The Nature and Content of Safety Contracts: Challenges and Suggestions For a Way Forward

Patrick Graydon[1]
[1]School of Innovation, Design, and Engineering
Mälardalen University
Box 883, 721 23 Västerås, Sweden
+46.21.10.14.21 (voice), +46.21.10.14.60 (fax)

Iain Bate[1,2]
[2]Department of Computer Science
University of York
Deramore Lane, Heslington, York, YO10 5GH, UK
+44.1904.325572 (voice), +44.1904.325599 (fax)

***Abstract*—Software engineering researchers have extensively explored the reuse of components at source-code level. Contracts explicitly describe component behaviour, reducing development risk by exposing potential incompatibilities early in the development process. But to benefit fully from reuse, developers of safety-critical systems must also reuse safety evidence. Full reuse would require both extending the existing notion of component contracts to cover safety properties and using these contracts in both component selection and system certification. This is not as simple as it first appears. Much of the review, analysis, and test evidence developers provide during certification is system-specific. This makes it difficult to define safety contracts that facilitate both selecting components to reuse and certifying systems. In this paper, we explore the definition and use of safety contracts, identify challenges to component-based software reuse safety-critical systems, present examples to illustrate several key difficulties, and discuss potential solutions to these problems.**

Keywords*—*Component-based software engineering, safety, contracts, safety arguments, modular safety case

Category: Regular Paper

I. Introduction

Software engineering researchers have extensively explored the reuse of components' implementations, usually at source-code level. Researchers have proposed leveraging component-based software engineering (CBSE) techniques such as assume-guarantee *contracts* to lower the cost of developing software for safety-critical systems [1], [2]. But the cost of generating and checking certification evidence dwarfs software implementation costs in such systems. Benefitting fully from reuse requires *incremental certification* that reuses as much safety evidence as practicable. Achieving incremental certification using CBSE would require both extending the existing notion of component contracts to cover safety properties and using these contracts in both component selection and system certification. This is not as simple as it first appears: many safety properties must play mutually-exclusive roles or are attested to by context-specific evidence.

In CBSE, developers describe software components using contracts [3]–[5]. Each contract describes a *guarantee* that can be made about a component's behaviour provided that an *assumption* is satisfied. For example, a fuel level estimator component might guarantee accurate output *if* sensors provide it accurate and timely input. These contracts facilitate both selecting components for a given application and reasoning about the behaviour of compositions of components [4]. To facilitate reuse, researchers participating in the SafeCer project [2] propose defining safety contracts to describe components' safety-related properties [5], [6]. These would help developers to address *development risk* by selecting components that have needed safety properties, avoiding surprises during later safety analysis. Safety contracts are also meant to aid *operational risk* assessment by documenting the properties each component claims to have. *Qualification* of components would attest that reusable safety evidence supports those claims. Unfortunately, the nature of some safety properties and the evidence supporting them makes it difficult to define contracts that are useful for both component selection and system certification.

In this paper, we build upon prior work [7] to explore the definition and use of safety contracts. We use a running example to illustrate subtle and surprising implications of that definition. Our contribution comprises (a) identifying challenges to using safety contracts to lower development cost and (b) proposing solutions to those problems.

In Section II, we define safety contracts and discuss how they might be used in system development and certification. In Section III, we introduce our specimen system. In Section IV, we illustrate how the role of safety contracts might change over the course of system development. In Section V, we discuss the contract content needed to support a component-based safety argument and discuss the issues that this content raises. Finally, in Section VI we present conclusions and recommendations for safety contract structure and content.

II. The Definition and Role of Safety Contracts

In this section, we present a vision for how safety contracts might be used in system design and incremental certification. While this vision comes from one particular project – SafeCer [2], [5], [6] – it represents a simple, obvious combination of existing ideas on software contracts, component-based software engineering, safety cases, and safety certification. It is useful to explore the implications of such a vision before proposing more complex approaches.

A. Components, Safety Evidence, and Contracts

In this work, we distinguish between *component types* and *component instances*: the former is the reusable, non-system-specific form of a component and the latter is a component as instantiated in a specific system [5]. This distinction helps

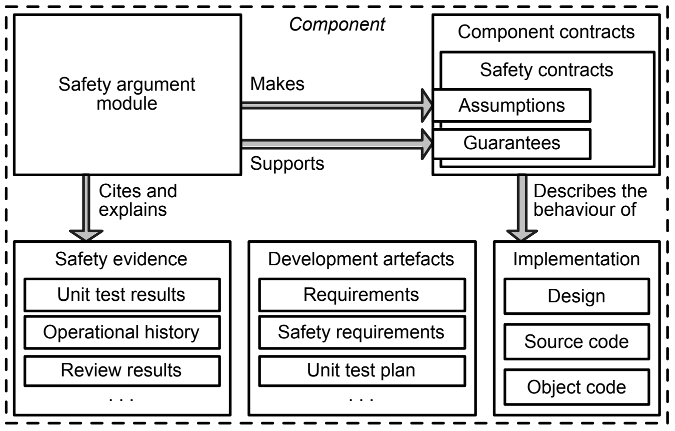


Fig. 1. Relationship between components, safety evidence, and contracts

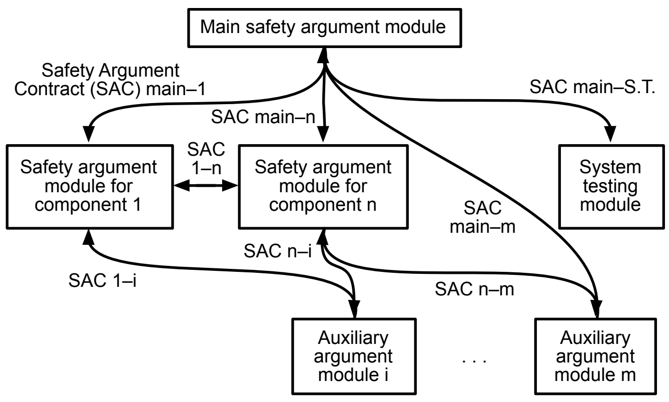

Main safety argument module
Safety Argument Contract (SAC) main–1
SAC main–n
SAC main–S.T.
Safety argument module for component 1
SAC 1–n
Safety argument module for component n
SAC main–m
System testing module
SAC n–i
SAC 1–i
SAC n–m
Auxiliary argument module i
. . .
Auxiliary argument module m

Fig. 2. Safety argument modules and contracts

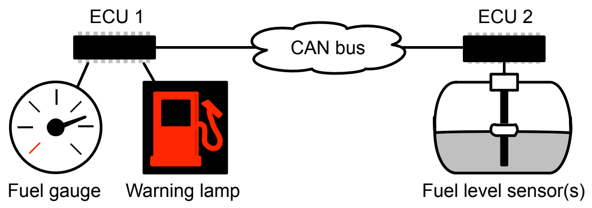


Fig. 3. Example Fuel Level Estimation System

to separate information that applies to multiple systems from information specific to a single system. A component type might be packaged and distributed as source code or as object code. A component instance comprises distinct portions of the system's design, the software source code, and the object code that will be deployed as part of a complete system.

Fig. 1 illustrates the relationship between safety contracts and components. While a component comprises design, source code, and object code, only the latter directly affects safety: design and source code flaws matter only insofar as they lead to object code flaws. Safety contracts must document the behaviour of the object code running on the target microprocessor. Source code and design properties are interesting only as (i) means of predicting components' behaviour during selection and (ii) indirect evidence of object code behaviour. Unfortunately, object code is rarely reused across systems. Thus, certification will typically require a combination of reused and system-specific safety evidence.

Component contracts describe the behaviour of each component (both as a type and as an instance) in terms of assumptions and guarantees. Researchers have given the following as examples of what contracts might specify [4], [8], [9]:

- If X is in [0, 10], then [output] Z is in [$-\infty$, 50]
- Output X is always less than the sum of inputs Y and Z
- Calls to a component that reads and writes files must follow the sequence (Open; (Read | Write)* ; Close)*
- The maximum dynamic memory usage is 100 bytes
- The WCET of the provided service A is 150 milliseconds
- Value errors on input port B do not affect output C

In the SafeCer component model, each component type is associated with zero or more safety and non-safety contracts [5]. When a component is instantiated in the context of a particular system, it implicitly inherits the contracts associated with its type. That is, the component type's contracts become the component instance's contracts. Safety and non-safety contracts might describe similar properties; the distinction is whether the guaranteed property is traceable to a hazard.

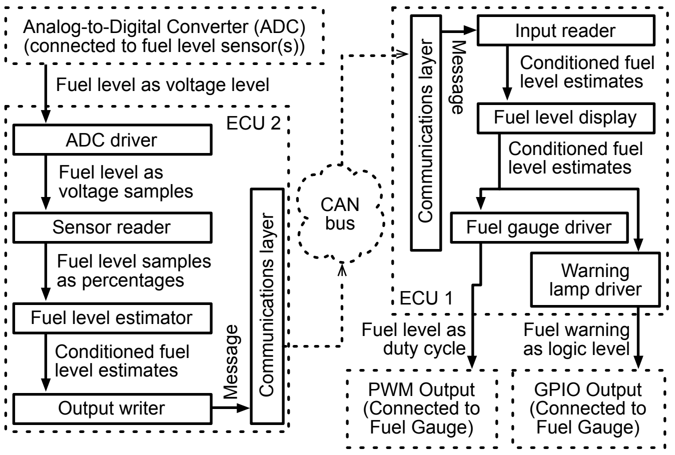
B. *Safety Contracts and Safety Arguments*

To explore the role of safety contracts in incremental certification, we examine how safety contracts could be used in a *safety case* [10], [11]. A safety case is 'a structured argument, supported by a body of evidence, that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment' [11]. Its purpose is to organise and explain the safety evidence [10]. In some domains, standards require the construction and evaluation of a safety case [11], [12]. However, even where developers write no safety argument, there is implicit logic linking safety evidence to certification and the decision to put a system into service [13], [14]. Investigating how safety contracts will be used in incremental certification reveals necessary properties.

To facilitate reusing evidence, each component should be associated with one or or more *safety argument modules* [5], [6], [10]. Fig. 2 illustrates this arrangement. Each such argument module explains how, given the assumptions in each component's safety contracts, evidence shows that it meets its guarantees. A main safety argument module presents the system's main safety claim and supports this with an argument over system hazards [15]. Hazard management claims would be broken down into claims about the properties of components. Those claims would be supported by the argument modules associated with those components. *Safety argument contracts* [10] (not safety contracts) show how each module's assumptions are justified (by other components' guarantees, the system context, etc.) so that the guarantees can be trusted.

III. An Example System: Fuel Level Estimation

We use a running example to make our discussion of safety contracts more concrete. Our illustrative system is inspired by a real system used to monitor fuel level in heavy road vehicles [16]. Loss of engine power in such vehicles can make them difficult to control, which would merit ASIL C according to ISO 26262's risk analysis scheme [12]. Fig. 3 illustrates



Analog-to-Digital Converter (ADC) (connected to fuel level sensor(s))
Fuel level as voltage level
ECU 2
ADC driver
Fuel level as voltage samples
Sensor reader
Fuel level samples as percentages
Fuel level estimator
Conditioned fuel level estimates
Output writer
Message
Communications layer
CAN bus
Communications layer
Message
Input reader
Conditioned fuel level estimates
Fuel level display
Conditioned fuel level estimates
Fuel gauge driver
Warning lamp driver
ECU 1
Fuel level as duty cycle
Fuel warning as logic level
PWM Output (Connected to Fuel Gauge)
GPIO Output (Connected to Fuel Gauge)

Fig. 4. Software Components of Example System

the major hardware components of the fuel level estimation system: (a) a fuel level sensor or sensors, (b) a microcontroller (ECU 2) connected to the sensors, (c) a microcontroller (ECU 1) connected to ECU 2 by a controller area network (CAN) bus, (d) a low-fuel warning indicator lamp, and (e) a fuel gauge. Fig. 4 illustrates the system's software components:

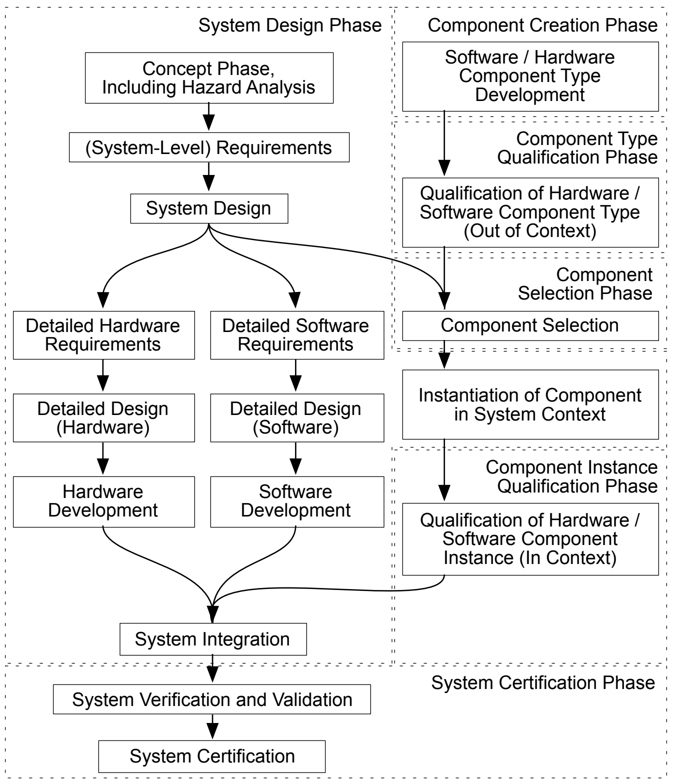
1) An ADC driver through which the system obtains digitised samples of the signal from the fuel level sensor(s) though the analog-to-digital converter (ADC)
2) A sensor reader component that periodically triggers the ADC and produces fuel level estimates as percentages
3) A fuel level estimator that filters fuel level samples to eliminate noise (e.g. from fuel sloshing in the tank)
4) An output writer that transmits fuel levels to other ECUs
5) The communications layer of the ECU platform
6) An input reader that receives estimates from ECU 2
7) A fuel level display component that periodically updates the fuel gauge and warning lamp
8) A fuel gauge driver through which the software drives pulse-width modulation hardware to drive the fuel gauge
9) A warning lamp driver through which the software controls power to the warning lamp

ECU 1 and ECU 2 also implement other functions. A real-time operating system sequences execution of components.

Variants of the system are used on several vehicles. Fuel tank size and shape vary. Vehicles use diesel, petrol, or liquified petroleum gas (LPG). LPG requires a different type of sensor than diesel or petrol. Microcontroller models vary with storage and speed needs and as manufacturers update offerings.

IV. THE LIFECYCLE OF A SAFETY CONTRACT

Fig. 5 illustrates a proposed incremental certification lifecycle for CBSE [7], [17]. This is simply a standard safety-related lifecycle (e.g. from ISO 26262 [12]) with additions for component *qualification*. Ideally, component types would be qualified so that assessors need check reused evidence only once. In the proposed lifecycle, safety contracts would fulfil a number of distinct roles and might undergo a related series of changes. In this section, we explore what safety contracts need to be to facilitate incremental certification.



System Design Phase
Component Creation Phase
Concept Phase, Including Hazard Analysis
Software / Hardware Component Type Development
(System-Level) Requirements
Component Type Qualification Phase
System Design
Qualification of Hardware / Software Component Type (Out of Context)
Component Selection Phase
Detailed Hardware Requirements
Detailed Software Requirements
Component Selection
Detailed Design (Hardware)
Detailed Design (Software)
Instantiation of Component in System Context
Hardware Development
Software Development
Component Instance Qualification Phase
Qualification of Hardware / Software Component Instance (In Context)
System Integration
System Certification Phase
System Verification and Validation
System Certification

Fig. 5. Phases of an incremental certification model for CBSE

Software must be maintained after a system is put into operation. Space constraints preclude fully exploring the role of safety contracts in maintenance (e.g. in component replacement) here. However, that typical safety practice uses change impact analysis to fake a rational process [18]; certification of a changed system might differ only in that unimpacted evidence need not be regenerated. We leave investigation of the role of contracts in the maintenance phase for future work.

A. A Component Type's Safety Requirements

Safety engineering is concerned with managing operational risk (i.e. the risk of harm to humans or the environment, as opposed to development risk). For example, developers of a vehicle and its systems must manage the risk of accident associated with loss of control of the vehicle. Typical safety processes (such as that defined by ISO 26262 [12]) require developers to (1) identify hazards, (2) analyse potential risk, (3) develop safety requirements that manage risk, and (4) allocate these requirements to system components. Developers must reason about how the behaviours of individual components could contribute to hazards. But 'hazard' (like safety) is irreducibly a system-level concept and component behaviour that is a hazardous contribution in one system may be of little consequence in another. For example, the fuel level estimator component in Fig. 4 has neither mass nor energy. Only when we consider (a) how the fuel level estimator's behaviour might misinform the driver about fuel level, (b) that a misinformed driver might run out of fuel, and (c) that loss of engine power makes some large vehicles uncontrollable do the safety implications of the this software component become clear. The lesson here is clear: it is imperative to evaluate the safety implications of components in their full system context.

To facilitate CBSE, developers of a component type might *assume* a set of safety requirements that *might* arise in some systems. Developers building a component for use in a product line might derive safety requirements from a product-line-oriented causal analysis such as product line fault tree analysis [19]. Developers building a general-purpose component might use a suitable safety analysis (e.g. a safety analysis for operating systems [20]) or simply ask potential customers what their applications require. In our example system, hazard and causal analyses of a defined product line or even of example systems selected to represent known variances would likely yield all component-level safety requirements of interest. But these techniques do not *guarantee* complete safety requirements. This observation reveals another lesson: component type safety contracts are *guesses* about what *might* be required.

After identifying safety requirements, developers would document them as contracts. Developers should endeavour to record all assumptions. However, neither formal analysis nor careful human enumeration can be guaranteed to reveal them all: it is impossible to prove that a formal model does not oversimplify the real world, and human software developers cannot reliably enumerate the assumptions their software relies on [21]. Failure to record and account for assumptions has caused accidents [22]. Another lesson is clear: developers instantiating a component cannot simply assume that if the stated assumptions are satisfied, everything is fine. Some researchers propose approaches based on reasoning about contracts as if the sets of assumptions are complete (e.g. [4]). But the assumption of completeness is unsupportable; in-context safety analysis (as standards dictate) is essential for safety assurance.

Developers instantiating the component in a system should perform a system-specific safety analysis and derive safety requirements for the component instance. Where (a) an assumed safety requirement satisfies a real derived requirement, (b) the documented assumptions are satisfied, and (c) safety analysts find no specific factors of the new context that undermine the safety evidence, developers can reuse the component type's safety case module in the complete system safety case. Where these conditions are not met, system developers will need to supply additional, system-specific, safety evidence.

B. *The Development Lifecycle*

Fig. 5 depicts the main phases of an incremental certification lifecycle for CBSE [6], [17]. During these phases, developers perform six component-related activities: (1) component creation, (2) component selection, (3) system design, (4) component type qualification, (5) component instance qualification, and (6) system certification.

1) Component Creation: Developers create component types to be instantiated either (i) in a particular system or systems or (ii) in other, unknown systems (i.e. out of context). A contract might represent the design requirements for the component. For example, developers building our example system might create contracts for its components and give these to sub-teams for detailed design, instantiation, and unit testing.

2) Component Selection: Developers building a new system might re-use appropriate component types selected from a library. Developers must use component types' contracts to predict how component instances might behave, then select components that are likely to provide needed functionality, have acceptable safety properties, and meet any other applicable requirements. For example, developers building our example system might search an in-house component library and select a sensor reader component to reuse. An appropriate component would have appropriate interfaces, be compatible with the vehicle's fuel level sensors, etc.

3) System Design: Developers must design a system around the components they have selected. During design, developers specify how components are connected, supply parameter values for each component (if any), and assign tasks to processors. Component instance contracts specify both known behaviours and design targets such as execution time budgets. The latter will evolve as engineers assess and refine the system design. For example, developers building our example system might supply parameters representing the fuel tank size, fuel tank geometry, and fuel level sensor gain to the sensor reader component. Developers might also specify code size, stack size, and static data budgets for each component.

4) Component Type Qualification: Qualification attests that a component type satisfies its safety contracts. Developers supply evidence showing that guarantees are met provided that assumptions hold; assessors confirm the evidence's adequacy. For example, reused components in our specimen system might come complete with test plans and qualification might attest that those test plans cover a specified set of requirements. If those requirements include the safety requirements assigned to the component instance, component type qualification might satisfy safety assurance obligations related to test coverage.

5) Component Instance Qualification: A component type contract holds only if the contract is used as directed. Component instance qualification confirms that the component type's assumptions are satisfied in a specific context, and thus that its guarantees hold in that context. For example, suppose that developers building our example system used a sensor reader component from an in-house component library. Component instance qualification would attest that developers had checked declared assumptions, systematically searched for violations of unstated assumptions, considered any new hazardous contributions the component might introduce, and found no unacceptable issues.

6) System Certification: Developers must show that the whole system meets its safety obligations. Component instance contracts represent derived safety requirements. Developers must use component qualification evidence and system-specific safety evidence to show that these requirements are met. For example, suppose that developers of our example system reused the sensor reader component at the source code level on a new model of microcontroller. Because the component was recompiled for a new target (even if its requirements and code did not change), developers must re-run its unit tests on the new microcontroller. This evidence, together with integration testing, software testing, and system testing evidence, would form part of the complete system safety case.

7) Extending Existing Systems: Section IV-B presents component type creation as though components are always built from scratch. In practice, developers will repurpose existing code to create components. That is, they will note a wider need for some functionality that they have already developed for a specific application, adapt that existing implementation for more general use, and package and deploy it as a component type. For example, developers might create a sensor reader component from code used successfully in prior products.

C. The Changing Role of Safety Contracts

The lifecycle described in Section IV-B makes a safety contract play four roles: (1) a means of encapsulating portions of the safety case; (2) a target for component type design; (3) a placeholder to facilitate system design; and (4) an indicator of expected performance. In this section, we discuss these roles and what it means to perform them adequately. Some roles are more crucial to safety than others. As we will show by example in later sections, creating contracts that perform all of these roles simultaneously is not straightforward.

1) A Means of Encapsulating Portions of the Safety Case: Perhaps the main role of a safety contract is that of encapsulating portions of the safety case. During system design, verification, validation, certification, and maintenance, encapsulation is critical to ensuring that the argument structure is both comprehensible and robust to expected changes. A 'good' set of safety contracts would address several challenges:

a) *Communicating clearly and accurately.* If component instance safety contracts are wrong or can be misinterpreted, assessors might certify a system as safe when it is not. For example, if a reasonable person reading the sensor reader component type's contract thought that it produced no output in the case of a hardware error when in fact it produced saturated-high output, safety analysis might miss important contributions to system hazards.
b) *Exposing the properties needed to argue safety.* If a component type contract does not guarantee a property that system safety relies on, system developers face extra work. For example, if the fuel level estimator component type's contract did not specify a response to input over 100%, developers instantiating that type in a system must specify this behaviour and provide appropriate evidence.
c) *Facilitating a compact, comprehensible argument.* Dividing the argument into component-related modules must not obscure the overall safety story. For example, engineers analysing contracts for separate speed and stability control systems in a car must not overlook the safety implications of feature interaction when the vehicle is driven on a slippery road surface. Moreover, since there is no known absolute scale for argument or evidence strength (see Section V-F), the argument must be compact enough to facilitate human understanding of how evidence quality affects confidence in safety claims.
d) *Being robust to change.* Contracts should encapsulate components' evidence and argument to facilitate substitution. For example, a sensor reader component type might use triple sensors. If its contract guarantees triple modular redundancy, designers cannot later substitute an ultra-reliable single sensor without revising the argument.

2) A Target or Placeholder for Design: Safety contracts play the role of both targets for component type design and placeholders for system design. Contracts record the detailed software requirements that system-specific components will later satisfy; these requirements serve as input to the Detailed Design (Software) phase shown in Fig. 5. For example, a safety contract for the fuel level estimator component in our sample system might specify (i) the filter function to be used, (ii) the maximum response time, (iii) how the component should react if an input sample is missing or out of range, and (iv) budgets for stack usage and the code and data segments. These might serve the component developer as a target, but system designers could also use this information to determine whether, *if* the instantiated components keep to their memory budgets, the software will fit into memory.

A safety contract serving as a design target or placeholder is subject to change. For example, developers might find a time or resource target impossible or unexpectedly difficult to meet. The developers of a speed control might discover the possibility of interaction with traction control through a safety analysis performed after both components have been designed, leading to a new requirement constraining speed control functionality when stability control is intervening. Because design placeholders change and depending on unstable contracts might lead to rework, developers must be able to distinguish stable contracts from design placeholders.

3) An Indicator of Expected Performance: The final role that safety contracts might play is as an indicator of expected performance. For example, suppose that contracts for the versions of the sensor reader component in the in-house component library specify execution time. Suppose further that this component is delivered as source code and compiled with other modules into an executable for the target system. The actual execution time will depend upon the target architecture, compiler, and compiler settings, and other factors. Any upper bound on execution time for the component *type* must be approximate, generous, or both.

However, developers selecting components for a new system could use data about execution times on well-known platforms to *approximate* the execution time of component instances in their system. Returning to our example, contracts for components in the library might specify execution time limits for combinations of compiler, compiler setting, microcontroller model, and microcontroller settings that have been used in earlier systems. While not useful for certification – developers instantiating the component in a new system would need new evidence establishing its worst case execution time – contracts expressing weak execution time limits could help to reduce the development risk of selecting a component that later testing would reveal to be too slow. The lesson here is that, to facilitate both component selection and safety arguing, we must be able to distinguish between indicative (low confidence) contracts and higher-integrity contracts.

V. Contracts Reflecting Safety Requirements

To fill the roles defined in Section IV-C, safety contracts must record assumed safety requirements, derived safety requirements, and anticipated safety requirements. To explore the issues involved in doing so, we first consider the sorts of component properties that safety arguments might depend on and then explore how contracts might express the properties that are relevant to that logic.

Consideration of the ways in which software is known to fail suggests that an enumeration of potential software contributions to system hazards would likely cover properties including: (i) nominal functional behaviour, (ii) resource usage, (iii) timing properties, (iv) platform assumptions, and (v) failure modes, propagation, and isolation. We examine each in turn, identifying issues related to specifying these properties in contracts and discussing the implications for safety-critical CBSE as researchers currently envisage it [5], [6].

A. Nominal Functional Behaviour

A component might contribute to a hazard by computing the wrong function. Both the identification of failure contributions and arguments about management of failure contributions will require reasoning about the functional behaviour of components. This might be expressed in terms of pre-conditions and post-conditions or valid operation sequences [5]. Functional behaviour might be described informally (e.g. using natural language text), formally (e.g. in Z [23] or Othello [9]), or a mix of the two.

1) Related Issues: Developers designing a component type will describe aspects of its behaviour that they think are important. When the component type is intended for a specific system, these designers will be aware of what is important and what is not important in that system's context. When a component type is designed for out-of-context use or used in a new context, the component type designer's notion of what aspects of behaviour are important might differ from those of the application developer. For example, the fact that a component temporarily suspends all interrupts might be utterly unremarkable in some contexts and important in others.

Some of the evidence supporting functional behaviour contracts is system-specific. For example, functional tests must use the chosen compiler and target computer if they are to reveal defects arising from compiler bugs or incorrect compiler settings. Even if a component's source code, safety requirements, and test plan do not change from platform to platform, developers must re-execute tests to confirm that the compiled object code makes the target behave as specified.

2) Implications: A component type's contracts might describe behaviour that is unremarkable in some systems but unsafe in others. While system-specific safety assessment (see Section IV-A) should reveal these, belated discovery that a component is unsuitable necessitates rework. To minimise the related development risk, component type developers should document characteristics that are plausibly important in some use scenarios even if they are not important in the context(s) in which the component type will be first instantiated.

When a component type will be used on multiple platforms, its functional behaviour contracts indicate what what instances are *expected* to do, not what reusable evidence *shows* that they do. The contract mechanism must make this distinction – and the need to re-run functional tests on the target hardware – clear to developers. One way to do this might be to associate each component with at least *two* argument

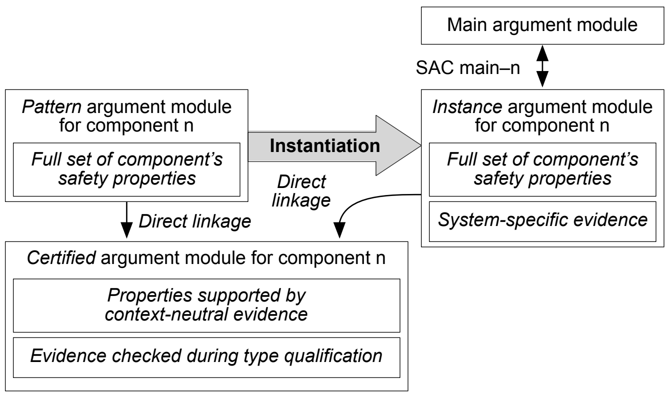

Main argument module
SAC main–n
Pattern argument module for component n
Full set of component's safety properties
Instantiation
Instance argument module for component n
Full set of component's safety properties
System-specific evidence
Direct linkage
Direct linkage
Certified argument module for component n
Properties supported by context-neutral evidence
Evidence checked during type qualification

Fig. 6. Multiple modules distinguish argument and evidence checked during component type certification from argument and evidence checked later

modules: (1) a *certified* argument module featuring only with properties checked during component type certification and (2) an *instance* argument module dealing with the complete set of component contracts. The latter would contain a safety argument *pattern* [10], with annotations indicating which pieces of evidence need to be supplied by the developer instantiating the component. For example, claims about test plan requirements coverage and structural coverage might be included in the certified argument module (and checked during component certification) because they not system-specific. The instance argument module might simply refer to the certified module for support of these properties. However, the behaviour claims themselves might appear *only* in the instance argument module, where they would be supported (in the context of the coverage claims) by testing evidence marked to indicate that developers should provide this evidence by executing the tests. Fig. 6 illustrates this approach.

Note that some test plan properties *are* application-specific. For example, boundary value coverage of sensor reader's test plan depends on the ADC's f_{max} and f_{min} properties.

3) Example Contracts: Nominal functional behaviour of component types and component instances can be described using typical contracts. For example, sensor reader might have the following contract:

Assume:	*Platform assumptions.* (See Section V-D.)
Guarantee:	Output fuel level f_o is related to the input ADC sample f_i, the sample value representing an empty tank f_{min}, and the sample representing a full tank f_{max} as specified by Equation 1.
Confidence:	Backing evidence will be appropriate for ASIL C. (See Section V-F.)
Status:	Design target (possibly subject to change).

$$f_o = max\left(0, min\left(100, \left\lfloor 100 \times \frac{f_i - f_{min}}{f_{max} - f_{min}}\right\rfloor\right)\right) \quad (1)$$

Sensor reader is distributed as source code. Because the target microcontroller and compiler will vary when the component is instantiated, the unit tests backing this contract must be rerun. The safety argument module associated with the sensor reader component type should be a *pattern* [10] with the unit test evidence marked as requiring instantiation. Developers instantiating the component would conduct the unit tests, remove the triangle decoration, and assess the completed argument. This logic depends on properties of the unit test plan, including requirements coverage. That coverage might have its own contract:

Assume:	None.
Guarantee:	Unit test plan TP-13A achieves coverage of requirements SUR-12 and SUR-16.
Confidence:	Backing evidence will be appropriate for ASIL C. (See Section V-F.)
Status:	Confirmed (unlikely to need higher SIL).

Requirements coverage evidence might come from inspections that need not be repeated when the component is recompiled for a different target. To benefit from qualification, the sensor reader component type's functional behaviour and test plan coverage contracts might be backed by evidence in separate arguments: one that is distributed as a pattern, and one that is delivered as a complete, verified argument.

B. Resource Usage

Identifying failure contributions and arguing about their management might require reasoning about components' resource usage. For example, developers might need to know limits on a component instance's usage of (a) program text, (static) data, and stack memory, (b) permanent storage (e.g. disk or flash memory), and (c) energy (if the system is power-constrained).

1) Related Issues: The amount of memory a component instance uses is not solely a function of its design and source code. Compiling the same source code with a different compiler, for a different microprocessor, or with a different optimisation setting might produce program text of a different length. A stack variable of type `int` in C might be as small as 16 bits but is often 32 bits and may be larger. Compilers pad data structures to comply with platform alignment requirements. Typical programming language rules do not forbid compilers to use arbitrary amounts of memory for purposes other than storing the variables, constants, and parameters specified in the source code. Thus, while we must know the memory consumed by a component instance for which we have object code, we cannot, in general, determine an upper bound for the amount of memory that a component type distributed as source code might consume when instantiated on an arbitrary platform.

2) Implications: Contracts for some resource usage properties might change during component instantiation. Contracts guaranteeing a component type's performance when instantiated under very strict conditions might serve to indicate component instances' expected performance (see Section IV-C3).

When an implementation is available, system designers could replace the estimated figure with a more considered one. If this contract is to serve to encapsulate portions of the system safety case (see Section IV-C1), developers would also supply appropriate evidence.

3) Example Contracts: The developers of component type *X* might specify its resource usage based on experience. For our example sensor reader component type:

Assume:	Sensor reader is compiled for a Freescale MPC 5554 using GCC 4.2 with no optimisation (`-O0`).
Guarantee:	Sensor reader uses no more than 4 KiB of the program text segment.
Confidence:	Informative only.
Status:	Confirmed (based on a completed product).

When sensor reader is instantiated, a related contract might serve as a placeholder to facilitate system design:

Assume:	Nothing.
Guarantee:	Sensor reader uses no more than 4 KiB of executable `ecu1fl.elf`'s text segment.
Confidence:	Informative only.
Status:	Design target (likely to change).

Note that memory usage evidence generally comes from tools that can analyse entire executables as easily as their components in isolation [24]. It makes more sense for safety cases to cite evidence of executables' memory usage than to argue over their components' memory usage. Developers should not specify high confidence for properties that will not be backed by evidence in the final, complete system safety case.

C. Timing Properties

Identifying failure contributions and arguing about their management might require reasoning about timing properties. A component instance might contribute to a system hazard if (i) its inputs are late, causing knock-on effects; (ii) its inputs that are samples of real-world values were taken too long ago, causing it to produce incorrect output; (iii) it delivers its output late; (iv) it consumes processor time that has been allocated to another component; (v) its execution is interrupted at the wrong time (e.g. during an operation that must be atomic); (vi) its execution causes or masks events that affect other components (e.g. disables interrupts); or (vii) the infrastructure does not execute it at the right time. Avoiding these problems will require defining and satisfying requirements such as [25]:

- Timing and timeliness requirements on inputs and outputs
- Requirements on when the component is executed (e.g. frequency, offset, and/or conditions)
- Lower bound on Best Case Execution Time (BCET)
- Upper bound on Worst Case Execution Time (WCET)
- Interrupt handling and/or masking behaviour
- Synchronisation requirements (e.g. assuming one core, using synchronisation primitives, or disabling interrupts

1) Related Issues: Software execution time might depend on properties that vary across the systems a component might be instantiated in. Compiling the same source code with a different compiler (even for the same platform) might result in object code differences that affect execution time. Optimisation settings can affect runtime. Even a component distributed as object code might run faster or slower in different contexts. For example, two applications might use processors that execute instructions in a functionally identical way but have different pipelines, branch prediction features, or caches. Identical processors might be configured with different clock rates or cache settings. Safety often requires knowing how long software will take to execute. Achieving target levels of processor utilisation often requires knowing this precisely [25]. Unfortunately, for the reasons given above, we cannot precisely know the execution time of a component type.

2) Implications: A wide range of factors affect execution time. It is unlikely that developers could specify assumptions so precisely and completely that the execution time evidence backing WCET guarantees would hold in any other system. As a result, component type WCET contracts should not claim high confidence even where the assumptions are satisfied.

As with memory usage, component type WCET contracts might be the basis for component instance contracts that serve as design placeholders. Given that WCET evidence is not typically reusable (for the reasons discussed above), it makes little sense use WCET contracts as intermediate claims in a safety argument about timing (see Section IV-C1). Moreover, if WCET evidence must be collected anew each time a component is instantiated, it might make more sense to gather evidence about threads' WCET than components' WCET.

3) Example Contract: Component type contracts for WCET might serve to indicate component instances' expected performance. These would follow the pattern for resource usage given in Section V-B3. For our example sensor reader component type:

Assume:	Sensor reader is compiled using GCC 4.2 with no optimisation (`-O0`), executed on a Freescale MPC 5554 at 200±4 megahertz with caches disabled, and not interrupted.
Guarantee:	The WCET of sensor reader's `UpdateFL` task is not more than 1 millisecond.
Confidence:	Informative only.
Status:	Design target (subject to change).

Suppose that a developer is building a new system and wishes to use this component. However, the new system will use a newer but similar model microcontroller. The developer would have to infer from existing, informative contracts whether the existing component's performance is likely to be good enough in the new context. Developers who judge the *development risk* of the component turning out to be too slow in practice to be high could address this risk through a risk reduction activity such as timing a prototype instantiation compiled for the new target.

D. Platform Assumptions

Any guarantee that a component makes about behaviour and timing will depend upon assumptions about the platform. For example, assumptions about instruction set architecture might underpin a WCET guarantee. Assumptions about instruction set architecture might even underpin functionality guarantees if the component's source code includes inline assembly code. Reasoning about safety argument module contracts might require information about:

- Processor type (e.g. instruction set and number of cores)
- Processor configuration (e.g. clock rate, cache enable, and cache lines locked)
- Other hardware, its configuration, and its performance (e.g. the existence of an digital to analog converter, its hardware address, and its conversion rate)
- Operating system configuration or services

1) Related Issues: Some platform assumptions will underpin many, most, or all contracts for a given component. For example, the component type contracts illustrated in Section V-B2 and Section V-C2 both specify a compiler and compiler settings. Our description is necessarily brief, but a full specification of relevant settings might be lengthy. Including such long detailed assumptions could bloat contracts, making them difficult for humans to read and understand.

2) Implications: A shorthand mechanism for denoting a set of assumptions might make contracts more readable. This would allow a contract to signify assuming several related properties using one convenient identifier.

3) Example Assumptions: Developers might specify platform assumptions as part of a statement of a component type's intended applicability. This specification might be as simple as a list of assumptions. For example, the sensor reader component's assumptions might include:

- PA1: A float-type fuel-level sensor is connected to an 8+-bit, 10+ kilohertz ADC that can be sampled through a component exporting the `ADCSampler` interface.
- PA2: The target microcontroller is a uniprocessor.
- PA3: No interrupt handler will alter memory in the portions of the text segment, data segment, and stack associated with sensor reader.

E. Failure Modes, Propagation, and Isolation

Identifying failure contributions and identifying system hazards will depend upon knowing both (1) how each component might fail and (2) how each component might propagate failures. A description might cover:

- Means of preventing, blocking, or tolerating interference (e.g. memory overwrites) [26], [27]
- A definition of 'abnormal' input data and a description of the component's reaction to it
- A definition of potential abnormal outputs and a description of the cases under which they might occur
- A description of anticipated hardware failures and the component's reaction to these

1) Related Issues: All analyses of how a component might contribute to a system hazard are system-specific: 'hazard' is defined in terms of the system's impact on the world. However, some behaviours (e.g. memory overwrites) are considered failures and would contribute to hazards in nearly all systems (barring those in which software is wholly irrelevant to safety). Component type contracts might cover these behaviours.

2) Implications: Evidence that all relevant failure contributions have been included must be system-specific. It would be helpful if the analysis providing this could leverage existing knowledge about specific problems to lower analysis cost.

3) Example Contracts: Means of preventing interference might be documented as contracts. For example, our sensor reader component might have the following contract:

Assume:	Nothing.
Guarantee:	Sensor reader will write to the stack only as its interface and the ABI dictate.
Confidence:	Backing evidence will be appropriate for ASIL C. (See Section V-F.)
Status:	Confirmed (generally necessary).

Evidence that source code has been mechanically verified to conform to the SPARK language rules [28], [29] might show that it does not violate memory constraints. Failure modes and propagation could be described using an existing notation such as the Failure Propagation and Transformation Notation (FPTN) [30] or the Architecture Analysis and Design Language (AADL) Error Model Annex [31].

F. *Confidence*

Some of the properties above, like WCET, cannot be established beyond doubt [25]. As a result, (1) contracts must specify the confidence with which components guarantee these properties, (2) components' safety evidence must justify the specified confidence, and (3) system developers must confirm that the specified confidence is sufficient given the potential consequence of not satisfying the guarantee. Researchers have proposed and practitioners use several different means of communicating confidence [32]:

- *Safety Integrity Levels (SILs).* Many standards define SILs and use these (roughly) as a metric for confidence: developers determine the potential impact of a failure, define appropriate SIL levels for system elements, and supply the forms of safety evidence dictated by SIL level.
- *Distributions of Failure Rates.* Some researchers propose modelling confidence in safety properties using statistical distributions of dangerous failure rates [32], [33].
- *Bayesian Belief Levels.* Researchers have proposed using Bayesian Belief Networks (BBNs) to model and reason about uncertainty in safety claims [32], [34]–[38].
- *Baconian Probabilities.* Other researchers favour modelling uncertainty using Baconian (as opposed to Pascalian) probability [32], [39]–[41].

1) Related Issues: Unfortunately, none of these is a perfect solution [32]. SIL-based approaches have several drawbacks. First, there is no evidential basis for the logic that two SIL x components make a SIL $x + 1$ component [32]. (One might use a joint failure distribution to make a claim about failure rates, but that is different.) Second, because the meaning of 'safety' changes from system to system and standards typically allow developers and assessors to agree arbitrary means of compliance, it is unclear from a SIL x designation what has been done and what confidence can be placed in a reused component [42]. Third, because standards define SILs, safety processes, and terms such as 'safety' and 'fault' differently, frustrating cross-domain reuse [43].

Because it is generally impractical to determine the distribution of failure rates for highly reliable software [44], [45], distributions of failure rates and BBNs based on them generally rely on expert opinion [32], [46]. While numbers give the impression of precision, reliance on dubious data creates a danger of producing "superficially plausible nonsense" [46].

Confidence concepts based on Baconian probability can help to identify threats to confidence [39], [41]. Some researchers have suggested specifying confidence in terms of the number of threats to a claim addressed [40], [41]. However, knowing that three of four identified threats have been addressed is only a measure of confidence if we know that there are no other threats [32].

2) Implications: We cannot avoid specifying confidence and no existing means of doing so is well-suited to all reuse scenarios. Given the familiarity of the SIL concept, many developers will want to specify confidence using SILs. Others will want to use component qualification evidence to help show conformance to a standard that uses SILs. To facilitate this use and help developers to avoid the associated pitfalls, safety experts might (1) define a domain-neutral SIL scale in a standard for component qualification and (2) publish guidance on accepting such qualifications as evidence of conformance to the relevant objectives of popular standards.

3) Pattern and Example: In Section V-A3, we used ISO 26262 ASIL levels [12] as a rough indicator of the quality of evidence underpinning functional behaviour contracts. In our example system, which is meant to conform to ISO 26262, ASIL might make a flawed but serviceable confidence metric: developers in one organisation, following one standard as reified in their in-house policy manual, will tend to use the same kind of evidence for each type of property at each ASIL.

VI. Conclusions

Researchers propose leveraging Component-Based Software Engineering techniques to lower the cost of developing safety-critical software systems [2]. In this paper, we have explored the roles that assume–guarantee contracts would play in such reuse and the requirements such contracts must satisfy. In doing so, we have revealed several insights about such contracts that have not been documented previously:

- Only in a specific system's context can we say which of a component's contracts are safety related. Safety engineers distinguish safety requirements from others to treat them more rigorously. We propose annotating contracts with a confidence specification to facilitate that distinction.
- Functional test evidence must be regenerated when components are recompiled. We propose associating components with both pattern and certified argument modules. Test plan properties could be specified in the former and qualified with the component type. The pattern module could show developers how to cite test results after they have run the tests on the target.
- There are properties such as memory usage and execution time whose role and meaning change over the development lifecycle. Component type contracts specifying these can aid component selection but cannot be guaranteed to the degree required for system certification. These distinct roles must be highlighted to avoid confusion. We propose associating each contract with confidence and status fields that indicate how much it can be trusted at any time.
- Properties such as memory usage and execution time are often the subject of speculation during early design; developers make guesses and then revise these as concrete data becomes available. It must be clear to developers which contracts are stable and which might be revised. We propose a status property of contracts to clarify this.
- Some assumptions (e.g. about the platform) might underpin several guarantees. We propose a shorthand notation for common assumptions to make them more readable.
- Because safety is a system property, developers must assess reused components' contributions to each system's hazards. We propose that the component instance qualification process include system-specific hazard and causal

analysis that take existing contracts' insights into account where doing so speeds the process.

- While many crucial safety properties cannot be 'proven', no existing means of specifying confidence is ideally suited to CBSE for safety-critical software. We propose further research into reliably communicating confidence.

Acknowledgment

The work was funded by the Swedish Foundation for Strategic Research (SSF) as part of the SYNOPSIS project and EU's Artemis funded SafeCer projects.

References

[1] J. L. Fenn, R. D. Hawkins, P. J. W. Williams, T. P. Kelly, M. G. Banner, and Y. Oakshott, "The who, where, how, why and when of modular and incremental certification," in *Proc. IET Int'l Conf. on Sys. Safety*, 2007.

[2] SafeCer. (2013, June) Safety certification of software-intensive systems with reusable components. [Online]. Available: http://www.safecer.eu

[3] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[4] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *Proc. EUROMICRO Conf. on Software Engineering and Advanced Applications*, 2012, pp. 21–28.

[5] J. Carlson, C. Ekelin, J.-L. Gilbert, Á. Herranz, and S. Puri, "Generic component meta-model, v. 1.0," SafeCer, Report D2.2.4, 2012.

[6] P. Graydon, I. Bate, L.-O. Berntsson, O. Bridal, J. Carlson, R. Land, A. Leitner, H. Martin, and B. Winkler, "Nature and derivation of safety contracts, v. 1.1," SafeCer, Report D2.3.2, 2013.

[7] P. Graydon and I. Bate, "Thoughts on the nature and content of safety contracts," in *Proc. Int'l Symp. on High Assurance Systems Engineering (HASE)*, Miami, FL, USA, January 2014, fast abstract paper.

[8] P. Böhm, J. Carlson, P. Conmy, C. Ekelin, J.-L. Gilbert, T. Gruber, Á. Herranz, and A. Martelli, "Specification of the requirements on the generic component model, including certification properties and safety contracts," SafeCer, Report D2.2.1 & D2.2.2, 2012.

[9] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta, "Validation of requirements for hybrid systems: A formal approach," *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 2012, to appear.

[10] K. Attwood *et al.*, *GSN Community Standard Version 1*. York, UK: Origin Consulting Limited, November 2011.

[11] Def. Stan. 00-56, *Safety Management Requirements for Defence Systems, Issue 4*. United Kingdom: Ministry of Defence, June 2007.

[12] ISO 26262:2011, *Road Vehicles — Functional Safety*. International Organization for Standardization, 2011.

[13] P. J. Graydon and T. P. Kelly, "Using argumentation to evaluate software assurance standards," *Information and Software Technology*, vol. 55, no. 9, pp. 1551–1562, 2013.

[14] C. M. Holloway, "Making the implicit explicit: Towards and assurance case for DO-178C," in *Proc. Int'l Sys. Safety Conf. (ISSC)*, 2013.

[15] P. Conmy and I. Bate, "Assuring safety for component based software engineering," in *Proc. Int'l Symp. on High Assurance Sys. Eng. (HASE)*, 2014, pp. 121–128.

[16] O. T. Jaradat, P. Graydon, and I. J. Bate, "The role of architectural model checking in conducting preliminary safety assessment," in *Proc. In'l Sys. Safety Conf. (ISSC)*, 2013.

[17] S. Björnander *et al.*, "A generic process model for integrated certification and development of component-based systems, v. 1.1," SafeCer, Report D2.1.1, 2011.

[18] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Trans. on Software Eng.*, vol. 12, no. 2, pp. 251–257, February 1986.

[19] J. Dehlinger and R. R. Lutz, "Software fault tree analysis for product lines," in *Proc. Int'l Symp. on High Assurance Sys. Eng. (HASE)*, 2004.

[20] P. M. Conmy, "Safety analysis of computer resource management software," Ph.D. dissertation, University of York, York, UK, 2006.

[21] M. Spiegel, P. F. Reynolds, Jr., and D. C. Brogan, "A case study of model context for simulation composability and reusability," in *Proc. Winter Simulation Conf.*, 2005.

[22] Ariane 501 Inquiry Board, *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*, Paris, France, July 1996.

[23] J. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall, 2001.

[24] AbsInt, "StackAnalyzer: Stack usage analysis," Web page: http://www.absint.com/stackanalyzer/index.htm, last checked 24 March 2012.

[25] P. Graydon and I. Bate, "Realistic safety cases for the timing of systems," *The Computer Journal*, 2013, in press.

[26] P. J. Graydon and T. P. Kelly, "Assessing software interference management when modifying safety-related software," in *Proc. Int'l Conference on Computer Safety, Reliability, and Security (SAFECOMP) Workshops*. Springer, September 2012, pp. 132–145.

[27] J. Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," Langley Research Center, Technical report NASA/CR-1999/209347, 2000.

[28] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[29] AdaCore, "SPARK Pro," Web page: http://www.adacore.com/sparkpro/.

[30] P. Fenelon, J. A. McDermid, M. Nicholson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *SIGAPP Applied Computing Review*, vol. 2, no. 1, pp. 21–32, March 1994.

[31] SAE AS5506/1, *SAE Architecture Analysis and Design Language (AADL) Annex*. SAE, 2006, vol. 1.

[32] P. J. Graydon, "Uncertainty and confidence in safety logic," in *Proc. Int'l Sys. Safety Conf. (ISSC)*, 2013.

[33] R. E. Bloomfield, B. Littlewood, and D. Wright, "Confidence: Its role in dependability cases for risk assessment," in *Proc. Int'l Conf. on Dependable Sys. and Networks (DSN)*, 2007, pp. 338–346.

[34] A. Ayoub, J. Chang, O. Sokolsky, and I. Lee, "Assessing the overall sufficiency of safety arguments," in *Proc. Safety-Critical Sys. Symp. (SSS)*, 2013.

[35] E. Denney, G. Pai, and I. Habli, "Towards measurement of confidence in safety cases," in *Proc. Int'l Symp. on Empirical Software Engineering and Measurement (ESEM)*, 2011.

[36] B. Littlewood and D. Wright, "The use of multi-legged arguments to increase confidence in safety claims for software-based systems: a study based on a BBN analysis of an idealised example," *IEEE Trans. on Software Eng.*, vol. 33, no. 5, pp. 347–365, May 2007.

[37] W. Wu and T. Kelly, "Combining Bayesian Belief Networks and the Goal Structuring Notation to support architectural reasoning about safety," in *Proc. Int'l Conf. on Computer Safety, Reliability, and Security (SAFECOMP)*, 2007, pp. 172–186.

[38] X. Zhao, D. Zhang, M. Lu, and F. Zeng, "A new approach to assessment of confidence in assurance cases," in *Proc. Int'l Conf. on Computer Safety, Reliability, and Security (SAFECOMP)*, 2012.

[39] R. Hawkins, T. Kelly, J. Knight, and P. Graydon, "A new approach to creating clear safety arguments," in *Proc. Safety-Critical Sys. Symp. (SSS)*, 2011, pp. 3–23.

[40] J. A. McDermid, "Risk, uncertainty and software safety," in *Proc. Int'l Systems Safety Conf. (ISSC)*, 2008.

[41] C. B. Weinstock, J. B. Goodenough, and A. Z. Klein, "Measuring assurance case confidence using Baconian probabilities," in *Proc. Wkshp on Assurance Cases for Software-Intensive Sys. (ASSURE)*, 2013.

[42] F. Redmill, "Safety integrity levels — theory and problems," in *Proc. Safety-critical Sys. Symp. (SSS)*, 2000.

[43] V. Manni *et al.*, "Baseline for the common certification language," Open Platform for EvolutioNary Certification Of Safety-critical Systems (OPENCOSS), Report D4.1, 2012.

[44] R. W. Butler and G. B. Finelli, "The infeasibility of experimental quantification of life-critical software reliability," in *IEEE Trans. on Software Eng.*, 1991, pp. 66–76.

[45] B. Littlewood and L. Strigini, "Validation of ultrahigh dependability for software-based systems," *Comm. ACM*, vol. 36, no. 11, pp. 69–80, 1993.

[46] B. Littlewood, "Dependability assessment of software-based systems: State of the art," in *Proc. Int'l Conf. on Software Eng. (ICSE)*, 2005, pp. 6–7, invited talk.