* AND **not** *faultRecordedPosition* AND $\mathrm{A}_{c-LAAP5}$;
$\mathbf{G}_{cf-LAAP-C4}$:	$\mathbf{G}_{LAAP1}$ AND **not** *faultFlow* AND **not** *faultActive*;

5 Related Work

Contract-based design has emerged as an interesting approach that facilitates a range of activities such as independent development, requirements structuring, compositional verification, and safety assurance argument generation, all useful for the development of safety-critical systems. Westman et al. [12] generalises the established contract theory [1] to environment-centric contracts to provide support for practical engineering and expressing of safety requirements using contracts. The environment-centric contracts relax the constrains on the scope of the assumptions and guarantees beyond the interface of the corresponding component. While environment-centric contracts theory does not distinguish explicitly between the rigid variables such as configuration parameters and other operational variables, Cimatti et al. [2] present a tool-supported contracts-refinement proof system that distinguishes between the two types of variables. Although they can be separately specified, they are treated equally within the contract assumptions, and hence the explicit distinction does not alleviate the challenge contracts have with the different context.

Schneider et al. [8] introduce the Digital Dependability Identities (DDIs) as a way to assure dependability of cyber-physical systems. DDIs represent modular, composable and possibly executable specification. One of the main goals of DDIs is to provide the basis for run-time certification for the dynamically reconfigurable systems. Conditional Safety Certification (ConSert) represent an initial implementation of DDIs. The conditions in ConSerts are captured between the potentially guaranteed safety requirements (guarantees), and the corresponding demanded safety requirements (demands). In contrast, in our work we use contracts to capture the safety-relevant behaviours needed for satisfaction of safety requirements. Similarly to the conditions in ConSert, we extend the notion of contracts to act as conditions based on the configuration parameters and identify which component behaviours are relevant for a particular system. Since contracts

can be used for generation of argument-fragments [10], the configuration-aware contracts can be viewed as means to achieve conditional safety arguments offline. Although configuration-aware contracts have potential for run-time certification for reconfigurable systems, that work is out of the scope of this paper.

Oliveira et al. [3] present a method for automatic allocation of safety requirements to components of a Software Product-line (SPL) by building upon HiP-HOPS (Hierarchically Performed Hazard Origin & Propagation Studies) [6]. The proposed method enumerates the SPL products enriched with hazard and failure information, and then uses HiP-HOPS for automatic allocation of ASILs (Automotive Safety Integrity Levels). Based on the ASIL allocations for each of the products, the proposed method identifies the most stringent allocation for each of the SPL components across the entire product family. In contrast, the configuration-aware contracts of a component can be used to verify that the SPL products in which the component is reused meet the minimum needed requirements to ensure that the requirements allocated to the reusable component are met. By using configuration-aware contracts, we alleviate the need for all the neighbouring components of the reusable component to be allocated with the most stringent ASIL in all configuration contexts.

6 Conclusions and Future Work

Contract-based design is a promising approach to facilitate independent development of components and their safety assurance within safety-critical systems. One of the challenges it faces is the troublesome issue of context when dealing with safety requirements. While one requirement can be safety-relevant in the context of a particular system, it may not be relevant in the context of some other system. In this paper we have argued that there is a need for more fine-grained handling of the context within the contracts. We have proposed to clearly distinguish between assumptions on configuration parameters and other operational variables. Unlike the operational variables, the configuration parameters have constant values within a particular system, and this makes them a useful source of information when developing a reusable component for a set of different contexts. We have proposed extended configuration-aware contracts that use the configuration parameters to filter out the assumptions over the operational parameters for the different configuration contexts. We have demonstrated on a real-world example how the multi-context components enriched with configuration-aware contracts provide a mechanism for imposing requirements on only those environments where actually needed.

As our future work, we intend to align this work with product-line engineering for safety-critical systems. While product-line feature modelling is done at a higher level, the configuration-aware contracts allow for tailoring the safety behaviour and the corresponding safety assurance case. We plan to investigate the usefulness of configuration-aware contracts for systems where cloud-computing is used to provide service to safety functions. We also plan to explore how

configuration-aware contracts can be used to assist reuse of safety assurance artefacts across different system concerns such as safety and security.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research (SSF) via project SYNOPSIS and FIC, as well as EU and VINNOVA via the ECSEL Joint Undertaking projects AMASS (No 692474) and SAFECOP (No 692529).

References

1. A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen. Contracts for System Design. Research Report RR-8147, Inria, November 2012.
2. A. Cimatti and S. Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97(3):333–348, 2014.
3. A. L. de Oliveira, Y. Papadopoulos, L. Azevedo, D. Parker, R. Braga, P. C. Masiero, I. Habli, and T. Kelly. Automatic allocation of safety requirements to components of a software product line. *IFAC-PapersOnLine*, 48(21):1309 – 1314, 2015.
4. A. K. Dey. Understanding and Using Context. *Personal Ubiquitous Computing*, 5(1):4–7, 2001.
5. ISO 26262-10. *Road vehicles — Functional safety — Part 10: Guideline on ISO 26262*. International Organization for Standardization, 2011.
6. Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz, and R. Lien. Engineering Failure Analysis and Design Optimisation With HiP-HOPS. *Engineering Failure Analysis*, 18(2):590–608, 2011.
7. F. Redmill. The COTS Debate in Perspective. In *20th International Conference on Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 119–129, London, UK, 2001. Springer.
8. D. Schneider, M. Trapp, Y. Papadopoulos, E. Armengaud, M. Zeller, and K. Höfig. WAP: Digital dependability identities. In *26th International Symposium on Software Reliability Engineering*, pages 324–329. IEEE, 2015.
9. I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Strong and Weak Contract Formalism for Third-Party Component Reuse. In *3rd International Workshop on Software Certification, International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 359–364. IEEE, November 2013.
10. I. Sljivo, B. Gallina, J. Carlson, and H. Hansson. Generation of Safety Case Argument-Fragments from Safety Contracts. In *33rd International Conference on Computer Safety, Reliability, and Security*, volume 8666 of *Lecture Notes in Computer Science*, pages 170–185. Springer, September 2014.
11. J. Varnell-Sarjeant, A. A. Andrews, and A. Stefik. Comparing Reuse Strategies: An Empirical Evaluation of Developer Views. In *8th International Workshop on Quality Oriented Reuse of Software*, pages 498–503. IEEE, 2014.
12. J. Westman and M. Nyberg. Environment-Centric Contracts for Design of Cyber-Physical Systems. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014*, volume 8767 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2014.