European Journal of Control (2012)3:217–238
© 2012 EUCA
DOI:10.3166/EJC.18.217–238

European Journal of Control

Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*

Alberto Sangiovanni-Vincentelli[1,3,**], Werner Damm[2,4], Roberto Passerone[3]

[1] University of California, Berkeley, CA, USA;
[2] Kuratorium OFFIS e.V., Oldenburg, Germany;
[3] DISI, University of Trento, Trento, Italy;
[4] Carl von Ossietzky University of Oldenburg, Oldenburg, Germany

Cyber-physical systems combine a cyber side (computing and networking) with a physical side (mechanical, electrical, and chemical processes). In many cases, the cyber component controls the physical side using sensors and actuators that observe the physical system and actuate the controls. Such systems present the biggest challenges as well as the biggest opportunities in several large industries, including electronics, energy, automotive, defense and aerospace, telecommunications, instrumentation, industrial automation.

Engineers today do successfully design cyber-physical systems in a variety of industries. Unfortunately, the development of systems is costly, and development schedules are difficult to stick to. The complexity of cyber-physical systems, and particularly the increased performance that is offered from interconnecting what in the past have been separate systems, increases the design and verification challenges. As the complexity of these systems increases, our inability to rigorously model the interactions between the physical and the cyber sides creates serious vulnerabilities. Systems become unsafe, with disastrous inexplicable failures that could not have been predicted. Distributed control of multi-scale complex systems is largely an unsolved problem.

A common view that is emerging in research programs in Europe and the US is "enabling contract-based design *(CBD)," which formulates a broad and aggressive scope to address urgent needs in the systems industry. We present a design methodology and a few examples in controller design whereby contract-based design can be merged with platform-based design to formulate the design process as a meet-in-the-middle approach, where design requirements are implemented in a subsequent refinement process using as much as possible elements from a library of available components. Contracts are formalizations of the conditions for correctness of element integration (horizontal contracts), for lower level of abstraction to be consistent with the higher ones, and for abstractions of available components to be faithful representations of the actual parts (vertical contracts).*

Keywords: Contract, cyber-physical, design methodologies, platform-based, correctness.

1. Introduction

System industry that includes automotive, avionics and consumer electronics companies are facing significant difficulties due to the exponentially rising complexity of their

**Correspondence to*: A. Sangiovanni-Vincentelli, E-mail: alberto@eecs.berkeley.edu

*Paper associated with the semi-plenary lecture of A. Sangiovanni-Vincetelli at the 50th IEEE CDC and ECC 2011, Orlando, FL, USA, December 2011.

Received 14 November 2011
Recommended by E.F. Camacho

products coupled with increasingly tight demands on functionality, correctness, and time-to-market. The cost due to being late to market or due to imperfections in the products is staggering as witnessed by the recent recalls and delivery delays that system industries had to bear. There are examples of the devastating effects that design problems may cause.

The specific root causes of these problems are complex and relate to a number of issues ranging from design processes and relationships with different departments of the same company and with suppliers[1] to incomplete requirement specification and testing.[2]

In addition, there is a widespread consensus in the industry that there is much to gain by optimizing the implementation phase that today is only considering a very small subset of the design space. Some attempts at a more efficient design space exploration have been afoot but there is a need to formalize the problem better and to involve in major ways the different players of the supply chain. Information about the capabilities of the subsystems in terms of timing, power consumption, size, weight and other physical aspects transmitted to the system assemblers during design time would go a long way in providing a better opportunity to design space exploration.

In this landscape, a wrong turn in a system design project could cause an important economic, social and organizational upheaval that may imperil the life of an entire company. No wonder that there is much interest in risk management approaches to assess risks associated to design errors, delays, recalls and liabilities. Finding appropriate countermeasures to lower risks and to develop contingency plans is then a mainstay of the way large projects are managed today. The overarching issue is the need of a substantive evolution of the design methodology in use today in system companies. The issue to address is the understanding of the principles of system design, the necessary change to design methodologies, and the dynamics of the supply chain. Developing this understanding is necessary to define a sound approach to the needs of the system companies as they try to serve their customers better, to develop their products faster and with higher quality.

The focus of this paper is on cyber-physical systems (CPS) [13], [28], [43]. Cyber-physical systems are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa.

[1] Toyota sticky accelerator problem came in part from components provided by two contractors whose interaction was not verified appropriately, Airbus delay problems were in part due to contractors who had different versions of the CAD software.

[2] Boeing stated that a structural problem was discovered late in the design process.

The emerging applications of cyber-physical systems are destined to run in distributed form on a platform that meshes high performance compute clusters (the infrastructure core) with broad classes of mobiles in turn surrounded by even larger swarms of sensors (from the very large to the microscopic). The broad majority of these new applications can be classified as "distributed sense and control systems" that go substantially beyond the "compute" or "communicate" functions, traditionally associated with information technology. These applications have the potential to radically influence how we deal with a broad range of crucial problems facing our society today: for example, national security and safety, including surveillance, energy management and distribution, environment control, efficient and reliable transportation and mobility, and effective and affordable health care. A recurring property of these applications is that they engage all the platform components simultaneously—from data and computing services on the cloud of large-scale servers, data gathering from the sensory swarm, and data access on the mobiles. Another property is that the resulting systems span many scales—in space (from the very large to the very small), in time (from the very fast to the very slow), in function (consisting of complex hierarchies of heterogeneous functionalities), and in technology (integrating a broad range of diverse technologies). Each of the components of this distributed platform (compute and data clusters, mobiles/portables, and sensory systems) forms a multi-scale system on its own, and offers some unique design challenges.

Engineers today do successfully design cyber-physical systems in a variety of industries. Unfortunately, the development of systems is costly, and development schedules are difficult to stick to. The complexity of cyber-physical systems, and particularly the increased performance that is offered from interconnecting what in the past have been separate systems, increases the design and verification challenges. As the complexity of these systems increases, our inability to rigorously model the interactions between the physical and the cyber sides creates serious vulnerabilities. Systems become unsafe, with disastrous inexplicable failures that could not have been predicted.

The challenges in the realization and operation of these multi-scale systems are manifold, and cover a broad range of largely unsolved design and run-time problems. These include: modeling and abstraction, verification, validation and test, reliability and resiliency, multi-scale technology integration and mapping, power and energy, security, diagnostics, and run-time management. Failure to address these challenges in a cohesive and comprehensive way will most certainly delay if not prohibit the widespread adoption of these new technologies.

We believe the most promising means to address the challenges in systems engineering of cyber-physical

systems is to employ structured and formal design methodologies that seamlessly and coherently combine the various dimensions of the multi-scale design space (be it behavior, space or time), that provide the appropriate abstractions to manage the inherent complexity, and that can provide correct-by-construction implementations.

The following technology issues must be addressed when developing new approaches to system design:

- The overall design flows for heterogeneous systems—meant here both in a technical and also an organizational sense—and the associated use of models across traditional boundaries are not well developed and understood.
- The verification of "complex systems," particularly at the system integration phase, where any interactions are complicated and extremely costly to address, is a common need in defense, automotive, and other industries.
- Dealing with variability, uncertainty, and life-cycle issues, such as extensibility of a product family, are not well-addressed using available systems engineering methodology and tools.
- System requirement capture and analysis is in large part a heuristic process, where the informal text and natural language-based techniques in use today are facing significant challenges. Formal requirement engineering is in its infancy: mathematical models, formal analysis techniques and links to system implementation must be developed.
- Design-space exploration is rarely performed adequately, yielding suboptimal designs where the architecture selection phase does not consider extensibility, re-usability, and fault tolerance to the extent that is needed to reduce cost, failure rates, and time-to-market.

The design technology challenge is to address the entire process and not to consider only point solutions of methodology, tools, and models that ease part of the design. Addressing this challenge calls for new modeling approaches that can mix different physical systems, control logic, and implementation architectures. In doing so, existing approaches, models, and tools must be subsumed and not eliminated to ensure that designers can evolve smoothly their design methods and do not reject the proposed design innovations. In particular, a design platform has to be developed to host the new techniques and to integrate a set of today's poorly interconnected tools.

A common view that is emerging in research programs in Europe and the US is "enabling contract-based design," which formulates a broad and aggressive scope to address urgent needs in the systems industry. Contracts in the layman use of the term are established when an OEM must agree with its suppliers on the subsystem or component to be delivered. Contracts involve a legal part binding the different parties and a technical annex that serves as a reference regarding the entity to be delivered by the supplier. Contracts can also be used through their technical annex in concurrent engineering, when different teams develop different subsystems or different aspects of a system within a same company. In our view of the term, contracts can be actually used everywhere and at all stages of system design, from early requirements capture, to embedded computing infrastructure and detailed design involving circuits and other hardware. In particular, contracts explicitly handle pairs of properties, respectively representing the assumptions on the environment and the promises of the system under these assumptions. More formally, a contract is a pair $\mathcal{C} = (A, G)$ of $\{$Assumptions, Promises$\}$, where both A and G are properties satisfied by the set of all inputs and all outputs of a design.

Assume/Guarantee reasoning has been known for quite some time, but it has been used mostly as verification mean for the design of software. Our purpose is much more ambitious: contract based design with explicit assumptions is a design philosophy that should be followed all along the design, with all kinds of models, whenever necessary. Here, the models we mean are rich—not only profiles, types, or taxonomy of data, but also models describing the functions, performances of various kinds (time and energy), and safety.

To make contract-based design a technique of choice for system engineers, we must develop:

- Mathematical foundations for contract representation and requirement engineering that enable the design of frameworks and tools;
- A system engineering framework and associated methodologies and tool sets that focus on system requirement modeling, contract specification, and verification for cyber-physical systems at multiple abstraction layers.
- A systems engineering framework focusing on cross-boundary design flows that include addressing the organizational impacts of contract design and the evolution over time of systems, including configuration management.

In this paper, it is our goal to describe *contract-based design* in the context of system level design.

In the following sections, we will review methods to cope with the challenges posed in the introduction. It is indeed our take that the concept of contract is a unifying view on how to formalize requirements and rules that appear at all steps of the design process. Then, we will provide a short formalization of the notion of contract. Armed with this notion, we show how to combine contracts with platform-based design to encompass all other methods. We present a simple control problem to demonstrate the use of the proposed methodology and we close the paper

presenting potential developments that could make the use of contracts pervasive in industry.

2. Addressing the System Design Challenges: Methodologies

System companies have not yet perceived design methodology or tools to be on their critical path; hence they have not been willing to invest in expensive tools. Clearly, as they are hitting a wall in the development of the next generation systems, this situation is rapidly changing. Major productivity gains are needed and better verification and validation is a necessity as the safety and reliability requirements become more stringent and complexity is hitting an all-time high. Our experience is that many of the design chain problems are typical of very diverse verticals, the difference between them being in the importance given to time-to-market and to the customer appeal of the products versus safety and hard-time constraints. This consideration motivates the view that a unified methodology and framework could be used in many (if not all) industrial vertical domains.

Our view, shared by the research community, is that a new design science must then be developed to address the challenges listed above where the physical is married to the abstract, where the world of analog signals is coupled with the one of digital processing, and where ubiquitous sensing and actuation make our entire environment safer and more responsive to our needs. System design should be based on the new design science to address the industry and society needs in a fundamental way. However, the present directions are not completely clear as the new paradigm has not yet fully emerged in the design community with the strength necessary to change the design technology landscape, albeit researchers have chartered the field with increasing clarity. We do believe that system design needs to be concerned about the entire industrial supply chain that spans from customer-facing companies to subsystem and component suppliers, since the health of an industrial sector depends on the smooth interaction among the players of the chain as if they were part of the same company. In this section we review some of the proposed system-design methods in place to cope with these challenges, from the point of view of the system, the supply chain, and the development process including requirement capture and optimization.

2.1. Coping with Complexity of Systems

Multiple lines of attack have been developed by research institutions and industry to cope with the exponential growth in systems complexity, starting from the iterative and incremental development several decades ago [27]. Among them, of particular interest to the development of embedded controllers are: Layered design, Component-based design, the V-model process, Model-based development, Virtual integration and Platform-Based Design. We review them next.

2.1.1. Layered Design

Layered design copes with complexity by focusing on those aspects of the system pertinent to support the design activities at the corresponding level of abstraction. This approach is particularly powerful if the details of a lower layer of abstraction are encapsulated when the design is carried out at the higher layer. Layered approaches are well understood and standard in many application domains. As an example, consider the AUTOSAR standard.[3]. This standard defines several abstraction layers. Moving from "bottom" to "top", the micro-controller abstraction layer encapsulates completely the specifics of underlying micro-controllers, the second layer abstracts from the concrete configuration of the Electronic Control Unit (ECU), the employed communication services and the underlying operating system, whereas the (highest) application layer is not aware of any aspect of possible target architectures, and relies on purely virtual communication concepts in specifying communication between application components. Similar abstraction levels are defined by the ARINC standard in the avionic domains.

The benefits of using layered design are manifold. Using the AUTOSAR layer structure as example, the complete separation of the logical architecture of an application (as represented by a set of components interconnected using the so-called virtual function bus) and target hardware is a key aspect of AUTOSAR, in that it supports complete decoupling of the number of automotive functions from the number of hardware components. In particular, it is flexible enough to mix components from different applications on one and the same ECU. This illustrates the double role of abstraction layers, in allowing designers to focus completely in this case on the logic of the application and abstracting from the underlying hardware, while at the same time imposing a minimal (or even no) constraint on the design space of possible hardware architectures. In particular, these abstractions allow the application design to be re-used across multiple platforms, varying in number of bus-systems and/or number and class of ECUs. These design layers can, in addition, be used to match the boundaries of either organizational units within a company, or to define interfaces between different organizations in the supply chain.

[3] See http://www.autosar.org/

The challenge, then, rests in providing the proper abstractions of lower-level design entities. which must meet the double criteria of, on one hand, being sufficiently detailed to support virtual integration testing even with respect to non-functional viewpoints on the next higher level, while at the same time not overly restricting the space of possible lower-level implementations. As a concrete example, consider the AUTOSAR application layer and an application requiring guaranteed service under a given failure hypothesis. Such failure hypothesis would typically relate both to failures observable on the application layer itself (such as a component sending an incorrect value, a component flushing its neighbors with unwanted messages), as well as to failures depending on the underlying (unknown!) target hardware. This points to an inherent dilemma: on one side, the desire of completely abstracting from the underlying hardware, while at the same time wishing to perform analysis of properties which inherently depend on it.

Using what we call vertical assumptions as abstractions of the underlying target hardware can solve this dilemma. Returning to the above example, such vertical assumptions could explicate the failure hypothesis of either execution platforms or communication platforms, and thus decorate either (individual Runnables[4]) components or entities of the virtual function bus. In more general terms, any logical communication must be seen as a (fictitious) component itself, which, at deployment time, will be mapped to communication services of the operating system.

2.1.2. *Component-Based Design*

Whereas layered designs decompose complexity of systems "vertically", component-based approaches reduce complexity "horizontally" whereby designs are obtained by assembling strongly encapsulated design entities called "components" equipped with concise and rigorous interface specifications. Re-use can be maximized by finding the weakest assumptions on the environment sufficient to establish the guarantees on a given component implementation. While these interface specifications are key and relevant for any system, the "quality attribute" of perceiving a subsystem as a component is typically related to two orthogonal criteria, that of "small interfaces", and that of minimally constraining the deployment context, so as to maximize the potential for re-use. "Small interfaces", i.e., interfaces which are both small in terms of number of interface variables or ports, as well as "logically small", in that protocols governing the invocation of component services

[4] `Runnables` are defined in the virtual bus function specifications of AUTOSAR. `Runnable` entities are the smallest code-fragments that are provided by the component and are (at least indirectly) a subject for scheduling by the operating system.

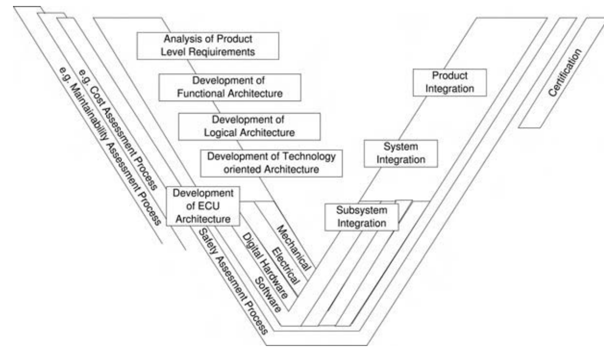

Fig. 1. The V model.

have compact specifications not requiring deep levels of synchronization, constitute evidence of the success of encapsulation. The second quality attribute is naturally expressible in terms of interface specifications, where re-use can be maximized by finding the weakest assumptions on the environment sufficient to establish the guarantees on a given component implementation.

One challenge, then, for component-based design of embedded systems, is to provide interface specifications that are rich enough to cover all phases of the design cycle. This calls for including non-functional characteristics as part of the component interface specifications, which is best achieved by using multiple viewpoints. Current component interface models, in contrast, are typically restricted to purely functional characterization of components, and thus cannot capitalize on the benefits of virtual integration testing, as outlined above.

The second challenge is related to product line design, which allows for the joint design of a family of variants of a product. The aim is to balance the contradicting goals of striving for generality versus achieving efficient component implementations. Methods for systematically deriving "quotient" specifications to compensate for "minor" differences between required and offered component guarantees by composing a component with a wrapper component (compensating for such differences as characterized by quotient specifications) exists for restricted classes of component models [36].

2.1.3. *The V-Model of the Design Process*

A widely accepted approach to deal with complexity of systems in the defense and transportation domain is to structure product development processes along variations of the V diagram shown in Fig. 1, originally developed for defense applications by the German DoD.[5]

Its characteristic V-shape splits the product development process into a *design* and an *integration* phase.

[5] See e.g., http://www.v-model-xt.de

Specifically, following product level requirement analysis, subsequent steps would first evolve a functional architecture supporting product level requirements. Subfunctions are then re-grouped taking into account re-use and product line requirements into a logical architecture, whose modules can be developed independently, e.g., by different subsystem suppliers. The realization of such modules often involves mechatronic design. The top-level of the technology-oriented architecture would then show the mechatronic architecture of the module, defining interfaces between the different domains of mechanical, hydraulic, electrical, and electronic system design, such as exemplified below for the mechatronic architecture of a simplified aircraft braking system. Subsequent phases would then unfold the detailed design for each of these domains, such as the design of the electronic subsystem involving among others the design of electronic control units. These design phases are paralleled by integration phases along the right-hand part of the V, such as integrating basic- and application software on the ECU hardware to actually construct the electronic control unit, integrating the complete electronic subsystems, integrating the mechatronic subsystem to build the module, and integrating multiple modules to build the complete product. Not shown, but forming an integral part of V-based development processes are testing activities, where at each integration level test-suites developed during the design phases are used to verify compliance of the integrated entity to their specification.

This presentation is overly simplistic in many ways. The design of electronic components in complex systems such as aircrafts inherently involves multi-site, multi-domain and cross-organizational design teams, reflecting, e.g., a partitioning of the aircraft into different subsystems (such as primary and secondary flight systems, cabin, fuel, and wing), different domains such as the interface of the electronic subsystem to hydraulic and/or mechanical subsystems, control-law design, telecommunications, software design, hardware design, diagnostics, and development-depth separated design activities carried out at the OEM and supplier companies. This partitioning of the design space (along perspectives and abstraction layers) naturally lends itself to a parallelization of design activities, a must in order to achieve timely delivery of the overall product, leading often into the order of hundreds of concurrent design processes.

Secondly, each of these sub-processes will have its own design basis, as determined by the role of an organization in the supplier chain. As previously pointed out in the section of layered design, abstraction levels define, then, what is seen as basic design-unit at a given level in the supplier hierarchy, such as on the module-level (such as an aircraft- engine), the ECU level (such as in traditional automotive development processes, where tier 1 suppliers were providing a complete ECU implementing a single new vehicle function), or the microprocessor layer. This approach is further elaborated in the section on platform-based design below.

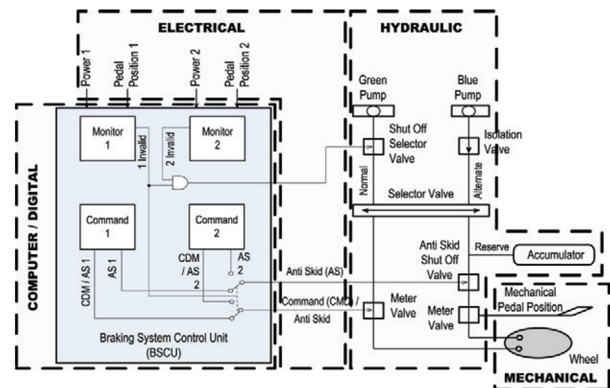

Fig. 2. The technical architecture of an airplane braking system.

Third, and tightly linked to the previous item, is the observation, that re-use strategies such as component-based design and product line design lead to separate design activities, which then short-cut or significantly reduce the effort both in design and integration steps in the individual sub-processes for an individual product.

Finally, Fig. 2 indicates the need of supporting processes for key viewpoints, such as for safety, where domain standards prescribe activities to be carried out during product development, which are often anchored with separate roles in the organization, e.g., Airbus Recommended Practices 4754 prescribes the activities in a safety assessment process as well as its interface to the aircraft development process, ultimately yielding a safety case to be approved by certification authorities.

2.1.4. *Model-Based Development*

Model-based development (MBD) is today generally accepted as a key enabler to cope with complex system design due to its capabilities to support early requirement validation and virtual system integration. MBD-inspired design languages and tools such as SysML[6] [33] and/or AADL [35] for system level modeling, Catia and Modelica [18] for physical system modeling, Matlab-Simulink [23] for control-law design, and UML[7] [7], [31] Scade [6] and TargetLink for detailed software design, depend on design layer and application class. The state-of-the-art in MBD includes automatic code-generation, simulation coupled with requirement monitoring, co-simulation of heterogeneous models such as UML and

[6] http://www.omg.org/spec/SysML/

[7] http://www.omg.org/spec/UML/

Matlab-Simulink, model-based analysis including verification of compliance of requirements and specification models, model-based test-generation, rapid prototyping, and virtual integration testing as further elaborated below.

In MBD today non-functional aspects such as performance, timing, power or safety analysis are typically addressed in dedicated specialized tools using tool-specific models, with the entailed risk of incoherency between the corresponding models, which generally interact. To counteract these risks, meta-models encompassing multiple views of design entities, enabling co-modeling and co-analysis of typically heterogeneous viewpoint specific models have been developed. Examples include the MARTE UML [32] profile for real-time system analysis, the SPEEDS HRC metamodel [37] and the Metropolis semantic meta-model [2], [3], [11], [41]. In Metropolis multiple views are accommodated via the concept of "quantities" that annotate the functional view of a design and can be composed along with subsystems. Quantities are equipped with an "algebra" that allows quantities associated to compositions of subsystems to be computed from the quantities of each of the subsystems. Multiple quantities such as timing and power can be handled simultaneously. Along the same lines, the need to enable integration of point-tools for multiple viewpoints with industry standard development tools has been the driving force in providing the SPEEDS meta-model building on and extending SysML, which has been demonstrated to support co-simulation and co-analysis of system models for transportation applications allowing co-assessment of functional, real-time and safety requirements, and forms an integral part of the meta-model-based inter-operability concepts of the CESAR (see www.cesarproject.eu) reference technology platform. The SPEEDS meta-model building on and extending SysML has been demonstrated to support co-simulation and co-analysis of system models for transportation applications allowing co-assessment of functional, real-time and safety requirements. It forms an integral part of the meta-model-based inter-operability concepts of the CESAR reference technology platform.

Meta-modeling is also at the center of the model driven (software) development (MDD) methodology. MDD is based on the concept of the model-driven architecture (MDA), which consists of a Platform-Independent Model (PIM) of the application plus one or more Platform-Specific Models (PSMs) and sets of interface definitions. MDA tools then support the mapping of the PIM to the PSMs as new technologies become available or implementation decisions change [30]. This is similar to Platform-Based Design, however the definition of platform is not fully described in MDD nor are the semantics to be used for embedded software design. The Vanderbilt University group [24] has evolved an embedded software design methodology and a set of tools based on MDD. In their approach, models explicitly represent the embedded software and the environment it operates in and capture the requirements and the design of the application, simultaneously, using domain-specific languages (DSL). The generic modeling environment (GME) [24] provides a framework for model transformations enabling easy exchange of models between tools and offers sophisticated ways to support syntactic (but not semantic) heterogeneity. The KerMeta metamodeling workbench [17] is similar in scope.

2.1.5. Virtual Integration

Rather than "physically" integrating a system from subsystems at a particular level of the right-hand side of the V, model-based design allows systems to be virtually integrated based on the models of their subsystem and the architecture specification of the system. 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—virtual system integration then rests on well defined principles allowing the integration of such heterogeneous models. Heterogeneous composition of models with different semantics was originally addressed in Ptolemy [16] and Metropolis [2], [11], [3] albeit with different approaches. These approaches have then been further elaborated in the SPEEDS metamodel of heterogeneous rich components [10]. Virtual integration involves models of the functions, the computer architecture with its extra-functional characteristics (timing and other resources), and the physical system for control. Some existing frameworks offer significant support for virtual integration: Ptolemy II, Metropolis, and RT-Builder. Developments around Catia and Modelica as well as the new offer SimScape by Simulink provide support for virtual integration of the physical part at an advanced level.

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. An approach to do so is contract-based virtual integration testing, where 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 subsystems reaction can be given. Our experience from a rich set of industrial applications shows that such virtual integration tests drastically reduce the number of late integration errors.

Instances of virtual integration tests revealing failed integration early in the V 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 (a property presumed by a neighboring subsystem);
- 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 (a property presumed by a neighboring subsystem);
- Signal strengths not meeting specified thresholds (a property presumed by a neighboring subsystem).

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 this contract specification will not invalidate the outcome of virtual integration testing. Note that using this method the IP of subsystem suppliers is protected—the only evidence required is the confirmation that their implementation meets the subsystem contract specification. 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 subsystems contract and the systems architecture (and evidence that the subsystem implementation is 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.

2.2. Coping with the Complexity of the Supply Chain

To ensure coherent product development across complex supply chains, standardization of design entities, and harmonization/standardization of processes are key trends. There are multiple challenges in defining technical annexes to contracts between OEM and suppliers. Specifications used for procurement should be precise, unambiguous, and complete. However, a recurrent reason for failures causing deep iterations across supply chain boundaries rests in incomplete characterizations of the environment of the system to be developed by the supplier, such as missing information about failure modes and failure rates, missing information on possible sources for interferences through shared resources, and missing boundary conditions. This highlights the need to explicate assumptions on the design context in OEM-supplier contracts. In the light of an increased sharing of hardware resources by applications developed by multiple suppliers, this contract-based approach seems indispensable for resolving liability issues and allowing applications with different criticality levels to co-exist (such as ASIL levels [42], [1] in automotive).

2.2.1. *Standardization of Design Entities*

By agreeing on (domain specific) standard representations of design entities, different industrial domains have created their own *lingua franca*, thus enabling a domain wide shared use of design entities based on their standardized representation. Examples of these standards in the automotive sector include the recently approved requirement interchange format standard RIF,[8] the AUTOSAR[9] de-facto standard, the OSEK[10] operating system standard, standardized bus-systems such as CAN[11] and Flexray,[12] standards for "car2X" communication, and standardized representations of test supported by ASAM.[13] Examples in the aerospace domain include ARINC standards[14] such as the avionics applications standard interface, IMA, RTCA[15] communication standards. In the automation domain, standards for interconnection of automation devices such as Profibus[16] are complemented by standardized design languages for application development such as Structured Text.

As standardization moves from hardware to operating system to applications, and thus crosses multiple design layers, the challenge increases to incorporate all facets of design entities required to optimize the overall product, while at the same time enabling distributed development in complex supply chains. As an example, to address the different viewpoints required to optimize the overall product,

---

8 http://www.w3.org/2005/rules/wiki/RIF_Working_Group
9 http://www.autosar.org/
10 http://www.osek-vdx.org/
11 http://www.iso.org/iso/search.htm?qt=Controller+Area+Network&searchSubmit=Search&sort=rel&type=simple&published=true
12 http://www.flexray.com/
13 http://www.asam.net/
14 http://www.aeec-amc-fsemc.com/standards/index.html
15 http://www.rtca.org/
16 http://www.profibus.com/

AUTOSAR extended in transitioning from release 3.1 to 4 its capability to capture timing characteristics of design entities, a key prerequisite for assessing alternate deployments with respect to their impact on timing. More generally, the need for overall system optimization calls for the standardization of all non-functional viewpoints of design entities, an objective yet to be achieved in its full generality.

2.2.2. Standardization/Harmonization of Processes

Harmonizing or even standardizing key processes (such as development processes and safety processes) provides for a further level of optimization in interactions across the supply chain. As an example, Airbus Directives and Procedures (ADBs) provide requirements for design processes of equipment manufactures. Often, harmonized processes across the supply chain build on agreed maturity gates with incremental acceptance testing to monitor progress of supplier development towards final acceptance, often building on incremental prototypes. Shared use of Product Lifcycle Management (PLM) [38] databases across the supply chain offers further potentials for cross-supply chain optimization of development processes. Also, in domains developing safety related systems, domain specific standards clearly define the responsibilities and duties of companies across the supply chain to demonstrate functional safety, such as in the ISO 26262[17] for the automotive domain, IEC 61508[18] for automation, its derivatives Cenelec EN 50128 and 50126[19] for rail, and Do 178 B[20] for civil avionics.

Yet, the challenge in defining standards rests in balancing the need for stability with the need of not blocking process innovations. As an example, means for compositional construction of safety cases are seen as mandatory to reduce certification costs in the aerospace and rail domains. Similarly, the potential of using formal verification techniques to cope with increasing system complexity is considered in the move from DO 178 B to DO 178 C standards.

2.3. Getting Initial Requirements Right

Depending on application domains, up to 50% of all errors result from imprecise, incomplete, or inconsistent and thus unfeasible requirements. Out of the many approaches taken in industry for getting requirements right, we focus here on those for initial systems requirements, relying on ISO 26262 compliant approaches.

To cope with the inherently unstructured problem of (in)completeness of requirements, industry has set up domain- and application-class specific methodologies. As particular examples, we mention learning process, such as employed by Airbus to incorporate the knowledge base of external hazards from flight incidents, the Code of Practice proposed by the Prevent Project using guiding questions to assess the completeness of requirements in the concept phase of the development of advanced driver assistance systems. Use-case analysis methods as advocated for UML based development process follow the same objective. A common theme of these approaches is the intent to systematically identify those aspects of the environment of the system under development whose observability is necessary and sufficient to achieve the system requirements. Pushing this further again leads to using contracts: based on a determined system boundary, responsibilities of achieving requirements are split into those to be established by the system-under-development (the "guarantees" of the contract) and those characterizing admissible environments of the system-under-development (the "assumptions" of the contract).

However, the most efficient way of assessing completeness of a set of requirements is by executing it. This consists in what David Harel called "playing out" for the particular case of live sequence charts [20], [21], [22], i.e., the use of formalized contract specifications to generate trajectories of interface observations compliant with the considered set of contracts. Such simulation capabilities turn out to be instrumental in revealing incompleteness: typically, they will exhibit unexpected traces, e.g., due to an insufficient restriction of the environment, or only partially specified system reactions. Executing requirements is only possible if semi-formal or formal specification languages are used, where the particular shape of such formalizations is viewpoint and domain dependent. Examples include the use of failure propagation models for safety contracts, the use of probabilistic timed automata to specify arrival processes, the use of live sequence charts for capturing scenarios in the interaction of actors and systems, or formalized requirement languages such as the PSL standard [34] combining temporal logic and automata based specifications used in the EDA domain, or the pattern-based contract specification language defined by the integrated project SPEEDS.

In addition, using contracts resting on logic-based formalisms comes with the advantage, that "spurious" unwanted behaviors can be excluded by "throwing in" additional contracts, or strengthening assumptions, or by considering additional cases for guarantees. A second advantage rests in the capability of checking for consistency by providing effective tests, whether a set of

[17] http://www.iso.org/iso/catalogue_detail.htm?csnumber=43464
[18] http://www.iec.ch/functionalsafety/
[19] http://www.cenelec.eu/Cenelec/CENELEC+in+action/Web+Store/Standards/default.htm
[20] http://www.do178site.com/

contracts is realizable, or whether, in contrast, facets of these are inherently conflicting, and thus no implementation is feasible.

2.4. Coping with Multi-Layer Design Optimization

System designs are often the result of modifications of previous designs with the attempt of minimizing risks and reducing delays and design costs. While this was an effective way of bringing new products to market in the past, with the increase in demand for new functionality and the advances of the implementation platforms, this strategy has yielded more problems than it has fixed. Indeed, there is a shared consensus that in most of the cases the designs are not optimized in the sense that the full exploitation of the new opportunities technology offers is not achieved and that having visibility of the available options and an evaluation framework for design alternatives are a sorely missing capability.

An ideal scenario for optimization is to have access to the entire design space at the lowest possible level of abstraction and then run a global optimization algorithm that could select these components satisfying constraints and optimizing multiple criteria involving non-functional aspects of the design. Unfortunately this approach is obviously out of the question for most designs given the size of the design space and the capabilities of optimization algorithms.

What is possible is to select solutions in a pre-selected design space where the number of alternatives to choose from is finite and searchable by state-of-the-art optimization programs. Indeed, the platform-based design paradigm offers scaffolding that would support this approach. In fact, at any abstraction layer, we need to optimize with respect to the components of the platform. The selection process will have to look only at feasible combinations of the components as dictated by the composability contracts.

2.5. Managing Risk Across the Development Process

The realization of complex systems calls for design processes that mitigate risks in highly concurrent, distributed, and typically multi-domain engineering processes, often involving more than one hundred sub-processes. Risk mitigation measures typically cover all phases of design processes, ranging from ensuring high quality initial requirements to early assessments of risks in realizability of product requirements during the concept phase, to enforcing complete traceability of such requirements with requirements management tools, to managing consistency and synchronization across concurrent sub-processes using PLM tools. A key challenge rests in balancing risk reduction versus development time and effort: completely eliminating the risks stemming from concurrent engineering essentially requires a complete synchronization along a fine-grained milestone structure, which would kill any development project due to the induced delays.

Current practice leads to typically implicit assumptions about design aspects to be guaranteed by concurrent processes—designers are "speculating" on outcomes of concurrent engineering sub-processes, based on their experiences from previous designs. These assumptions should be made explicit—emphasizing once again the high methodological value of assumptions—and associate these with risk-levels, which qualify or quantify the expected risks in not achieving such assumptions [9]. This very same instrument can be put in place during the concept phase of development processes, where vertical assumptions form the key basis for assessing realizability of requirements.

3. Contract Model Overview

In this section we briefly summarize the main concepts behind contract-based design by presenting a simple generic contract model centered around the notion of *component*. A component is a hierarchical entity that represents a unit of design. Components are connected together by sharing and agreeing on the values of certain ports and variables. A component may include both *implementations* and *contracts*. An implementation M is an instantiation of a component and consists of a set P of ports and variables (in the following, for simplicity, we will refer only to ports) and of a set of behaviors, or runs, also denoted by M, which assign a history of "values" to ports. Runs are generic and abstract, since we do not need a predetermined form of behavior for our basic definitions. The particular structure of the runs is defined by specific instances of the model. For instance, runs could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model. Our basic definitions will not vary, and only the way operators are implemented is affected.

We build the notion of a contract for a component as a pair of assertions, which express its assumptions and promises. An assertion E is modeled as a set of behaviors over ports, precisely as the set of behaviors that satisfy it. An implementation M satisfies an assertion E whenever they are defined over the same set of ports and all the behaviors of M satisfy the assertion, i.e., when $M \subseteq E$. A contract is an assertion on the behaviors of a component (the promise) subject to certain assumptions. We therefore represent a contract C as a pair (A, G), where A corresponds to the assumption, and G to the promise. An implementation of a component *satisfies a contract* whenever it satisfies its promise, subject to

the assumption. Formally, $M \cap A \subseteq G$, where M and C have the same ports. We write $M \models C$ when M satisfies a contract C.

Intuitively, an implementation can only provide promises on the value of the ports it controls. On ports controlled by the environment, instead, it may only declare assumptions. Therefore, we will distinguish between two kinds of ports: those that are *controlled* and those that are *uncontrolled*. Uncontrollability can be formalized as a notion of receptiveness: for E an assertion, and $P' \subseteq P$ a subset of its ports, E is said to be P'-*receptive* if and only if for all runs σ' restricted to ports belonging to P', there exists a run $\sigma \in E$ such that σ' and σ coincide over P'. In words, E accepts any history offered to the subset P' of its ports. This closely resembles the classical notion of inputs and outputs in programs and HDLs; it is more general, however, as it encompasses not only horizontal compositions within a same layer, but also cross-layer integration such as the integration between application and execution platform performed at deployment. Contracts are therefore enriched with a profile $\pi = (\mathbf{u}, \mathbf{c})$ that partitions its set of ports.

The combination of contracts associated to different components can be obtained through the operation of parallel composition. If $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ are contracts (possibly over different sets of ports), the composite must satisfy the guarantees of both, implying an operation of intersection. The situation is more subtle for assumptions. Suppose first that the two contracts have disjoint sets of ports. Intuitively, the assumptions of the composite should be simply the conjunction of the assumptions of each contract, since the environment should satisfy all the assumptions. In general, however, part of the assumptions A_1 will be already satisfied by composing C_1 with C_2, acting as a partial environment for C_1. Therefore, G_2 can contribute to relaxing the assumptions A_1. And vice-versa. The assumption and the promise of the composite contract $C = (\pi, A, G)$ can therefore be computed as follows:

$$A = (A_1 \cap A_2) \cup \neg(G_1 \cap G_2), \tag{1}$$

$$G = G_1 \cap G_2, \tag{2}$$

which is consistent with similar definitions in other contexts [12], [14], [29]. For the profiles, we enforce the property that each port should be controlled by at most one contract. Hence, parallel composition is defined only if the sets of controlled ports of the contracts are disjoint.

Parallel composition can be used to construct complex contracts out of simpler ones, and to combine contracts of different components. Despite having to be satisfied simultaneously, however, multiple viewpoints *associated to the same component* do not generally compose by parallel composition. Take, for instance, a functional viewpoint C_f and an orthogonal timed viewpoint C_t for a component M. Contract C_f specifies allowed data pattern for the environment, and sets forth the corresponding behavioral property that can be guaranteed. For instance, if the environment alternates the values $\mathtt{T}, \mathtt{F}, \mathtt{T}, \ldots$ on port a, then the value carried by port b never exceeds x. Similarly, C_t sets timing requirements and guarantees on meeting deadlines. For example, if the environment provides at least one data per second on port a ($1ds$), then the component can issue at least one data every two seconds ($.5ds$) on port b. Parallel composition fails to capture their combination, because the combined contract must accept environments that satisfy either the functional assumptions, or the timing assumptions, or both. In particular, parallel composition computes assumptions that are too restrictive. We would like, instead, to compute the *conjunction* $\sqcap$ of the contracts, so that if $M \models C_f \sqcap C_t$, then $M \models C_f$ and $M \models C_t$. This can best be achieved by first defining a partial order on contracts, which formalizes a notion of substitutability, or refinement. We say that $C = (A, G)$ *dominates* $C' = (A', G')$, written $C \preceq C'$, if and only if $A \supseteq A'$ and $G \subseteq G'$. Dominance amounts to relaxing assumptions and reinforcing promises, therefore strengthening the contract. Clearly, if $M \models C$ and $C \preceq C'$, then $M \models C'$.

Given the ordering of contracts, we can compute greatest lower bounds and least upper bounds, which correspond to taking the conjunction and disjunction of contracts, respectively. For contracts $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ (in canonical form), we have

$$C_1 \sqcap C_2 = (A_1 \cup A_2, G_1 \cap G_2), \tag{3}$$

$$C_1 \sqcup C_2 = (A_1 \cap A_2, G_1 \cup G_2). \tag{4}$$

Conjunction of contracts amounts to taking the union of the assumptions, as required, and can therefore be used to compute the overall contract for a component starting from the contracts related to multiple viewpoints.

Relations between Contracts: We have already discussed two relations that can be established between contracts and implementations. A relation of *satisfaction*, that denotes when an implementation satisfies a contract (i.e., provides the required promises under the stated assumptions), and a relation of *refinement* between contracts, that denotes the process of concretizing the requirements by strengthening the promises and relaxing the assumptions. Contracts can also be related by a notion of *consistency* and *compatibility*. Technically, these two notions refer to individual contracts. In practice, however, violations of these properties occur as a result of a parallel composition, so that we can refer to the collection of components forming a contract as consistent or compatible.

The notion of receptiveness and the distinction between controlled and uncontrolled ports is at the basis of our

relations of consistency and compatibility between contracts. Our first requirement is that an implementations M with profile $\pi = (\mathbf{u}, \mathbf{c})$ be **u**-receptive, formalizing the fact that an implementation has no control over the values of ports set by the environment. For a contract C, we say that C is *consistent* if G is **u**-receptive, and *compatible* if A if **c**-receptive.

The sets A and G are not *required* to be receptive. However, if G is not **u**-receptive, then the promises constrain the uncontrolled ports of the contract. In particular, the contract admits no receptive implementation. This is against our policy of separation of responsibilities, since we stated that uncontrolled ports should remain entirely under the responsibility of the environment. Corresponding contracts are therefore called *inconsistent*.

The situation is dual for assumptions. If A is not **c**-receptive, then there exists a sequence of values on the controlled ports that are refused by all acceptable environments. However, by our definition of satisfaction, implementations are allowed to output such sequence. Unreceptiveness, in this case, implies that a hypothetical environment that wished to prevent a violation of the assumptions should actually prevent the behavior altogether, something it cannot do since the port is controlled by the contract. Therefore, unreceptive assumptions denote the existence of an incompatibility internal to the contract, that cannot be avoided by any environment.

4. Platform-Based and Contract-Based Design

In the previous sections, we reviewed a number of approaches that tackle the challenges set up in the introduction and we introduced the basics of contracts. We argue here that Platform Based Design (PBD) subsumes most of the other approaches to system level design and for this reason, we will use it to develop the concept of contract-based design, albeit the extreme flexibility of contracts allows their universal use in all methodologies.

Contract-based design can be merged with platform-based design to formulate the design process as a meet-in-the-middle approach, where design requirements are implemented in a subsequent refinement process using as much as possible elements from a library of available components. Contracts are formalizations of the conditions for correctness of element integration (horizontal contracts), for lower level of abstraction to be consistent with the higher ones, and for abstractions of available components to be faithful representations of the actual parts (vertical contracts). A typical use of contracts in cyber-physical system design would be to govern the horizontal composition of the cyber and the physical components and to establish the conditions for correctness of their composition.

4.1. Platform-Based Design

Platform-based design was introduced in the late 1980s to capture a design process that could encompass horizontal (component-based design, virtual integration) and vertical (layered and model-based design) decompositions, and multiple viewpoints and in doing so, support the supply chain as well as multi-layer optimization.

The idea was to introduce a general approach that could be shared across industrial domain boundaries, that would subsume the various definition and design concepts, and that would extend it to provide a rigorous framework to reason about design. Indeed, the concepts have been applied to a variety of very different domains: from automotive, to System-on-Chip, from analog circuit design, to building automation to synthetic biology.

The basic tenets of platform-based design are as follows: The design progresses in precisely defined abstraction layers; at each abstraction layer, functionality (what the system is supposed to do) is strictly separated from architecture (how the functionality could be implemented). This aspect is clearly related to layered design and hence it subsumes it.

Each abstraction layer is defined by a design platform. A design platform consists of

- *A set of library components.* This library not only contains computational blocks that carry out the appropriate computation but also communication components that are used to interconnect the computational components.
- *Models of the components that represent a characterization in terms of performance and other non-functional parameters together with the functionality it can support.* Not all elements in the library are pre-existing components. Some may be “place holders” to indicate the flexibility of “customizing” a part of the design that is offered to the designer. In this case the models represent estimates of what can be done since the components are not available and will have to be designed. At times, the characterization is indeed a constraint for the implementation of the component and it is obtained top-down during the refinement process typical of layered designs. This layering of abstractions based on mathematical models is typical of model-based methods and the introduction of non-functional aspects of the design relates to viewpoints.
- *The rules that determine how the components can be assembled and how the the functional and non-functional characteristics can be computed given the ones of the components to form an architecture.* Then, a platform represents a family of designs that satisfies a set of platform-specific constraints. This aspect is related to component-based design enriched with multiple viewpoints.

This concept of platform encapsulates the notion of re-use as a family of solutions that share a set of common features (the elements of the platform). Since we associate the notion of platform to a set of potential solutions to a design problem, we need to capture the process of mapping a functionality (what the system is supposed to do) with the platform elements that will be used to build a platform instance or an "architecture" (how the system does what it is supposed to do). The strict separation between function and architecture as well as the mapping process have been highly leveraged in AUTOSAR. This process is the essential step for refinement and provides a mechanism to proceed towards implementation in a structured way. Designs on each platform are represented by platform-specific design models. A design is obtained by a designer's creating platform instances (architectures) via composing platform components (process that is typical of component-based design), by mapping the functionality onto the components of the architecture and by propagating the mapped design in the design flow onto subsequent abstraction layers that are dealt with in the same way thus presenting the design process as an iterative refinement. This last point dictates how to move across abstraction layers: it is an important part of design space exploration and offers a way of performing optimization across layers. In this respect PBD supports multiple perspectives in a general way.

4.2. Contract-Based Design

In PBD, contracts can play a fundamental role in determining the correct composition rules so that when the architecture space is explored, only "legal" compositions of available components are taken into consideration. They can be used to verify whether the system obtained by composing the library elements according to the horizontal contracts satisfies the requirements posed at the higher level of abstraction. If these sets of contracts are satisfied, the mapping mechanism of PBD can be used to produce design refinements that are correct by construction.

To be more precise about these concepts, consider a snapshot in a platform based design process, as shown in Fig. 3, covering adjacent design layers $N+1$, N, and $N-1$. System S is realized at layer N by the composition of systems S_1, S_2, and S_3.

Horizontal Contracts: In this setting, contracts serve different objectives. As highlighted in the subsection on virtual integration testing, a key value of contracts is to detect integration errors early. We use the term *horizontal contract* related to virtual integration testing, which thus in Fig. 3 define under what conditions the integration of subsystems into the composite system S is considered successful. Specifically, a horizontal contract $\mathcal{C}^H$ of a system

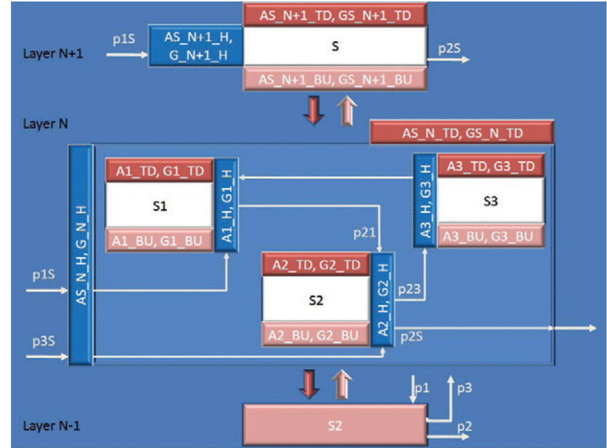

Fig. 3. Contracts in a PBD flow.

S represents in its assumption A^H the constraints it itself imposes on any possible integration context $\mathcal{C}[\,]$, so as to be able to realize its function F.

As an example of this contract-based virtual integration testing, consider Fig. 3. Each of the subsystems S_j are equipped with horizontal contracts $\mathcal{C}^H(S_j) = (A_j^H, G_j^H)$. Contract-based virtual integration testing then requires to be able to demonstrate all assumptions A_j^H from the given design context. For example, let us consider subsystem S_2. Its design context is in part given by subsystem S_1, which thus becomes responsible for establishing those horizontal assumptions relating to S_2Â's in-port p_{21}. Intuitively, then, we expect the guarantee G_1^H of the horizontal contract of S_1 to be sufficient to demonstrate compliance of any restrictions S_2 was placing on allowed uses of p_{21}. Note also the dependency of S_2 on the yet undetermined part of the design context of S_2 reflected by input p_{3S} to the realization of system S on layer N. In general, then, in contract-based virtual integration testing, this yet unknown design context is represented by horizontal contracts of the composed system S itself; Fig. 3 highlights the horizontal contract $\mathcal{C}_N^H(S) = (A_N^H(S), G_N^H(S))$ of S at layer N. This contract will enforce, that any design context of S will be guaranteed to be compliant to $A_N^H(S)$. Thus, when checking the design context of S_2 for compliance to its horizontal assumptions on uses of port p_{3S}, we expect this to be derivable from $A_N^H(S)$. In general, then, in contract-based virtual integration testing, we need to demonstrate that all horizontal assumptions of subsystems can be derived from the conjunction of all horizontal guarantees of subsystems and horizontal assumptions of the composed system.

Circular Reasoning: At the current level of discourse we point to the fact, that the above argument typically involves circular reasoning. For example, in Fig. 3 G_1^H will only be guaranteed for legal design contexts of S_1. Thus, only once A_1^H is established, can we actually rest our argumentation

on G_1^H. Establishing, then, A_1^H, we would like to involve G_3^H, which, however, is only sound once A_3^H is established. This, finally, would involve G_2^H as witness, but it is exactly for the purpose of establishing A_2^H, that this reasoning chain is established. The mathematical theory essentially justifies the use of such seemingly circular arguments, for classes of contracts whose assumptions and guarantees are expressible in the rich set of safety properties (which can always be proven and disproved by finite observations). However, certain restrictions on how assumptions and guarantees refer to out-ports respectively in-ports of a system have to be observed.

Vertical Contracts: Each of the subsystems S_j can then either be further refined, or assumed to be given as design basis at layer N, as platform library elements. Such components, as S_2 in Fig. 3, could be placeholders, to be then elaborated in a design process at layer $N-1$. Symmetrically, Fig. 3 shows the system S at layer N as a realization of the placeholder S at layer $N+1$. To transition across design layers, we use what we call *vertical contracts*.

Specifically, when using placeholder S at layer $N+1$, *bottom-up vertical contracts* are used to capture what is expected to be offered by possible implementations of S at layer N, so as to be able for S to perform its intended function at layer $N+1$ as expressed by a *top-down vertical contract*. This entails, that the correctness of the level $N+1$ design hinges on finding an implementation of S meeting this bottom-up vertical contract.

When using *budgeting*, the designer assigns responsibilities to the subsystems of S by deriving top-down contracts for each, which jointly establish SÂ´s bottom-up vertical contract. Alternatively, when using a bottom-up approach, we assume the top-down vertical contracts of S_j as given, and establish either directly or passing through a characterization of the functionality realized by S at layer N (as a top-down contract), that the layer $N+1$ bottom up contract of S is satisfied. In both the top-down and bottom up approach, the verification of this cross-layer design steps would assume that the contract-based virtual integration test was successful. This allows using the guarantees of horizontal contracts as additional premise in the verification of refinement steps.

We finally point out that additional verification steps are required for each component to demonstrate that, based on the expected capabilities of its realization, as expressed by its bottom-up vertical contract, the functionality of the component as expressed by its top-down vertical contract can be achieved. Again, this proof can take horizontal contracts of the component as additional supportive arguments. For composed systems, such as the system S at layer N in Fig. 3, the bottom-up contracts are given by the set of bottom-up contracts of its leaf components.

Crossing design layers thus asks for verification of either refinement (top-down) or aggregation (bottom-up) steps. The presentation given so far ignores extensions of the framework required in practice to deal with what is often called *interface refinement*, e.g., [8], [25]. Due to the very purpose of abstraction layers of hiding complexity, a representation of a design at level N will typically explicate implementations aspects such as representations of messages and variables, protocols used for communication and synchronization. In general, both the representation of the system in the data-domain as well as in the time domain may change, calling for notions of refinement which are expressive enough to deal both with re-timing and type conversions. The theory for these notions of weak simulation relations is well understood for particular classes of mathematical models (see [19]), which jointly are rich enough to support a broad spectrum of viewpoints, including safety, real-time, performance, power.

To allow to build on these in the methodology for contract-based design, we introduce what we call *simulation components* relating traces, i.e., sequences of observations of ports of a level $N+1$ component S to sequences of observations of ports of components S at level N. Referring to Fig. 3, this component would thus have an interface towards layer $N+1$ observing ports p_{1S} and p_{2S}, and an interface towards layer N observing ports p_{1S}, p_{2S}, and p_{3S}. Simulation components can use contracts to characterize the intended inter-relation between valuations of these. These contracts can take the form of both logic-based and automata-based formalisms, giving sufficient expressivity in capturing the intended relations between traces of interface objects of S at level $N+1$ and level N [4].

Strong vs. Weak Assumptions and the Issue of Compatibility: We close this section by pointing out a subtle, but highly relevant, difference in the methodological use of assumptions in horizontal and vertical contracts. Within horizontal contracts, assumptions are used to restrict the allowed design context of a component. By enforcing contract-based virtual integration testing, as discussed above, we therefore complement each model-based integration steps with verification activities demonstrating that the currently known design context $\mathcal{C}[\,]$ of a component S actually complies to these restrictions. This is key to enforcing what has been called *composability* of systems by [26], a fundamental principle in good architecture design ensuring functionality realized by components of the architecture are maintained when integrating these into a compound architecture. It is the purpose of assumptions to support this composability property. Specifically, if system S realizes function $F(S)$ (e.g., as expressed in a top-down vertical contract), and $\mathcal{C}[\,]$ meets the contract-based

virtual integration test for S, then S will be guaranteed to offer its functionality $F(S)$ when being put into this design context $\mathcal{C}[\,]$. We refer to assumptions which must be *enforced* for the component to behave as expected as *strong* assumptions.

In contrast, additional assumptions may be added to the strong assumption to ensure that if these assumptions are met, then "non essential" but desired properties are guaranteed. These additional assumptions are called in contrast *weak* assumptions. In vertical contracts, in particular in bottom-up contracts, weak assumptions represent anticipations often based on experience or estimation functions on what *could be assumed* to be realizable by lower implementations levels. As the designs refines vertically across multiple layers, eventually such assumptions either become validated based on top-down contracts of completed designs, or invalidated (e.g., due to insufficient processing power, non-matching assumptions on failure distributions, or insufficient signal strength). By maintaining dependency between contracts, it is then possible to backtrack to the higher-level bottom up assumption which thus became invalidated, and explore possibilities of weakening, such as by re-budgeting.

Turning assumptions in vertical contracts to *strong* assumptions would entail a *binding restriction* of the design space: a failure to meet such strong vertical assumptions would be considered a contract failure. Strong vertical assumptions can be used to enforce compliance to standards, or within the supply chain hierarchy, to eliminate the likelihood of deep design iterations crossing organizational boundaries. In a generalized setting, such as currently pushed in the context of the German Innovation Alliance for Embedded Systems,[21] we thus allow contracts to refer to both strong and weak assumptions, allowing to customize design processes supporting additional use cases of strong assumptions as outlined above.

5. Control Design and Contracts with an Example

In this section we present a simple example of control of a cyber-physical system design that makes use of the contract-based design methodology. The example, a Water Flow Control system, was first proposed by the Israel Aerospace Industries Ltd. (IAI) in the context of the SPEEDS project, and has been analyzed by Parades using hybrid modeling techniques [5]. Here we present a version using a continuous model to highlight the use of contracts in a familiar, equation-based notation. We will discuss how to model the system requirements, as well as how these are partitioned in assume/guarantee pairs (contracts) for each component of the system. Different verification and design activities can be carried out using this model.

5.1. The Water Flow Control System

A cylindrical water container is equipped with an inlet pipe at the top, and an outlet pipe at the bottom. The container has a diameter $D = 5m$ and a height $H = 9m$. The inlet and outlet cross sections are $S_{in} = 0.5m^2$ and $S_{out} = 0.16m^2$, respectively. We are to design a system that guarantees a continuous outlet flow F_{out} of $1.0 \leq F_{out} \leq 2.0m^3/sec$, after 10 seconds since startup. In addition, the system must guarantee that the container will not overflow, and that the energy consumption is lower than a limit E_l. The designer can assume a constant inlet pressure $P \geq 5,000pa$, and a maximum evaporation rate $\epsilon = 0.25m^3/hour$.

To formalize the problem, we construct a component representing the overall Water Flow Control system, with input, output and parameters corresponding to the above specification. To simplify our task, we decide to make state variables, such as the water level, visible as primary outputs. The WFC formal specification is therefore composed of the following items:

- Input: Inlet pressure P
- Output: Water Level wl, outlet flow rate F_{out}, energy consumption E
- Parameters: container size D and H, inlet cross sections S_{in} and S_{out}, evaporation rate ϵ.

To proceed with the system specification we define a contract that the implementation must satisfy. The contract distinguishes between the assumptions and the guarantees that must be enforced. Assuming t represents time, the above conditions can be formally specified as follows:

- Assumptions: $P \geq 5,000$.
- Promises:

$$\forall t.(t \geq 10 \implies (1.0 \leq F_{out} \leq 2.0))$$
$$\forall t.(wl(t) \leq H)$$
$$E \leq E_l$$

5.2. Design Solution

There are many ways to guarantee the required properties given the assumptions. Here we examine a solution method based on the regulation of the water level. From the Bernoulli Law, we know that the outlet flow rate depends on the water level according to the formula

$$F_{out} = V \cdot S_{out} = \sqrt{2g\,wl} \cdot S_{out}$$

[21] See SPES2020 Architecture Modeling Deliverable of the German Innovation Alliance on Embedded Systems SPES 2020,BMBF grant FK 01 IS O8045 W, http://spes2020.informatik.tu-muenchen.de

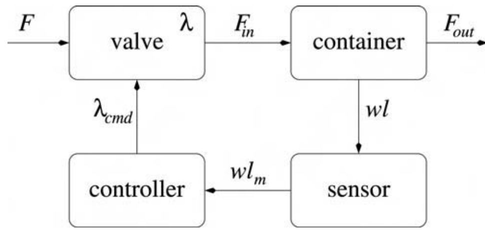

Fig. 4. Block diagram of the Water Flow Control system.

where V is the velocity. The water level is therefore given by

$$wl = \left(\frac{F_{out}}{S_{out}}\right)^2 \cdot \frac{1}{2g}.$$

Thus, the promise $1.0 \leq F_{out} \leq 2.0$ is equivalent to having

$$2.0 \leq wl \leq 8.0.$$

We will therefore approach the problem by controlling the water level in the container through a valve at the inlet. As a result, the system will be composed of an inlet valve, the water container, a water level sensor and a controller that controls the opening and closing of the valve based on the measured water level, as shown in Fig. 4.

Our methodology is the following:

- We define for each component the contract that it must satisfy
- We compose the contracts for each component
- We finally verify that the composite contract refines the contract for the system, given above.

Having verified the system at the virtual integration level, the contract theory ensures that a composition of components, each satisfying its contract, will also satisfy the system specification.

5.2.1. *Model for the Valve*

The inlet flow is controlled by a valve that may get position commands from the controller. We denote the valve aperture by λ, where $0 \leq \lambda \leq 1$. The valve is controlled by a signal λ_{cmd}, coming from the controller, whose range is also $0 \leq \lambda_{cmd} \leq 1$. The position λ of the valve follows that of the aperture command λ_{cmd} at a rate of $0.5/sec$.

Assume that F is the flow rate at the input of the valve, and call F_{in} the flow rate at the output of the valve, which is also the flow rate at the input of the container. We can express F_{in} as a function of the current valve position as follows:

$$F_{in} = F \cdot (0.2\lambda^2 + 0.8\lambda)$$

In summary, the sets of inputs and outputs for the valve is

- Input: λ_{cmd}, F
- Output: λ, F_{in}

In this simplified model, the valve must satisfy a contract that makes no assumption. In practice, one can use the assumption to limit the range of validity of the model, for example by requiring that the flow rate at the input be less than a certain value. This translates, after composition with the rest of the components, in a requirement on the environment that the pressure P be less than a certain value. Obviously, if one such assumption is introduced, the overall contract will not be satisfied, as no constraint is imposed at the system level for the pressure P other than it be greater than a certain value.

The valve satisfies the following promises.

- Rate of change of valve position
$$\frac{d\lambda}{dt} = \text{sgn}(\lambda_{cmd}(t) - \lambda(t)) \cdot 0.5$$
- Flow rate at the output of the valve
$$F_{in} = F \cdot (0.2\lambda^2 + 0.8\lambda)$$
- The initial position of the valve is closed
$$\lambda(0) = 0.$$

Sometimes it may appear ambiguous whether a certain requirement should be guaranteed by a component, or assumed from the environment. For instance, one could take the initial position of the valve as an assumption. The ambiguity disappears when one considers which component is responsible for setting a certain value. Since the position of the valve is an *output* of the valve, it is the valve responsibility to ensure its initial value, and the requirement is therefore a guarantee. A valve that does not satisfy this condition will simply not satisfy the contract.

5.2.2. *Model for the Container and the Outlet*

The container is characterized by the following inputs and outputs:

- Input: the inlet flow rate F_{in}
- Output: the water level wl and the outlet flow rate F_{out}.

The water level depends on the inlet and the outlet flow rate, as well as on the evaporation rate ε. We assume that the evaporation rate is bounded. In order to model this situation, we must add ε to the set of inputs. Then, the container must satisfy the following contract.

For the assumptions, we assume that the evaporation rate is bounded:

$$\forall t.\varepsilon(t) \leq 0.25$$

The container must ensure the following promises:

- The water level is given by the integral of the difference between the water coming in and the water going out (including the evaporation), divided by the base are of the container. Formally,

$$\forall t, t'. t' > t \implies wl(t') = wl(t) + \frac{1}{\pi(D/2)^2} \times \int_t^{t'} (F_{in}(t'') - F_{out}(t'') - \varepsilon(t''))dt''$$

- The outlet water flow is given by the Bernoulli law

$$F_{out} = V \cdot S_{out} = \sqrt{2g\, wl} \cdot S_{out}$$

5.2.3. *Model for the Water Level Sensor*

The sensor is modeled simply as a transducer that outputs a measured water level wl_m as an approximation of the real water level wl. Thus, the sensor has wl as an input and wl_m as an output. The sensor makes no assumption, and makes the promise:

$$\forall t.\, 0.95 \cdot wl(t) \leq wl_m(t) \leq 1.05 \cdot wl(t),$$

i.e., the sensor has a 5% error.

5.2.4. *Model for the Controller*

The controller takes as input the measured water level wl_m, and controls the position of the valve through the signal λ_{cmd}, which is therefore an output of the controller.

We initially experiment with a simple control function. In order to maintain the required output flow rate, and to avoid the container overflow, the valve will be opened when the water in the container goes below a certain level wl_{min} (so that the container will be filled), and will be closed when the water goes above a certain level wl_{max} (to avoid overflow). The promises of the controller are therefore as follows:

$$wl_m \leq wl_{min} \implies \lambda_{cmd} = 1$$
$$wl_m \geq wl_{max} \implies \lambda_{cmd} = 0$$

Note that the specification of the controller makes no promise when wl_m is between wl_{min} and wl_{max}. Thus the specification admits several different possible implementations for the controller.

5.2.5. *Determination of Consumed Energy*

We assume that the energy consumption is due primarily to the valve motion. We also assume that the energy is proportional to the distance traveled by the valve, which can be expressed as follows:

$$\Lambda(T) = \int_0^T \left| \frac{d\lambda}{dt} \right| dt.$$

The average distance traveled at time T is therefore

$$\overline{\Lambda(T)} = \frac{\Lambda(T)}{T}.$$

The energy can be computed using an appropriate constant c

$$E(T) = c \cdot \overline{\Lambda(T)} = c \cdot \frac{\Lambda(T)}{T}.$$

The total energy is therefore given by

$$E = c \cdot \lim_{T \to \infty} \frac{\Lambda(T)}{T}.$$

We therefore add an output E to the valve, and add the additional promise that expresses the value of E as a function of λ.

5.3. System Composition

Having defined the contracts for the component of the system, our aim is to verify that their collective requirements are consistent with the overall system contract. To do so, we must derive an overall system by taking the composition of all the contracts of the components described above. Composition in the context of this model is simple, and corresponds to putting all the equations in a system so that they are all satisfied simultaneously (i.e., we must take the intersection of the sets of solutions of the individual equations). One, however, has to take care of separating the assumptions from the guarantees, and make sure that assumptions that are not already discharged by other components of the system are properly propagated to the composite.

To give an example, we compose the model of the valve V with the model of the container C. The composition is defined, since the set of outputs of the two components is disjoint, and therefore there is no conflict over which equation to use to define the value of a variable. The composition has the following interface signals

$$I = \{\lambda_{cmd}, F, \varepsilon\}$$
$$O = \{\lambda, F_{in}, wl, F_{out}\}$$

which are obtained by taking as output any of the outputs of the two components, and as inputs the remaining signals. The composite must satisfy the following assumption, which is an assumption of the container which is not discharged by the valve:

$$\forall t. \varepsilon(t) \leq 0.25$$

In addition, the composite must satisfy all of the following promises:

$$\frac{d\lambda}{dt} = \text{sgn}(\lambda_{cmd}(t) - \lambda(t)) \cdot 0.5$$
$$F_{in} = F \cdot (0.2\lambda^2 + 0.8\lambda)$$
$$\lambda(0) = 0$$
$$\forall t, t'. t' > t \implies wl(t') = wl(t) + \frac{1}{\pi(D/2)^2} \times \int_t^{t'} (F_{in}(t'') - F_{out}(t'') - \varepsilon(t''))dt''$$
$$F_{out} = V \cdot S_{out} = \sqrt{2g\, wl} \cdot S_{out}$$

Note that some of the outputs may now be hidden in the composition. For instance, the output F_{in} does not need to appear explicitly, as long as it is considered in the promises. That is, we need to replace the guaranteed expression of F_{in} in the expression for wl, and then remove F_{in} from the set of outputs. Likewise, we could remove λ from the set of outputs. However, since λ is not defined explicitly as a function (but rather as the solution to a differential equation) the substitution is problematic from a formal point of view. From a theoretical standpoint, however, if the constraint on λ were to be expressed for example as an extended state machine, the usual procedure of taking the product can be used to compute the final result.

Note also that the assumption coming from the container is also an assumption of the composite. This is because the other component in the composition does not discharge the assumption, which must therefore be maintained and propagated to the environment of the composition.

The parallel composition can then be extended to include the water level sensor and the controller. The final set of inputs and outputs (without hiding) is the following:

$$I = \{F, \varepsilon\}$$
$$O = \{\lambda, \lambda_{cmd}, F_{in}, wl, wl_m, F_{out}\}$$

with the additional promises

$$\forall t.\, 0.95 \cdot wl(t) \leq wl_m(t) \leq 1.05 \cdot wl(t)$$
$$wl_m \leq wl_{min} \implies \lambda_{cmd} = 1$$
$$wl_m \geq wl_{max} \implies \lambda_{cmd} = 0$$

In addition to that, we can add the output E for the energy consumption and the corresponding promise to compute the energy consumption as a function of the position of the valve. The total set of inputs and outputs is therefore:

$$I = \{F, \varepsilon\}$$
$$O = \{\lambda, \lambda_{cmd}, F_{in}, wl, wl_m, F_{out}, E\}$$

5.4. Contract Verification

Contract verification consists now in checking whether the contract for the composition that we have derived in the previous section *refines* the contract for the system, outlined in Section 5-A. Refinement, as discussed in Section 3, amounts to checking that the guarantees offered by the collection of components are stronger than the guarantees required by the overall specification (the implementation promises at least the same, or more), under a weaker set of assumptions (the implementation assumes the same from the environment, or less). These conditions, in turn, can be verified by comparing the set of solutions of the equations. Stronger guarantees mean a smaller set of solutions for the promises (a more constrained behavior), while weaker assumptions imply a larger set. Formally, if we take A and G as the sets of solutions, the contracts must satisfy the usual relation

$$A' \subseteq A$$
$$G \subseteq G'$$

where $C' = (A', G')$ is the system contract, while $C = (A, G)$ is the contract obtained by taking the composition of the contracts for each component. This formulation, however, is effective only when comparing contracts that have the *same* set of inputs and outputs. This is not the case here, since the overall system contract specification and the system composite are defined on slightly different alphabets of signals. Hence, the set of inputs and outputs must somehow be equalized. One solution is to extend the system specification to include the ports of the composition, such as λ, λ_{cmd}, F_{in} and wl_m. The promises of the system specification do not change, so that in practice the system specification allows any value on those ports. Alternatively, we may hide the extra outputs, and keep only the relevant ones, i.e., wl, F_{out} and E.

The situation is different for the inputs. First, the composition depends on F rather than on P. Thus, we must add to the composition a component PF that translates the value of P into the corresponding value of F, by applying again the Bernoulli law. That is, PF has P as an input and F as an output. After the composition, the overall composite will have P as an input (since it is an input of PF and it is not an output of any other component), and F as an output (since it is an output of PF and an input of the valve). Obviously, at this point, the port F must be hidden, since it is not an output in the system specification.

To equalize on ε, we may add it as an input to the system specification. The equations do not change, so that the system specification is effectively independent of the value of ε.

After equalization, we can check containment of the solution sets. It is apparent that the condition on the

assumptions is not satisfied: in fact, A requires that ε be bounded, which is a condition that is not specified by the system specification A'. The problem can be solved by changing the way we modeled the container. There, we made the assumption that the evaporation rate is bounded. Thus, we had to take ε as an input of the specification. A closer look, however, reveals that the evaporation rate is a function of the shape of the container. Hence, the rate of evaporation can actually be guaranteed by the container itself. Thus, ε should actually be an output of the container, and the assumption on boundedness is changed into a guarantee.

Checking the guarantees is more complex, and requires solving the system of equations that characterize the composite and deriving the expression of wl, F_{out} and E explicitly. After that, we need to check the containment relation. In our specific case, we can solve the equation for the valve under some hypothesis on the value of the command λ_{cmd}. However, the equation for the water level requires a numerical solution. One way to address this problem is to construct hybrid models of the system, as described in our previous work [5]. Questions of scalability do arise, and abstractions must be typically employed to make the solution practical.

Note that we made no assumption on P in the implementation. Therefore, A admits more solutions than A'. We can take advantage of this fact, and only check that

$$A' \subseteq A$$
$$G \cup \neg A \subseteq G' \cup \neg A'$$

since a promise is effective only if the corresponding assumptions are satisfied. Because $\neg A$ is smaller than $\neg A'$, satisfying the condition on the guarantees is easier, as the formulation applies the assumptions for the system specification to the implementation.

The model that we have developed still allows several possible implementations. In particular, nothing is said about the behavior of the controller when the water level is between wl_{min} and wl_{max}. That is, the controller may arbitrarily switch between open and close valve while in that range. While all choices may actually be such that the guarantee on F_{out} is satisfied, some choices may lead to a violation of the guarantee of the energy consumption. This would be detected during verification, if the tools used for refinement checking are powerful enough to handle the continuous time specification. The ability to formalize and check non-functional requirements, under the assumptions, is a critical advantage of a contract-based design methodology in which different viewpoints can be mixed in the specification. Observe also how the energy consumption depends on the actual behavior of the implementation, so that the two viewpoints must be integrated to obtain significant results.

Besides verification, controller synthesis can be applied to derive automatically a controller that satisfies the system contract. In this case, we take the composition of all the components, except for the controller itself. The problem consists of deriving a contract for the controller, such that when the controller is composed with the rest of the system, the composition satisfies the system specification. This problem is subject of current research, and typically suffers from high computational complexity, especially in the case of timed systems. The synthesis problem has been addressed and solved in certain circumstances through an operator of quotient [40], [39].

5.5. Vertical Contracts in Control

Contracts are most naturally established between entities or components that operate at the same level of abstraction. By sharing a common understanding of the system, two components rely on each other's guarantees to fulfil the system requirements, while assumptions formalize this interdependence thereby enabling their separate and independent implementation. Of potentially greater interest, however, is the use of contracts *across* different levels of abstraction, as described in Section 4-B. When used this way, a *vertical* contract defines a relation between the properties of a system and those of its implementation platform. In other words, the system requirements can be satisfied by operating not only at the level of the application, but also by configuring execution parameters and by taking advantage of the expected behavior, as described by the assumptions, of both the application and the platform. Co-design and multi-layer techniques are therefore fully supported by the contract models, and are well incorporated and extended by the Platform-Based Design paradigm discussed in the previous sections.

These aspects are of increasing importance in the context of control design. Martin Törngren describes controllers as "bound by contracts to the plant", in the sense that the controller parameters must refer to closed loop system dynamics, which are in turn determined by the plant dynamics [44]. Likewise, timing constraints refer to both open and closed loop systems, as the controller parameters depend on the chosen sampling period and on the particular techniques (such as delay compensation) used in designing the control loop. These contracts, therefore, extend to the implementation platform. Indeed, in control design there are three entities that interact in different ways, as illustrated in Fig. 5.

The controller implements the control law in a tight loop with the plant. At the same time, the implementation platform executes the controller and physically interfaces with the plant, defining the critical non-functional parameters (delay, jitter and throughput) that concur in establishing the system properties.

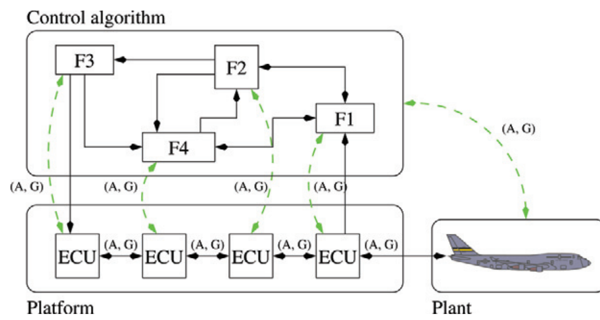

Fig. 5. Typical interactions between controller, plant and implementation platform.

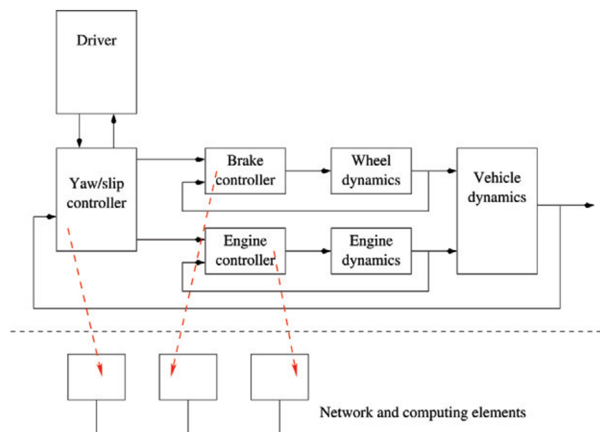

Fig. 6. Block diagram of a stability control application.

In this setting, we focus in particular on the interaction between the controller and the platform. Here the controller defines requirements in terms of several aspects that include the timing behavior of the control tasks and of the communication between tasks, their jitter, the accuracy and resolution of the computation, and more generally requirements on power and resource consumption. These requirements are taken as assumptions by the controller, which in turn provides guarantees in terms of the amount of requested computation, activation times and data dependencies.

Examples of this kind of interaction abound, and highlight the need for a theory that can integrate vertical as well as horizontal contracts. A typical application, shown in Fig. 6 and inspired by the cited presentation of Martin Törngren, is the implementation of a vehicle stability control system, in which three different controllers related to the yaw, the brakes and the engine must interact together with the wheel, the engine and the overall vehicle dynamics.

Different implementation platforms can be used to support the functionality, guaranteeing different quality levels. In this application, the control systems depends upon several subsystems, each integrated on a separate platform, and connected through often heterogeneous communication fabric. Horizontal contracts at the level of the platform can be used to understand the interactions between the subsystems, and between the system and the plant with respect to non-functional properties. Similarly, at the level of the application horizontal contracts define the global properties and the interaction between the control algorithm and the plant with respect to the functional control specification. Vertical contracts fit across these two levels as bridges that relate the performance of the different implementation platforms to their mapped applications.

The relations between contracts outlined in Section 3 can be applied to vertical contracts, as well. The relation of satisfaction is unchanged, since it involves the comparison between an implementation and its contract. In the case of an execution platform, this typically requires showing that the guaranteed timing constraints are met under the load conditions assumed by the contract. Compatibility is more interesting. In the context of a platform, the “environment” in a vertical contract refers to the other possible applications running on the same execution platform. Consequently, the composition (mapping) of an application on its platform defines the conditions (assumed and promised) under which other tasks can be run without breaking the original contract. Quantitative notions of robustness could be introduced [15] to provide a measure of the possible violations, and therefore instruct the designers or an automatic mapping tool on the steps to be taken to optimize the architecture.

6. Moving Forward: the Importance of Contracts

We argued that contracts in their most elementary form may just take the form of informal textual requirements, yet with the key distinguishing feature of explicating the separation of concerns: what must be guaranteed by the system itself, and what are the constraints on environments, which are fundamentally required so as to allow the system—based on such assumptions—to enforce its guarantees. We have then seen how this core paradigm matches well with the orthogonal notion of viewpoints: contracts can thus be flagged as to the viewpoint they relate to. Clearly, the number of viewpoints to be supported may vary from application to application—thus it is up to the customization of contract-based design within a company’s development process, to determine the set of viewpoints that must be supported. For sure, this will go beyond capturing the functionality, with safety viewpoints and real-time viewpoints being a necessity in safety relevant embedded systems development. Business related viewpoints such as the ones reflecting costs, constraints from manufacturing, maintainability are natural choices, as are those related to resource consumption.

Orthogonal to this discussion is the degree of formalization used in contracts. As highlighted above, there is already high methodological value when using informal contracts. A natural next step is to restrict the vocabulary of contracts to (domain specific) ontologies. Further steps towards formalization are viewpoint dependent: they can, for example, take the form of automata-based specifications, employ suitable logics, build on a library of patterns, and capitalize on sequence charts. The additional effort in providing a degree of formalization is typically well invested due to the additional benefits we have outlined above, such as testing consistency of requirements, identifying complex integration errors early through virtual integration testing, boosting re-use though component-based design, and allowing cross-layer design optimizations based on performing platform-based design. The key point we raise is that this formalization can be done incrementally and on a case-by-case basis. Thus, there is a clear migration strategy from using contracts informally, to incorporating domain ontologies, to gradually enriching the number of covered viewpoints, and to gradually increase the degree of formalization. No matter in which order such steps are taken, each of these comes with significant potentials for process improvements.

It is the objective of this paper to let designers capitalize on this so-far largely unexploited tool. The value proposition of adding contracts to system companies development processes is now on the table—the ultimate test rests in the market take up. Strong indications of market acceptance are the anchoring of the contract-based approach within the CESAR[22] Reference Technology Platform (RTP), where 25 European global players in the systems market team up with leading vendors, research institutes, and SMEs, to create an innovation ecosystem around the emerging standard meta-model of the CESAR RTP.

Acknowledgements

This work was supported in part by the EU project COMBEST grant n. 215543, the EU NoE ArtistDesign grant n. 214373, the EU DANSE IP, the European project CESAR of the ARTEMIS Joint Undertaking, and by the Multiscale Systems Center, one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

1. Road vehicles—functional safety. Standard ISO 26262.
2. Balarin F, Hsieh H, Lavagno L, Passerone C, Sangiovanni-Vincentelli AL, Watanabe Y. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 2003; 36(4): 45–52.
3. Balarin F, Davare A, D'Angelo M, Densmore D, Meyerowitz T, Passerone R, Pinto A, Sangiovanni-Vincentelli A, Simalatsar A, Watanabe Y, Yang G, Zhu Q. Platform-based design and frameworks: METROPOLIS and METRO II. In Nicolescu G, Mosterman PJ, editors, *Model-Based Design for Embedded Systems*, chapter 10, page 259. CRC Press, Taylor and Francis Group, Boca Raton, London, New York, November 2009.
4. Balarin F, Passerone R. Specification, synthesis and simulation of transactor processes. *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.*, 2007; 26(10): 1749–1762.
5. Benvenuti L, Ferrari A, Mangeruca L, Mazzi E, Passerone R, Sofronis C. A contract-based formalism for the specification of heterogeneous systems. In *Proceedings of the Forum on Specification and Design Languages (FDL08)*, 142–147, Stuttgart, Germany, September 23–25, 2008.
6. Berry G. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.
7. Booch G, Rumbaugh J, Jacobson I. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.
8. Broy M. Compositional refinement of interactive systems. *J. ACM*, 1997; 44(6): 555–600.
9. Damm W. Controlling speculative design processes using rich component models. In *Proceedings of 5th International Conference on Application of Concurrency to System Design (ACSD)*, 2005.
10. Damm W, Votintseva A, Metzner A, Josko B, Peikenkamp T, Böde E. Boosting reuse of embedded automotive applications through rich components. In *Proceedings of Foundations of Interface Technologies (FIT05)*, San Francisco, CA, August 21, 2005.
11. Davare A, Densmore D, Meyerowitz T, Pinto A, Sangiovanni-Vincentelli A, Yang G, Zhu Q. A next-generation design framework for platform-based design. In *Design Verification Conference (DVCon)*, San Jose', California, 2007.
12. de Alfaro L, Henzinger TA. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.
13. Derler P, Lee EA, Sangiovanni Vincentelli A. Modeling cyber-physical systems. *Proc. IEEE*, 2012; 100(1): 13–28.
14. Dill DL. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.
15. Doyen L, Henzinger T, Legay A, Nickovic D. Robustness of sequential circuits. In *Proceedings of the 10th International Conference on Application of Concurrency to System Design (ACSD 2010)*, Braga, Portugal, June 21–25, 2010.
16. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y. Taming heterogeneity - the ptolemy approach. *Proc IEEE*, 2003; 91(1): 127–144.
17. Fleurey F, Muller PA, Jzquel JM. Weaving executability into object-oriented meta-languages. In *Proceedgins of the 8th International Conference on Model Driven Engineering Languages and Systems (MODELS05)*, October 2005.
18. Fritzson P. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. Wiley, 2003.

[22] `www.cesarproject.eu`

19. Glabbeek R, Weijland WP. Branching time and abstraction in bisimulation semantics. *J ACM*, 1996; 43(3): 555–600.
20. Harel D, Kugler H, Marelly R, Pnueli A. Smart play-out of behavioral requirements. In *FMCAD*, 378–398, 2002.
21. Harel D, Marelly R. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003. http://www.wisdom.weizmann.ac.il/∼harel/. ComeLetsPlay.pdf
22. Harel D, Segall I. Planned and traversable play-out: A flexible method for executing scenario-based programs'. In *TACAS*, 485–499, 2007.
23. Karris S. Introduction to Simulink with Engineering Applications. Orchard Publications, 2006.
24. Karsai G, Sztipanovits J, Ledeczi A, Bapty T. Model-integrated development of embedded software. *Proc IEEE*, 2003; 91(1): 145–164.
25. Kesten Y, Piterman N, Pnueli A. Bridging the gap between fair simulation and trace inclusion. *Information and Computing*, 2005; 200(1): 35–61.
26. Kopetz H. Composability in the time-triggered architecture. *SAE International Congress and Exhibition (2000-01-1382), Detroit, MI, USA, 6–9 March 2000*, March 2000.
27. Larman C, Basili VR. Iterative and incremental developments: a brief history. *Computer*, 2003; 36(6): 47–56.
28. Lee EA. Cyber physical systems: Design challenges. In *Proceedings of the* 11th *IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC08)*, 363–369, May 2008.
29. Negulescu R. Process spaces. In *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
30. Object Management Group (OMG). Model driven architecture (MDA) FAQ. [online], http://www.omg.org/mda/.
31. Object Management Group (OMG). Unified Modeling Language (UML) specification. [online], http://www.omg.org/spec/UML/.
32. Object Management Group (OMG). A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG, August 2007.
33. Object Management Group (OMG). System modeling language specification v1.1. Technical report, OMG, 2008.
34. The Design Automation Standards Committee of the IEEE Computer Society, editor. *1850–2010—IEEE Standard for Property Specification Language (PSL)*. IEEE Computer Society, 2010.
35. Hudak J, Feiler P, Gluch D. The Architecture Analysis and Design Language (AADL): An Introduction. *Software Engineering Institute (SEI) Technical Note, CMU/SEI-2006-TN-011*, February 2006.
36. Passerone R, de Alfaro L, Henzinger TA, Sangiovanni-Vincentelli A. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of International Conference on Computer Aided Design*, San Jose, CA., 2002.
37. Passerone R, Hafaiedh IB, Graf S, Benveniste A, Cancila D, Cuccuru A, Gérard S, Terrier F, Damm W, Ferrari A, Mangeruca L, Josko B, Peikenkamp T, Sangiovanni-Vincentelli A. Metamodels in Europe: Languages, tools, and applications. *IEEE Design Test Computers*, 2009; 26(3): 38–53.
38. Sudarsan R, Fenves SJ, Sriram RD, Wang F. A product information modeling framework for product lifecycle management. *Computer-Aided Design*, 2005; 37: 1399–1411.
39. Raclet J-B, Badouel E, Benveniste A, Caillaud B, Legay A, Passerone R. Modal interfaces: Unifying interface automata and modal specifications. In *Proceedings of the Ninth International Conference on Embedded Software (EMSOFT09)*, 87–96, Grenoble, France, October 12–16, 2009.
40. Raclet J-B, Badouel E, Benveniste A, Caillaud B, Legay A, Passerone R. A modal interface theory for component-based design. *Fundamenta Informaticae*, 2011; 108(1–2): 119–149.
41. Sangiovanni-Vincentelli A, Shukla S, Sztipanovits J, Yang G, Mathaikutty D. Metamodeling: An emerging representation paradigm for system-level design". *Special Section on Meta-Modeling, IEEE Design and Test*, 2009; 26(3): 54–69.
42. Functional safety of electrical/electronic/programmable electronic safety-related systems. Standard IEC 61508.
43. Sztipanovits J. Composition of cyber-physical systems. In *Proceedings of the* 14th *Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS07)*, 3–6, March 2007.
44. Törngren M. Timing problems and opportunities for embedded control systems modeling and co-design, September 16, 2011. Seminar at the University of California, Berkeley.