Using Contract-based Component Specifications for Virtual Integration Testing and Architecture Design

Werner Damm*, Hardi Hungar*, Bernhard Josko*, Thomas Peikenkamp*, Ingo Stierand†

*OFFIS, Germany
{werner.damm|...|thomas.peikenkamp}@offis.de

†University of Oldenburg, Germany
stierand@informatik.uni-oldenburg.de

***Abstract*—We elaborate on the theoretical foundation and practical application of the contract-based specification method originally developed in the Integrated Project SPEEDS [11], [9] for two key use cases in embedded systems design. We demonstrate how formal contract-based component specifications for functional, safety, and real-time aspects of components can be expressed using the pattern-based requirement specification language RSL developed in the Artemis Project CESAR, and develop a formal approach for virtual integration testing of composed systems based on such contract-specifications of subsystems. We then present a methodology for multi-criteria architecture evaluation developed in the German Innovation Alliance SPES on Embedded Systems.**

I. Introduction

The lack of an overall understanding of the interplay of subsystems and the difficulties encountered in integrating within one system subsystems from multiple engineering domains (mechanical, hydraulic, electronic) and from multiple organizations (such as from different suppliers) cause system integration to become a nightmare in the system industry, as demonstrated by Table I[1].

Tackling this system integration challenge requires multiple lines of attack, ranging from process issues to methods focusing on giving precise interface specifications and tracing their interdependencies across the complete design space, covering all engineering domains and all abstraction layers. While this paper focuses on capitalizing on the value of precise interface specifications in multi-criteria architecture design and virtual integration testing, it is part of an overall systems-engineering approach currently pushed by the Artemis Flagship Project CESAR[2], building on a common meta-model for multiple-domain system engineering artifacts and process improvements supporting dependency management of contract-based specifications of these. It builds on substantial results of the Integrated Project SPEEDS[3] and the Strep Project COMBEST[4], all focusing on *contract-based embedded systems development*, see [6], [7], [8], [4], [13], [3], [11], [14], and of cause previous, related work ([12], [5], to mention but a few). The proposed approach is compatible with any of the currently used systems-engineering methods, such as SysML, AADL, industry standard tools such as Matlab-Simulink/Stateflow or industry standards covering lower levels of the design, such as Autosar or IMA, in that it allows to enrich purely static interface specifications by *contracts* characterizing allowed design contexts, and specifying both functional and extra-functional characteristics *guaranteed* by design artifacts, and *assumed* from their design context. While viewpoint specific extensions of system engineering methods exist (such as the MARTE profile focusing on real-time), a key asset of the proposed approach is the capability of providing a *unified data-model* of design artifacts integrating all viewpoints required for systems engineering. This allows to capture both cross-engineering domain and cross-supplier chain interfaces with a level of precision expected to drastically reduce the system integration problems highlighted in Table I.

This paper focuses on boosting the quality of virtual integration testing through the use of multi-viewpoint formal contracts, i.e. contracts with a rigorous and unambiguous semantics. Rather than "physically" integrating a system from subsystems at a particular level of the right-hand side of the V, model-based design allows to *virtually* integrate systems based on the models of their subsystem and the architecture specification of the system, which in particular explicates the information flow between subsystems and the systems environment. Such virtual integration thus allows detecting potential integration problems up front, in the early phases of the V. Virtual system integration is often a source of heterogeneous system models, such as when realizing an aircraft function through the combination of mechanical, hydraulic, and electronic systems. While virtual integration is already well anchored in many system companies' development processes, the challenge rests in lifting this from the current level of simulation-based analysis of functional system requirements to rich virtual integration testing catering as well for non-functional requirements. In *contract-based virtual integration testing*, both subsystems and the complete system are equipped with multi-viewpoint contracts. Since subsystems now characterize their legal environments, we can flag situations, where a subsystem is used out of specification, i.e. in a design context, for which no guarantees on the subsystem's reaction can be

This research has been partly funded by the Federal Ministry for Education and Research of Lower Saxony under Grant No. O1IS08045 W. The responsibility for the content lies with the authors.

[1]VDC Research, 2008 Embedded Software Market Intelligence Program, Track 3: Embedded Systems Market Statistics, Volume 1: Automotive/Transportation; Volume 3: Industrial Automation; and Volume 4: Medical Devices, 2008

[2]www.cesarproject.eu

[3]www.speeds.eu.com

[4]www.combest.eu

978-3-9810801-7-9/DATE11/© 2011 EDAA

Design Task	Tasks delayed automotive	Tasks delayed industrial automation	Tasks delayed medical devices	Tasks causing delay automotive	Tasks causing delay industrial automation	Tasks causing delay medical devices
System integration test and verification	63, 0%	56, 5%	66, 7%	42, 3%	19, 0%	37, 5%
System architecture design and specification	29, 6%	26, 1%	33, 3%	38, 5%	42, 9%	31, 3%
Software application and/or middleware development and test	44, 4%	30, 4%	75, 0%	26, 9%	31, 0%	25, 0%
Project management and planning	37, 0%	28, 3%	16, 7%	53, 8%	38, 1%	37, 5%

Note: Percentages sum to greater than 100% due to multiple responses.

TABLE I: Difficulties Related to System Integration

given. Our experience from a rich set of industrial applications shows, that such virtual integration tests drastically reduce the number of late integration errors. Special instances of failed virtual integration tests include the lack of a component to provide complete fault isolation (a property presumed by a neighboring subsystem), the lack of a subsystem to stay within the failure hypothesis assumed by a neighboring subsystem, the lack of a subsystem to provide a response within an expected time-window, the unavailability of a shared resource such as a bus-system in a specified time-window, non-allowed memory accesses, glitch rates exceeding specified bounds, and signal strengths not meeting specified thresholds.

Multi-viewpoint contracts in virtual integration testing thus drastically extend the potential to uncover integration errors early. Additionally, using such rich contracts in system specification comes with two key benefits, which help dealing with the complexity in the OEM-supplier relationship.

First, the above approach to virtual integration testing is purely based on the subsystems' contract specifications. In other words, if virtual integration testing is successful, *any* implementation of a subsystem compliant to its contract specification will not invalidate the outcome of the test. Second, assuming that the virtual integration test was passed successfully, we can verify whether the system itself meets its contract purely based on the knowledge of the subsystem contracts and the system architecture (and evidence that the subsystem implementations are compliant with this contract).

This entails that, at any level of the supplier hierarchy, the higher-level organization can – prior to contracting suppliers – analyze, whether the subsystems contracts pass the virtual integration test and are sufficient to establish the system requirements. By then basing the contracts to suppliers on the subsystem contracts, and requiring subsystem suppliers to give evidence (such as through testing or through formal analysis methods) that their implementation complies to their contract, the final integration of subsystems to the complete system will be free of all classes of integration errors covered by contracts in the virtual integration test. Note that this method allows to protect the IP of subsystem suppliers – the only evidence required, is the confirmation that their implementation meets the subsystem contract specification.

This paper is organized as follows. We introduce the key concepts of our design methodology in Section II. Section III shows how to formalize contracts based on the requirement specification language RSL. Section IV illustrates multi-aspect contract based architecture design, focusing on two key aspects, safety and real-time.

II. Key Concepts

Rich Components: The main object of the design process is a (rich) component, which represents the system or one of its parts in a certain stage of design. The rich component may (and usually will) show different *aspects* of the design object: functionality, timing, safety, cost, weight, security etc. Under each aspect, the component has an interface consisting of *ports* by which it may be connected to other components or the environment, and exchange information. The aspect under which a component is viewed has an obvious influence on the kind of information. For instance, for safety considerations it is important to know which faults may have occurred, the functional aspect will be concerned with the relation between input and output, while timing considers temporal relations and not the values themselves. So the designer is free to choose different (or differently timed) ports for the diverse aspects.

The language HRC of heterogeneous rich components provides the syntactical means to define the design artifacts. Its syntactic categories include components, interfaces, connectors and compositions. It imports languages for contract and behavior specifications as explained later in this paper.

Design Space: We imagine the design artifacts, characterized in their aspects as indicated above, to be placed on a two-dimensional map. One dimension is the *level of detail* or *abstraction*, the other is spanned by *perspectives*. These are viewpoints which, in contrast to the aspects which are oriented towards properties and behavior, concern more the nature and constitution of the design entity. We identify an operational, a functional, a logical, a technical and a geometrical perspective as ones which may play a prominent role in the design of embedded systems. The design of a new system would start, for instance, with a coarse-grained view on the system's main operational features, and end with detailed descriptions of the technical and geometrical realization, proceeding perhaps along a roughly diagonal path through the design space while detailing the different aspects along the way.

Key Design Steps: In traversing the design space, the artifacts become more and more detailed and concrete, while each new artifact is developed from a previous one and bears a particular relationship to it. We distinguish three main kinds. *Decomposition* details the structure of a component. It consists of *parts* – instances of other components – and

connectors, which link a port of a part with another or an external port. While decomposition need not necessarily lead to a new (lower) level of abstraction, the more general *realization* implements a higher-level component by other means, e.g. replacing abstract data by concrete representations. *Deployment* is the step from one perspective to another, often entailing a structural change, when, for instance, functions are allocated onto logical components, or logical components are split over controllers in the technical perspective.

Logical Specifications – Contracts: To specify an aspect of a rich component, one may use a predicative description in the form of *contracts*, saying what the component *guarantees* if *assumptions* about its environment are met. Such contracts permit a formalization of informal descriptions of qualities and properties which make up a large part of design documents. The formalization helps in making them more precise, less ambiguous and easier to check.

A contract has the format (A, B, G), where A and B are, resp., *strong* and *weak assumptions*, and G is the *guarantee*. Intuitively, the strong assumption shall express conditions that are necessary for the component to perform any meaningful operation, its "operational envelope". In the technical perspective, this could be the availability of power, a maximal temperature, etc., in the functional perspective one might postulate e.g. type correctness. The weak assumption shall describe sorts of environments in which the component provides certain services. The guarantee part then specifies what the component itself will ensure, provided that the assumptions are fulfilled.

Logically, this semantics corresponds to $A \wedge B \Rightarrow G$. On the logical level, there is no difference between weak and strong assumption, but there is a methodological one in their usage.

Each of the three formulas which make up a contract denotes, semantically, a set of *traces*, where a trace assigns a value to each port of a component for every point in time. The exact nature of the time domain is not important for the generic method. A very general domain may have a continuum of points (i.e., the real numbers) to represent continuous evolutions, with additional breaks enabling to incorporate discrete actions in adequate abstract way. With $[\![.]\!]$ denoting the semantical mapping and $(.)^{cmpl}$ trace-set complementation, $[\![(A, B, G)]\!] = ([\![A]\!] \cap [\![B]\!])^{cmpl} \cup [\![G]\!]$. If a component aspect is given by a set of contracts, the respective trace sets are intersected — this corresponds to the (common) view that all specifications given for an entity should be satisfied.

The language in which the properties A, B and G are expressed may vary. In the next section, we will present a natural-language-like dialect of temporal logic apt for industrial usage.

A fundamental relation between contracts is *refinement*. It is similar to satisfaction; a contract C' refines a contract C if $[\![C']\!] \subseteq [\![C]\!]$, i.e., if, logically, $C' \Rightarrow C$. Refinement can often be proved by showing that the refining contract's assumptions are weaker and its commitment is stronger. This stronger relation is called *dominance*.

Due to the predicative nature of contract specifications it may be difficult to see whether a given specification is meaningful, i.e., whether there is a system satisfying the specification. Though there is no simple general answer to this problem, there are some useful properties which can be established. It may be possible, for instance, to check whether a given set of contracts is logically satisfiable. Another approach takes the operational intuition into account. A formula is *receptive* on a set of ports if it does not restrict the values of those ports. There are, for many logics, sufficient syntactic criteria for that. A contract is called *directed*, if its assumptions are receptive on the outports of a component and its guarantee is receptive on the inports. Directed contracts always have a nontrivial implementation.

Behavioral Specifications – State Machines: An operational specification can be given in the form of *state machines*. State machines enable an early animation of (parts of) the system which supports explorations and plausibility checks. UML state machines or closely related formats, e.g. forms of hybrid automata, may be used for that purpose. In the following, we will not detail state machines and their usage in the design process, but will concentrate on contracts.

A state machine $\mathtt{M}$ *satisfies* a contract specification C (denoted as $\mathtt{M} \models C$), if its trace set is contained in that of the contracts. This is the most basic notion of implementation and underlies all relations which give a precise meaning, on the semantic level, to the correctness of design steps. Note that satisfaction is not meant to be achieved by making the state machine not reactive, the machine shall answer to all input stimuli in some form. Rather, the machine will usually be more deterministic – while the specification will not restrict the behavior in many situations (or only barely will do so), an implementation will be more definite and concrete.

Semantical Representations of Design Steps: Decomposition is, apart from changes to sets of observables, a form of parallel composition. On the semantic level, it is essentially – neglecting the necessary interface operations – the intersection of the trace sets of the parts of the aggregate. This corresponds to a synchronous composition. If an asynchronous composition is to be treated, there are ways to reduce it to the synchronous operator, for instance by an explicit modeling of communication media. A decomposition is correct, if the intersection of the trace sets of the parts form a subset of permitted trace sets of the aggregate.

Realization needs a mapping of the lower-level observations to the higher-level ones. This may be a simple function on the value domains of ports, or a regrouping and combination of trace elements (for instance, if an abstract operation is realized by a lower-level protocol). Correctness is mainly trace set inclusion subject to the mapping. However, a realization will in general be only partial, that is, it will work only for a subset of input values or sequences. Thus, additional restrictions will have to be imposed on the higher-level specifications to be satisfied by the realization.

A similar mechanism is used for deployment, where again the change of the semantical domain necessitates a mapping.

Virtual Integration Testing (VIT): The design steps of decomposition, realization and deployment have their counterparts in logical operations on contract specifications, and their

correctness translates to logical conditions. Let a component M with specification C and parts $\mathtt{M}_i$, $i = 1, \ldots, n$ with respective specifications C_i be given. If the parts satisfy their contracts ($\bigwedge_{i=1}^{n}(\mathtt{M}) \models C_i$) and the combined contracts imply the main specification

$$\left(\left(\bigwedge_{i=1}^{n} C_i\rho_i\right)\rho\right) \Rightarrow C$$

(the ρ_i and ρ mimic the ports connections logically in the form of substitutions), then $\mathtt{M} \models C$, i.e., the composed system is correct. This is the main condition of what we call the *virtual integration test*. Additionally, one has to establish that the strong assumptions of the parts are either discharged by guarantees of other parts or covered by the strong assumption of the aggregate. This translates to

$$\left(\bigwedge_{i=1}^{n} C_i\right) \wedge (A\rho^{-1}) \Rightarrow \bigwedge_{i=1}^{n} A_i \ .$$

The more general design steps of realization and deployment have similar, yet a bit more complex logical counterparts. These conditions enable to check the correctness of design decisions early in the process – before the parts are implemented one can check their cooperation. This is what we call the *virtual integration test*.

In the ensuing sections, we will demonstrate how the design method can be realized and employed, starting with an introduction to the pattern-based language used to formulate contracts.

III. RSL

Formulating requirements in natural language has the disadvantage that the lack of formality may lead to ambiguities, incompleteness or even inconsistencies. These may remain undetected and later on cause realizations to not conform to the intended meaning of their specification or being incompatible with other parts of the system. Formal languages, on the other hand, are often hard to learn and, even if correctly employed, might result in specifications which are difficult to understand and use. The pattern-based *Requirements Specification Language* (RSL) fills this gap by providing an easy to learn formal language with a fixed semantics that is still readable like natural language.

Patterns consist of static text elements and attributes being filled in by the requirements engineer. Each pattern has a well defined semantics in order to ensure a consistent interpretation of the written system specification across all project participants. Different sets of patterns have been defined to adequately capture the different aspects of a design. *Functional* patterns provide means to express the relationships between events, the handling of conditions and port assignments, and when these relationships should hold. *Safety* patterns speak about elementary faults, occurrences of failures and dependencies between failures or hazards. *Timing* patterns can be used to describe real-time behaviour of systems, including periodic and aperiodic phenomena and things like jitter and delay. RSL

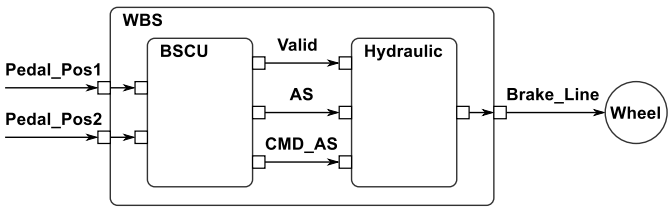

Fig. 1: High-level View of the Wheel Brake System

covers also further aspects. Instead of a full presentation, we provide a picture of the language by a giving a set of typical examples. A more complete presentation can be found in [10], including its formal semantics.

Fig. 1 gives a very high-level architecture of a safety-critical wheel-brake system (c.f. [1], [2]), with a controller (BSCU) on the left-hand side and a hydraulic subsystem on the right. The BSCU controls the regular braking process via CMD_AS. There is also a mechanical backup system (activated if Valid goes down), which is of no concern here.

One of the requirements on the system is that the delay between a brake command (given via the brake pedal) and its execution (by applying brake force to the wheel) shall not exceed $10\,ms$. An according property in the timing aspect is:

Delay between
(*change*(Pedal_Pos1) **||** *change*(Pedal_Pos2))
and *change*(Brake_Line) **within** [0 ms,10 ms]

Here, *change*(p) refers to the event of a change of the port p, not specifying the exact value. This reflects a convenient abstraction for studying timing properties. Similar, *falls*(p) defines the event that occurs when the (boolean valued) port p changes its value from 1 to 0.

Examples of functional patterns are:

always Valid
always Pedal_Pos1 == Pedal_Pos2

These two are expressing invariant properties over values of ports.

Safety contracts specify (a general form of) fault containment properties. For component BSCU in Fig. 1 for example, it shall be assumed that neither of the pedal position inputs fails. This can be stated in RSL as following:

Globally none of failure-sets
{ *fail*(Pedal_Pos1) }, { *fail*(Pedal_Pos2) } **occur**

The expression *fail*(.) refers to the (boolean) condition flag of the respective port.

Assumptions and guarantees of contracts are built up from instances of RSL patterns, where, if necessary, also boolean combinations of patterns may be used. In this way, we get highly readable and clear specifications with nonetheless precise semantics.

IV. Multi-Criteria Architecture Design

We demonstrate the contract-based design process by a few illustrative steps, starting with the timing requirement on the wheel brake system given in the following contract.

WBS:

A: **always** Pedal_Pos1 == Pedal_Pos2
G: **Delay between** (*change*(Pedal_Pos1) **||** *change*(Pedal_Pos2)) **and** *change*(Brake_Line) **within** [0 ms, 10 ms]

The strong assumption presupposes that the two ports signaling the pedal position to the WBS, which are later used in redundant subcomponents, always carry the same signal.

Time Budgeting: The first design step is time budgeting which distributes this latency requirement to the electronic and hydraulic components.

BSCU:

A: **always** Pedal_Pos1 == Pedal_Pos2
G: **Delay between** (*change*(Pedal_Pos1) **||** *change*(Pedal_Pos2)) **and** (*change*(CMD_AS) **||** *falls*(Valid)) **within** [0 ms, 5 ms]

Hydraulic:

A: - -
G: **Delay between** *change*(CMD_AS) **and** *change*(Brake_Line) **within** [0 ms, 5 ms]

The two component contracts imply the contract on WBS under the additional provision that Valid remains up. This is a (simple) instance of a realization relation, where refinement – and accordingly, the VIT – must be relativized to take low-level phenomena into account. Dealing with Valid is up to the safety analysis.

Safety: A more detailed view on the BSCU, depicted in Fig. 2, shows that it is redundantly implemented. If BSCU1 fails, the Select_Switch puts the backup signal from BSCU2 through.

The safety analysis enriches the design model by faults and introduces new observables. In our case, we have elementary faults of the four units Monitor1, Command1, Monitor2 and Command2, which are represented by ports (*fault*(Monitor1) etc.), and failures of the signals CMD_AS1, CMD_AS2 and CMD_AS (*fail*(CMD_AS etc.). A fault of Command1 leads to *fail*(CMD_AS1), which, if Monitor1 is not at fault, is signaled by *falls*(Valid1). Our example analysis assumes that at most one of the basic faults occurs, abbreviated as No_Double_Fault which stands for a condition expressing:

always ((*fault*(Command1)==0 **&&** *fault*(Monitor1)==0 **&&** *fault*(Monitor2)==0)
or **always** ((*fault*(Command1)==0 **&&** *fault*(Monitor1)==0 **&&** *fault*(Command2)==0)
or ....

Other components are assumed to be reliable (which would be ensured by an adequate design assurance level). The safety analysis abstracts from exact timing and represents important timing issues, similar to elementary faults, by additional inputs. Here, we introduce CMD_AS_too_late, which, in the timing aspect, is defined as

Delay between
(*change*(Pedal_Pos1) **||** *change*(Pedal_Pos2))
and *change*(CMD_AS) **within** (5 ms, ∞] .

A typical element of a component specification with respect to the safety aspect takes the following form:

BSCU1:

A: **Globally none of failure-sets** { *fail*(Pedal_Pos1) } **occur**
B: **always** (*fault*(Command1) ==0 **&&** *fault*(Monitor1) ==0)
G: **Globally none of failure-sets** { *fail*(CMD_AS1) } **occur**

From similar failure occurrence and propagation characterizations of the parts of BSCU, one can derive via VIT steps:

BSCU:

A: **Globally none of failure-sets** { *fail*(Pedal_Pos1) }, { *fail*(Pedal_Pos2) } **occur**
B: No_Double_Fault
G: (**always** Valid1) **or** (**always** Valid2)

and

A: **Globally none of failure-sets** { *fail*(Pedal_Pos1) }, { *fail*(Pedal_Pos2) } **occur**
B: No_Double_Fault **and** (**always not** CMD_AS_too_late)
G: **Globally none of failure-sets** { *fail*(CMD_AS) } **occur**

The condition on the timeliness of CMD_AS is subsequently treated by returning to timing considerations and refining that aspect.

Refined Timing: Of the available 5 ms to change CMD_AS after a change to the pedal position, 4 ms may be taken by the BSCUi:

G: (**always** Valid$i$) **implies Delay between** *change*(Pedal_Pos$i$) **and** (*change*(CMD_ASi)) **within** [0 ms, 4 ms]

A switch to the backup unit takes at most 1 ms.

Select_Switch:

G: **Delay between** *falls*(Valid1) **and** CMD_AS=CMD_AS2 **within** [0 ms, 1 ms]

Otherwise, the signal CMD_AS1 just has to be put through:

Select_Switch:

B: **always** Valid1
G: **Delay between** *change*(CMD_AS1) **and** *change*(CMD_AS) **within** [0 ms, 0.25 ms]

VIT lets us conclude that, given parts satisfying these contracts, the component BSCU satisfies the following:

A: **always** Pedal_Pos1 == Pedal_Pos2
B: (**always** Valid1) **or** (**always** Valid2)
G: **Delay between** *change*(Pedal_Pos1) **||** *change*(Pedal_Pos2) **and** *change*(CMD_AS) **within** [0 ms,5 ms]

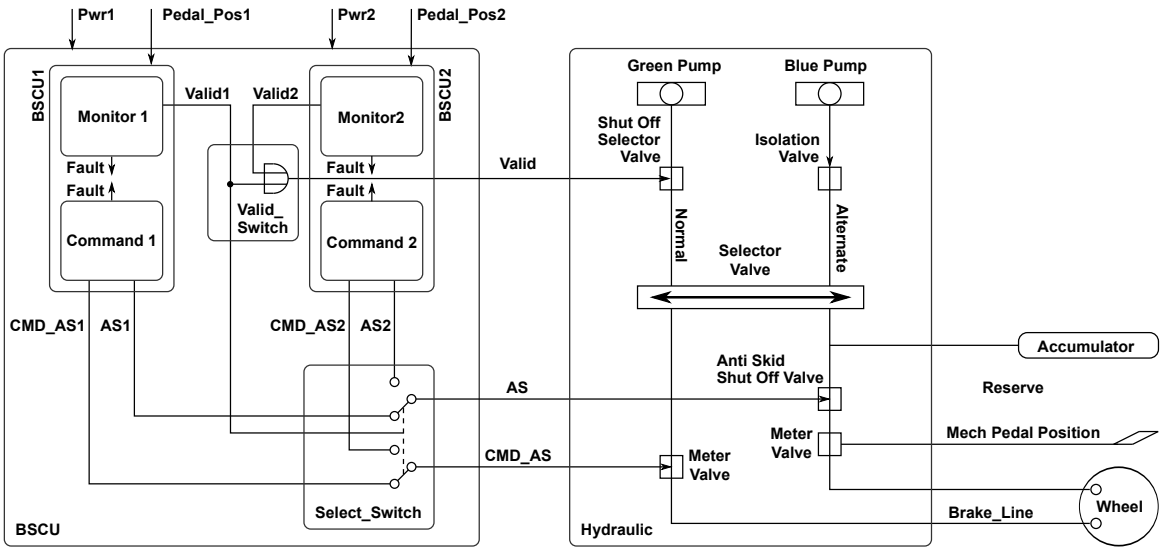

Fig. 2: Structure of the Wheel Brake System

The assumption A tells us, that `BSCU1` and `BSCU2` get the same input. To derive G, let us assume that "**always** `Valid1`" or "**always** `Valid2`" holds. If the first is true, `BSCU1` reacts in at most 4 ms (first contract for i=1) and `Select_Switch` in at most 0.25 ms (second contract for `Select_Switch`). In the second case, the argument is similar (employing the remaining contracts), and times add up to 5 ms.

Combining timing and safety: The refined timing analysis above yields, assuming additionally that either `Valid1` or `Valid2` remains up, that the `BSCU` meets its timing requirement. The safety analysis has already established that no single fault leads to both `Validi` going down. Now, we can strengthen the result of the safety analysis by stating that also `CMD_AS_too_late` will not occur under the condition `NO_Double_Fault`. Also, we can replace the condition that `Valid` remains up in the first time budgeting step by `NO_Double_Fault`.

Thus, the `WBS` has no single point of failure, if the subcomponents conform to their specifications, and it also never violates its timing requirements.

V. Conclusion and Future Work

We have outlined the foundations and the mode of application of a design style which uses contracts to specify properties of different categories (aspects) of design artifacts. Instances and elaborations of this approach are under development in various contexts – see the introduction –, which includes tools to support the application and guidelines for tailoring it to different application areas. Space limitations keep us from providing more detail in this paper. A more comprehensive presentation is in preparation, a preliminary version of which may be obtained by contacting the authors of this overview.

Acknowledgement: We sincerely thank our colleagues Eckard Böde and Markus Oertel for their contributions to earlier versions of this paper.

References

[1] ARP4754. *Certification Considerations for Highly-Integrated or Complex Aircraft Systems*. Aerospace Recommended Practice, USA, 1996.

[2] ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. Aerospace Recommended Practice, Society of Automotive Engineers, USA, 1996.

[3] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In Frank S. Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul Roever, editors, *Formal Methods for Components and Objects, FMCO '07*, LNCS 5382, pages 200–225. Springer, 2008.

[4] A. Benveniste, B. Caillaud, R. Passerone, B. Josko, E. Badouel, and B. Delahaye. SPEEDS Meta-model Behavioural Semantics. Technical report, SPEEDS Consortium, Jan 2007.

[5] Manfred Broy. A functional rephrasing of the assumption/commitment specification style. *Formal Methods in System Design*, 13(1):87–119, 1998.

[6] W. Damm. Controlling speculative design processes using rich component models. In *5th International Conference on Application of Concurrency to System Design (ACSD 2005)*, 2005.

[7] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting re-use of embedded automotive applications through rich components. In *Foundations of Interface Technologies, FIT'05*, 2005.

[8] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A next-generation design framework for platform-based design. In *Proc. of Conference on Using Hardware Design and Verification Languages (DVCon) '07*, 2007.

[9] A. Engel, M. Winokur, G. Döhmen, and M. Enzmann. Assumptions / promises - shifting the paradigm in systems-engineering. In *Proceedings of the INCOSE International Symposium 2008, June 2008 Utrecht*, 2008.

[10] V. Gafni, A. Benveniste, B. Caillaud, S. Graf, and B. Josko. SPEEDS Deliverable D.2.5.4: Contract Specification Language (CSL). D_2_5_4_RE_Contract_Specification_Language.pdf on http://www.speeds.eu.com/downloads/, 2008.

[11] B. Josko, Q. Ma, and A. Metzner. Designing embedded systems using heterogeneous rich components. In *Proceedings of the INCOSE International Symposium 2008, June 2008 Utrecht*, 2008.

[12] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[13] R. Passerone, J.R. Burch, and A.L. Sangiovanni-Vincentelli. Refinement preserving approximations for the design and verification of heterogeneous systems. *Form. Methods Syst. Des.*, 31:1–33, August 2007.

[14] R. Passerone, I. Ben Hafaiedh, S. Graf, A. Benveniste, D. Cancila, A. Cuccuru, S. Gerard, F. Terrier, W. Damm, A. Ferrari, L. Mangeruca, B. Josko, T. Peikenkamp, and A. Sangiovanni-Vincentelli. Metamodels in Europe: Languages, tools, and applications. *IEEE Design and Test of Computers*, 26:38–53, 2009.