Multiple Viewpoint Contract-Based Specification and Design*

Albert Benveniste[1], Benoît Caillaud[1], Alberto Ferrari[2], Leonardo Mangeruca[2], Roberto Passerone[2,3], and Christos Sofronis[2]

[1] IRISA / INRIA, Rennes, France
{albert.benveniste,benoit.caillaud}@irisa.fr
[2] PARADES GEIE, Rome, Italy
{alberto.ferrari,leonardo,rpasserone, christos.sofronis}@parades.rm.cnr.it
[3] University of Trento, Trento, Italy
roberto.passerone@unitn.it

Abstract. We present the mathematical foundations and the design methodology of the contract-based model developed in the framework of the SPEEDS project. SPEEDS aims at developing methods and tools to support "speculative design", a design methodology in which distributed designers develop different aspects of the overall system, in a concurrent but controlled way. Our generic mathematical model of contract supports this style of development. This is achieved by focusing on behaviors, by supporting the notion of "rich component" where diverse (functional and non-functional) aspects of the system can be considered and combined, by representing rich components via their set of associated contracts, and by formalizing the whole process of component composition.

1 Introduction

Several industrial sectors involving complex embedded systems design have recently experienced drastic moves in their organization—aerospace and automotive being typical examples. Initially organized around large, vertically integrated companies supporting most of the design in house, these sectors were restructured in the 80's due to the emergence of sizeable competitive suppliers. OEMs performed system design and integration by importing entire subsystems from suppliers. This, however, shifted a significant portion of the value to the suppliers, and eventually contributed to late errors that caused delays and excessive additional cost during the system integration phase.

In the last decade, these industrial sectors went through a profound reorganization in an attempt by OEMs to recover value from the supply chain, by focusing on those parts of the design at the core of their competitive advantage. The rest of the system was instead centered around standard platforms that could be developed and shared by otherwise competitors. Examples of this trend are AUTOSAR in the automotive industry [1], and Integrated Modular Avionics (IMA) in aerospace [2]. This new organization

* This research has been developed in the framework of the European IP-SPEEDS project number 033471.

requires extensive virtual prototyping and design space exploration, where component or subsystem specification and integration occur at different phases of the design, including at the early ones [3].

Component based development has emerged as the technology of choice to address the challenges that result from this paradigm shift. In the particular context of (safety critical) embedded systems with complex OEM/supplier chains, the following distinguishing features must be addressed. First, the need for high quality, zero defect, software systems calls for techniques in which component specification and integration is supported by clean mathematics that encompass both static and *dynamic* semantics—this means that the behavior of components and their composition, and not just their port and type interface, must be mathematically defined. Second, system design includes various aspects—functional, timeliness, safety and fault tolerance, etc.—involving different teams with different skills using heterogeneous techniques and tools. Third, since the structure of the supply chain is highly distributed, a precise separation of responsibilities between its different actors must be ensured. This is addressed by relying on contracts. Following [4] a contract is a component model that sets forth the assumptions under which the component may be used by its environment, and the corresponding promises that are guaranteed under such correct use.

The semantic foundations that we present in this paper are designed to support this methodology by addressing the above three issues. At its basis, the model is a language-based abstraction where composition is by intersection. This basic model can then be instantiated to cover functional, timeliness, safety, and dependability requirements performed across all system design levels. No particular model of computation and communication is enforced, and continuous time dynamics such as those needed in physical system modeling is supported as well. In particular, executable models such as state transition systems and hybrid automata can be used for the description of the behaviors. On top of the basic model, we build the notion of a contract, which is central to our methodology, by distinguishing between assumptions and promises. This paper focuses on developing a generic compositional theory of contracts, providing relations of contract satisfaction and refinement called dominance, and the derivation of operators for the correct construction of complete systems.

Our key contribution is the handling of multiple viewpoints. We observe that combining contracts for different components and combining contracts for different viewpoints attached to the same component requires different operators. Thus, in addition to traditional parallel composition, and to enable formal multi-viewpoint analysis, our model includes boolean meet and join operators that compute conjunction and disjunction of contracts. To be able to blend both types of operations in a flexible way, we introduce a new operator that combines composition and conjunction to compute the least specific contract that satisfies a set of specifications, while at the same time taking their interaction into account. The operators are complemented by a number of relations between contracts and their implementations. Of particular interest are the notion of satisfaction between an implementation and its contract, and relations of compatibility and consistency between contracts. Specifications are also introduced to model requirements or obligations that must be checked throughout the design process. Our second contribution consists in organizing these relations in a design and analysis methodology that spans a wide range of levels of abstraction, from the functional definition to its final hardware implementation.

The rest of paper is organized as follows. We first review and discuss previous work related to the concept of contract in the context of our contribution in Section 2. We then introduce our model by first motivating our choices, and then by defining formally the notions of component, contract, their implementations and specification in Section 3. In addition, in the same section, we introduce and discuss a number of operators and relations that support the incremental construction and verification of multi-viewpoint systems. After that, we discuss the design methodology in Section 4. Finally, Section 5 presents an illustrative example of the use of the model.

2 Related Work

The notion of contract has been applied for the first time by Meyer in the context of the programming language Eiffel [5]. In his work, Meyer uses *preconditions* and *post-conditions* as state predicates for the methods of a class, and *invariants* for the class itself. Preconditions correspond to the assumptions under which the method operates, while postconditions express the promises at method termination, provided that the assumptions are satisfied. Invariants must be true at all states of the class regardless of any assumption. The notion of class inheritance, in this case, is used as a refinement, or subtyping, relation. To guarantee safe substitutability, a subclass is only allowed to weaken assumptions and to strengthen promises and invariants.

Similar ideas were already present in seminal work by Dijkstra [6] and Lamport [7] on *weakest preconditions* and *predicate transformers* for sequential and concurrent programs, and in more recent work by Back and von Wright, who introduce contracts [8] in the *refinement calculus* [9]. In this formalism, processes are described with guarded commands operating on shared variables. Contracts are composed of *assertions* (higher-order state predicates) and *state transformers*. This formalism is best suited to reason about discrete, untimed process behavior.

Dill presents an asynchronous model based on sets of sequences and parallel composition (trace structures) [10]. Behaviors (traces) can be either accepted as *successes*, or rejected as *failures*. The failures, which are still possible behaviors of the system, correspond to unacceptable inputs from the environment, and are therefore the complement of the assumptions. Safe substitutability is expressed as trace containment between the successes and failures of the specification and the implementation. The conditions obtained by Dill are equivalent to requiring that the implementation weaken the assumptions of the specification while strengthening the promises. Wolf later extended the same technique to a discrete synchronous model [11]. More recently, De Alfaro and Henzinger have proposed Interface Automata which are similar to synchronous trace structures, where failures are implicitly all the traces that are not accepted by an automaton representing the component [12]. Composition is defined on automata, rather than on traces, and requires a procedure to restrict the state space that is equivalent to the process called autofailure manifestation of Dill and Wolf. The authors have also extended the approach to other kinds of behaviors, including resources and asynchronous behaviors [13,14]. A more general approach along the lines proposed by Dill and Wolf is the work by Negulescu with Process Spaces [15], and by Passerone with Agent Algebra [16], both of which extend the algebraic approach to generic behaviors introduced by Burch [17]. In both cases, the exact form of the behavior is abstracted, and only the properties of composition are used to derive general results that apply to both asynchronous and synchronous models. An interesting aspect of Process Spaces is the identification of several derived algebraic operators. In contrast, Agent Algebra defines the exact relation between concepts such as parallel composition, refinement and compatibility in the model.

Our notion of contract supports *speculative design* in which distributed teams develop partial designs concurrently and synchronize by relying on the notions of rich component [4] and associated contracts. We define assumptions and promises in terms of behaviors, and use parallel composition as the main operator for decomposing a design. This choice is justified by the reactive nature of embedded software, and by the increasing use of component models that support not only structured concurrency, capable of handling timed and other non-functional properties, but also heterogeneous synchronization and communication mechanisms. Contracts in [8] are of a very different nature, since there is no clear indication of the role (assumption or promise) a state predicate or a state transformer may play. We developed our theory on the basis of assertions, i.e., languages of traces or runs (not to be confused with assertions in [8], which are state predicates).

Our contracts are intended to be abstract models of a component, rather than implementations, which, in our context, may equivalently be done in hardware or software. Similarly to Process Spaces and Agent Algebra, we develop our theory on the basis of languages of generic "runs". However, to attain the generality of a metamodel, and to cover non-functional aspects of the design, we also develop a concrete model enriched with real-time information that achieves the expressive power of hybrid systems. Behaviors are decomposed into assumptions and promises, as in Process Spaces, a representation that is more intuitive than, albeit equivalent to, the one based on the successes and failures of asynchronous trace structures. Unlike Process Spaces, however, we explicitly consider inputs and outputs, which we generalize to the concept of controlled and uncontrolled signals. This distinction is essential in our framework to determine the exact role and responsibilities of users and suppliers of components. This is concretized in our framework by a notion of compatibility which depends critically on the particular partition of the signals into inputs and outputs. We also extend the use of receptiveness of asynchronous trace structures, which is absent in Process Spaces, to define formally the condition of compatibility of components for open systems.

Our refinement relation between contracts, which we call *dominance* to distinguish it from refinement between implementations of the contracts, follows the usual scheme of weakening the assumption and strengthening the guarantees. The order induces boolean operators of conjunction and disjunction, which resembles those of asynchronous trace structures and Process Spaces. To address mutliple viewpoints for multiple components, we define a new *fusion* operator that combines the operation of composition and conjunction for a set of contracts. This operator is introduced to make it easier for the user to express the interaction between contracts related to different viewpoints of a component.

The model that we present in this paper is based on execution traces, and is therefore inherently limited to representing linear time properties. The branching structure of a process whose semantics is expressed in our model is thus abstracted, and the exact state in which non-deterministic choices are taken is lost. Despite this, the equivalence relation that is induced by our notion of dominance between contracts is more distinguishing than the traditional trace containment used when executions are not represented as pairs (assumptions, promises). This was already observed by Dill, with the classic example of the vending machine [10], see also Brookes et al. on refusal sets [18]. There, every accepted sequence of actions is complemented by the set of possible *refusals*, i.e., by the set of actions that may not be accepted after executing that particular sequence. Equivalence is then defined as equality of sequences with their refusal sets. Under these definitions, it is shown that the resulting equivalence is stronger than trace equivalence (equality of trace sets), but weaker than observation equivalence [19,20]. A precise characterization of the relationships with our model, in particular with regard to the notion of composition, is deferred to future work.

3 Model Overview

In the SPEEDS project, a major emphasis has been placed on the development of a model that supports concurrent system development in the framework of complex OEM-supplier chains. This implies the ability to support abstraction mechanisms and to work with multiple viewpoints that are able to express both functional (discrete and continuous evolutions) and non-functional aspects of a design. In particular, the model should not force a specific model of computation and communication (MoCC).

The objective of this paper is to develop a theory and methodology of component based development, for use in complex supply chains or OEM/supplier organizations. Two broad families of approaches can be considered for this purpose:

– Building systems from library components. This is perhaps the most familiar case of component based development. In this case, emphasis is on reuse and adaptation, and the development process is largely in-house dominated. In this case, components are exposed in a simplified form, called their interface, where some details may be omitted. The interface of components is typically obtained by a mechanism of abstraction. This ensures that, if interfaces match, then components can be safely composed and deliver the expected service.
– Distributed systems development with highly distributed OEM/supplier chains. This second situation raises the additional and new issue of splitting and distributing responsibilities between the different actors of the OEM/supplier chain, possibly involving different viewpoints. The OEM wants to define and know precisely what a given supplier is responsible for. Since components or sub-systems interact, this implies that each entity in the area of interaction must be precisely assigned for responsibility to a given supplier, and must remain out of control for others.

Thus each supplier is given a design task in the following form: A goal, also called *guarantee* or *promise*, is assigned to the supplier. This goal involves only entities the supplier is responsible for. Other entities, which are not under the responsibility of this supplier, may still be subject to constraints that are thus offered to this supplier as *assumptions*. Assumptions are under the responsibility of other actors of the OEM/supplier chain, and can be used by this supplier for achieving its own promises. This mechanism of assumptions and promises is structured into *contracts*, which form the essence of distributed systems development involving complex OEM/supplier chains.

3.1 Components and Contracts

Our model is based on the concept of *component*. A component is a hierarchical entity that represents a unit of design. Components are connected together to form a system 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. This model essentially follows the Tagged-Signal model introduced by Lee and Sangiovanni [21], which is shown appropriate for expressing behaviors of a wide variety of models of computation. However, unlike the Tagged-Signal model, we do not need a predetermined form of behavior for our basic definitions, which will remain abstract. Instead, the way sets of behaviors are represented in specific instances will define their structure. For example, an automata based model will represent behaviors as sequences of values or events. Conversely, behaviors in a hybrid model will consist of alternations of continuous flows and discrete jumps. Our basic definitions will not vary, and only the way operators are implemented is affected. This way, our definitions are independent of the particular model chosen for the design. Thus, because implementations and contracts may refer to different viewpoints, we refer to the components in our model as *heterogeneous rich components* (HRC).

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 a property that may or may not be satisfied by a behavior. Thus, assertions can again be modeled as a set of behaviors over ports, precisely as the set of behaviors that satisfy it. Note that this is unlike preconditions and postconditions in program analysis, which constrain the state space of a program at a particular point. Instead, assertions in our context are properties of entire behaviors, and therefore talk about the dynamics of a component. 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. Satisfaction can be checked using the following equivalent formulas, where $\neg A$ denotes the set of all runs that are not runs of A:

$$M \models C \iff M \subseteq G \cup \neg A \iff M \cap (A \cap \neg G) = \emptyset$$

There exists a unique maximal (by behavior containment) implementation satisfying a contract C, namely $M_C = G \cup \neg A$. One can interpret M_C as the implication $A \Rightarrow G$.

Clearly, $M \models (A, G)$ if and only if $M \models (A, M_C)$, if and only if $M \subseteq M_C$. Because of this property, we can restrict our attention to contracts of the form $C = (A, M_C)$, which we say are in *canonical form*, without losing expressiveness. The operation of computing the canonical form, i.e., replacing G with $G \cup \neg A$, is well defined, since the maximal implementation is unique, and it is idempotent. Working with canonical forms simplifies the definition of our operators and relations, and provides a unique representation for equivalent contracts.

In order to more easily construct contracts, it is useful to have an algebra to express more complex contracts from simpler ones. The combination of contracts associated to different components can be obtained through the operation of parallel composition, denoted with the symbol $\|$. If $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ are contracts (possibly over different sets of ports), the composite $C = C_1 \parallel C_2$ 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. Formally, this translates to the following definition.

Definition 1 (Parallel Composition). *Let $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. The parallel composition $C = (A, G) = C_1 \parallel C_2$ is given by*

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

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

This definition is consistent with similar definitions in other contexts [12,10,15]. C_1 and C_2 may have different ports. In that case, we must extend the behaviors to a common set of ports before applying (1) and (2). This can be achieved by an operation of inverse projection. Projection, or elimination, in contracts requires handling assumptions and promises differently, in order to preserve their semantics.

Definition 2 (Elimination). *For a contract $C = (A, G)$ and a port p, the* elimination of p in *C is given by*

$$[C]_p = (\forall p\, A,\ \exists p\, G) \tag{3}$$

where A and G are seen as predicates.

Elimination trivially extends to finite sets of ports, denoted by $[C]_P$, where P is the considered set of ports. For inverse elimination in parallel composition, the set of ports P to be considered is the union of the ports P_1 and P_2 of the individual contracts.

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 a given value 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. Figure 1 illustrates this. Figure 1(a) shows the two contracts (C_f on the left, and C_t on the right) as truth tables. Figure 1(b) shows the corresponding inverse projection. Figure 1(d) is the parallel composition computed according to our previous definition, while Figure 1(c) shows the desired result. We would like, that is, 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.

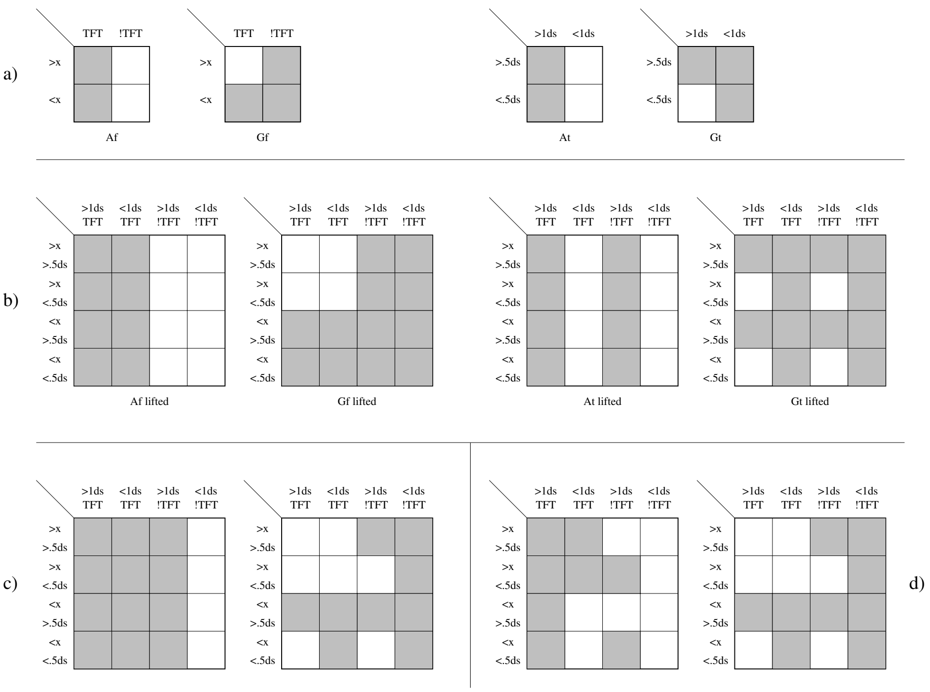


Fig. 1. Truth tables for the synchronization of categories. The four diagrams on the top are the truth tables of the functional category C_f and its assumption A_f and promise G_f, and similarly for the timed category C_t. Note that these two contracts are in canonical form. In the middle, we show the same contracts lifted to the same set of variables b, d_b, x, d_x, combining function and timing. On the bottom, the two tables on the left are the truth tables of the greatest lower bound $C_f \sqcap C_t$. For comparison, we show on the right the truth tables of the parallel composition $C_1 \parallel C_2$, revealing that the assumption is too restrictive and not the one expected.

Definition 3 (Dominance). *We say that* $C = (A, G)$ dominates $C' = (A', G')$*, written* $C \preceq C'$*, if and only if*

$$A \supseteq A', \text{and}$$
$$G \subseteq G',$$

and the contracts have the same ports.

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.

Definition 4 (Bounds). *For contracts* $C_1 = (A_1, G_1)$ *and* $C_2 = (A_2, G_2)$ *(in canonical form), we have*

$$C_1 \sqcap C_2 = (A_1 \cup A_2, G_1 \cap G_2), \tag{4}$$
$$C_1 \sqcup C_2 = (A_1 \cap A_2, G_1 \cup G_2). \tag{5}$$

The resulting contracts are in canonical form. 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.

The operations of parallel composition and conjunction are related by the result below, which allows the designer to relate in a precise way the designs obtained by following different implementation flows:

Theorem 1. *Let* C*,* C_1 *and* C_2 *be contracts. Then,*

$$(C \sqcap C_1) \parallel (C \sqcap C_2) \preceq C \sqcap (C_1 \parallel C_2)$$

when both sides of the inequality are defined.

Proof. Let $C = (A, G)$, $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ be contracts. Then, by (1), (2) and (4),

$$C \sqcap (C_1 \parallel C_2) = (A \cup (A_1 \cap A_2) \cup \neg(G_1 \cap G_2), G \cap G_1 \cap G_2).$$

Similarly,

$$(C \sqcap C_1) \parallel (C \sqcap C_2) = ((A \cup A_1) \cap (A \cup A_2) \cup \neg(G \cap G_1 \cap G_2), G \cap G_1 \cap G_2).$$

Clearly,

$$\neg(G_1 \cap G_2) \subseteq \neg(G \cap G_1 \cap G_2).$$

In addition,

$$A \cup (A_1 \cap A_2) = (A \cup A_1) \cap (A \cup A_2).$$

The result then follows by definition of dominance.

The left hand side of the formula of Theorem 1 yields the contract obtained by first combining viewpoints for each component and then composing the components. On the other hand, the right hand side of the same formula yields the contract obtained by applying the converse flow. Thus, the theorem expresses that component centric design (left hand side) results in less flexibility in the implementations than a viewpoint centric design (right hand side) would do.

3.2 System Obligations

Contracts are not the only way a designer would like to express system's requirements. System obligations are typically high level requirements that the designer would like to hold without considering any environment nor assumption. System obligations are useful for both overall system requirements and for overall properties of the computing platform.

System obligations are formally defined as assertions, i.e., sets of behaviors. An important point is that system obligations should be checked on contracts as early as possible in the design flow, because this significantly reduces the analysis effort, required to prove or disprove the obligation, and the design effort, required to revise the contract if the obligation is not met. To formalize this idea we introduce the *conformance* relation: a contract $C = (A, G)$ *conforms to* a system obligation B if $A \cap G \subseteq B$.

With each contract $C = (A, G)$ we can associate an obligation, that we call the *contract obligation*, defined as $B_C = A \cap G$. Hence, a contract conforms to a system obligation if its contract obligation is contained in (i.e., it is stronger than) the system obligation. There is a simple relationship between the maximal implementation M_C of contract C and the corresponding contract obligation B_C, namely $A \cap M_C = B_C$ and $M_C = B_C \cup \neg A$. Indeed:

$$A \cap M_C = A \cap (G \cup \neg A) = A \cap G = B_C$$
$$B_C \cup \neg A = A \cap G \cup \neg A = A \cap G \cup \neg A \cap G \cup \neg A = G \cup \neg A = M_C$$

The formulation of the conformance relation in terms of the contract obligation suggests an extension of the notion of conformance to contracts: a contract C_2 conforms to a contract C_1 if $B_{C_2} \subseteq B_{C_1}$. This definition ensures that conformance is transitive, thereby implying that contract C_2 conforms to any system obligation which C_1 conforms to. Conformance is compositional with respect to parallel composition. This follows from the fact that $B_{C_1 || C_2} = B_{C_1} \cap B_{C_2}$, i.e., the contract obligation associated with the parallel composition of C_1 and C_2 is the intersection of their contract obligations. This can be shown by using equations (1) and (2) of parallel composition. Contracts and system obligations are specifications that are intended to guide the designer(s) towards a consistent system's implementation. Hence, in the design process we intend to relate implementations to contracts and system obligations. In particular, implementations are used in the contexts defined by contracts and are meant to satisfy all system obligations. In more precise terms, given an implementation that satisfies a contract that conforms to a system obligation, we want that such an implementation also satisfy in some sense to the system obligation. To formalize this we say that an implementation M *satisfies* a system obligation B *through* a contract $C = (A, G)$ if $A \cap M \subseteq B$. It can be readily observed that if an implementation M satisfies a contract $C = (A, G)$, and if C conforms to a system obligation B, then M satisfies B through C.

Conformance and dominance between contracts are complementary, in the sense that one does not imply the other. Nevertheless, there is a strong relationship between the two. Specifically, given two contracts, $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$, if C_2 dominates C_1, then C_2 conforms to C_1 if and only if $A_2 \subseteq A_1 \cup \neg G_2$, as shown below:

$$G_2 \subseteq G_1 \Rightarrow A_2 \cap G_2 \subseteq G_1 \text{ and } A_2 \subseteq A_1 \cup \neg G_2 \iff A_2 \cap G_2 \subseteq A_1$$

Note that the condition $A_2 \subseteq A_1 \cup \neg G_2$ requires that if a given behavior is not allowed by the contract C_1 (i.e., is not in A_1), but is possible in C_2 (i.e., is in G_2), then it must be disallowed also by contract C_2 (i.e., is not in A_2). This condition together with the dominance relation is called *strong dominance*.

3.3 The Asymmetric Role of Ports

So far we have ignored the role of ports and the corresponding splitting of responsibilities between the implementation and its environment, see the discussion above. Such a splitting of responsibilities avoids the competition between environment and implementation in setting the value of ports and variables.

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 for implementations and contracts: 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.

In some cases, different viewpoints associated with the same component need to interact through some common ports. This motivates providing a scope for ports, by partitioning them into ports that are *visible* (outside the underlying component) and ports that are *local* (to the underlying component). The above discussion can be summarized as a *profile* $\pi = (\mathbf{vis}, \mathbf{loc}, \mathbf{u}, \mathbf{c})$, which partitions a set of ports P into subsets such that

$$\begin{aligned} P &= \mathbf{vis} \uplus \mathbf{loc} = \{\text{visible}\} \uplus \{\text{local}\} \\ P &= \quad \mathbf{u} \uplus \mathbf{c} \quad = \{\text{uncontrolled}\} \uplus \{\text{controlled}\} \end{aligned}$$

Thus, in addition to sets of runs, components, implementations and contracts can be characterized by a profile over a set of ports P. As before, for a contract $C = (A, G)$ or an implementation M, the sets A, G and M are constrained to include only runs over P.

The satisfaction and the dominance relations are easily extended to take profiles into account, by simply insisting that the implementations and the contracts that are put in relation have the same profile. Consequently, conjunction (the greatest lower bound) can only be taken between contracts over the same profiles. If two contracts have different profiles, then an operation of inverse projection is required. However, the resulting profiles must be consistent regarding which ports are controlled and which are uncontrolled, and local. This restriction highlights the fact that the logical operations that we have defined are relative to contracts that refer to the same components, and which must therefore treat controlled and uncontrolled ports in the same way.

The situation is different for parallel composition. Here, 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. However, one contract may regard one port as controlled, and the other as uncontrolled. In this case, we are simply stating that the controlling contract determines the value of the port for the other contract. Thus, in the composite contract, a port is controlled exactly when it is controlled by one of the component contracts. Uncontrolled ports of the contracts remain uncontrolled in the composite provided that they are not already controlled by the other contract. A similar reasoning is applied to visible and local ports. In this case, however, we distinguish between the composition of contracts for the same component, and contracts for different components. In the first case, local ports have no effect on the composition, since the scope of local ports extends to the entire component. In the second case, instead, the set of local ports of one contract must be disjoint from the set of ports of the other contract.

More formally, for contracts $C_1 = (\pi_1, A_1, G_1)$ and $C_2 = (\pi_2, A_2, G_2)$ for the same underlying component, parallel composition is defined if and only if $\mathbf{c}_1 \cap \mathbf{c}_2 = \emptyset$, and in that case is the contract $C = (\pi, A, G)$ defined by:

$$\begin{aligned} \mathbf{vis} &= \mathbf{vis}_1 \cup \mathbf{vis}_2, \\ \mathbf{loc} &= (\mathbf{loc}_1 \cup \mathbf{loc}_2) - (\mathbf{vis}_1 \cup \mathbf{vis}_2), \\ \mathbf{c} &= \mathbf{c}_1 \cup \mathbf{c}_2, \\ \mathbf{u} &= (\mathbf{u}_1 \cup \mathbf{u}_2) - (\mathbf{c}_1 \cup \mathbf{c}_2), \end{aligned}$$

The formulas are the same for contracts of different components, where composition is defined only if $\mathbf{loc}_1 \cap P_2 = \mathbf{loc}_2 \cap P_1 = \emptyset$.

3.4 Consistency and Compatibility

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 $\pi = (\mathbf{vis}, \mathbf{loc}, \mathbf{u}, \mathbf{c})$ be $\mathbf{u}$-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 $\mathbf{u}$-receptive, and
- *compatible* if A if $\mathbf{c}$-receptive.

The sets A and G are not *required* to be receptive. However, if G is not $\mathbf{u}$-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 $\mathbf{c}$-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.

The notion of consistency and compatibility can therefore be extended to pairs of contracts. We say that two contracts C_1 and C_2 are *consistent* or *compatible* whenever their parallel composition is consistent or compatible.

Consistency and compatibility may not be preserved by Boolean operations and by parallel composition. For example, one obtains an inconsistent contract when taking the greatest lower bound of two contracts, one of which promises that certain behaviors will never occur in response to a certain input, while the other promises that the remaining behaviors will not occur in response to the same input. This is because the contracts control the same ports, and composition of promises is by intersection. Similarly, assumptions may become unreceptive as a result of taking the least upper bound. We do not generally use least upper bounds, so we do not elaborate further on this situation. In general, however, the conjunction of compatible contracts is still compatible, since assumptions compose by union.

Another form of inconsistency may arise when taking parallel composition. In this case, certain input sequences may be prevented from happening because they might activate unstable zero-delay feedback loops. The resulting behaviors may have no representations in our model, thus resulting in an empty promise. This problem can be avoided by modeling the oscillating behaviors explicitly (perhaps using a special value that denotes oscillation) [11]. We assume that this kind of inconsistency is taken care of by the user or by the tools.

Assumptions may also become unreceptive as a result of a parallel composition even if they are not so individually. This is because the set of controlled ports after a composition is strictly larger than before the composition. In particular, ports that were uncontrolled may become controlled, because they are controlled by the other contract. In this case, satisfying the assumptions is the responsibility of the other contract, which acts as a partial environment. If the assumptions are not satisfied by the other contract, then the assumptions of the composition become unreceptive. That is, 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 one of the contract. Therefore, unreceptive assumptions denote the existence of an internal incompatibility within the composition.

Finally, we point out that the operation of transforming a contract to its canonical form preserves consistency (and compatibility), since the promises G are replaced by their most permissive version $G \cup \neg A$.

3.5 Fusion

When several viewpoints and several components are present in a system, combining conjunction and parallel composition may not be trivial. To overcome the problem, we define a unique operator, which combines the operations of conjunction and parallel composition, and results in an overall contract for a system. We call this operation a *fusion* of contracts. The fusion operator takes a finite set of contracts $(C_i)_{i \in I}$ as operand, and a set of ports Q to be eliminated, because internal to the component. The *fusion of* $(C_i)_{i \in I}$ *with respect to* Q is defined by

$$[\![(C_i)_{i \in I}]\!]_Q = \bigsqcap_{J \subseteq I} \left[\, \|_{j \in J} C_j \,\right]_Q, \tag{6}$$

where J ranges over the set of all subsets of I for which composition is defined and, after the composition, no input is contained in Q. In other words, fusion considers only compositions of contracts for which internal connections have been fully established and discharged, and therefore talk only about the global input to output behavior. To guarantee the maximal flexibility in fusion, the subsets J are also chosen to be maximal with respect to containment. By doing so, we avoid considering partial compositions which, when taking the conjunction, restrict the range of accepted environments, and therefore strengthen the assumptions.

Certain particular cases are of interest. For instance, when $Q = \emptyset$, the fusion reduces to the greatest lower bound: $[\![(C_i)_{i \in I}]\!]_\emptyset = \sqcap_{i \in I} C_i$. Likewise, if for $i = 1, 2$,

$$\forall Q\, (A_i \cup \neg G) \supseteq \forall Q\, (A_1 \cup A_2) \tag{7}$$

then fusion reduces to the parallel composition operator: $[\![(C_i)_{i \in \{1,2\}}]\!]_Q = [C_1 \parallel C_2]_Q$. Condition (7) says that the restriction to Q of each contract is a valid environment for the restriction to Q of the other contract. This situation corresponds to two components that interact through ports in Q, which are subsequently hidden from outside. In practice, fusion computes the parallel composition of contracts attached to different sub-components of a composite, whereas contracts attached to the same composite that involve the same inputs and outputs (including their direction) fuse via the operation of conjunction. The general case lies in between and is given by formula (6).

4 Methodology

The aforementioned relations and operations among contracts set the basis for composition and manipulation of components composed of contracts belonging to more than one viewpoint. Moreover, we provide a methodology to orchestrate the usage of the relations and give guidelines to the user on how to design and verify her model against a number of requirements/constraints that follow the laws presented in the previous sections.

We distinguish between the *Design* and the *Analysis* methodology. The former defines the design steps that the user can take for the evolution of her system, while the latter specifies the relations that should be established (or re-established) depending on the corresponding design step.

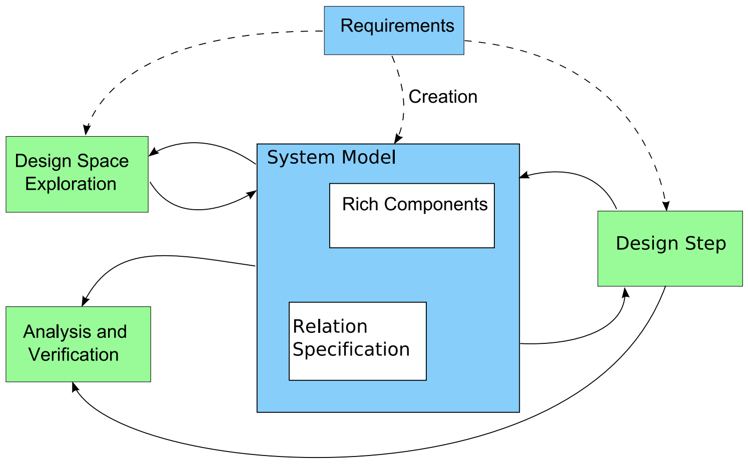


Fig. 2. Abstract view of the design methodology

An abstract representation of the design methodology is shown in Figure 2. Initially, from a set of requirements, we derive a system model composed of rich components and a (initially empty) set of relations between them. From this point, the user may take a number of steps, perform analysis to enhance this set of relations or to perform design space exploration. The latter is subject of future work.

We detail on the design steps later on, after we specify the different elements of the design.

4.1 Elements of the Design

We distinguish between four categories of design elements, as defined in Section 3: contracts, implementations, system obligations, and relations. A rule of thumb to distinguish the relations is that consistency, compatibility and dominance relations are established between two contracts; the satisfaction relation between an implementation and a contract; and the conformance relation between a contract and a system obligation. As part of the design methodology, we consider an organization of those elements in three *design spaces* and furthermore in *layers*. These are: the *implementation space*, the *contract space* and the *system obligation space*. Relations (since they are not syntactic elements of the model) are represented as "connections" between elements of the same or different spaces. For example, dominance "connects" two elements within the contract space, while satisfaction "connects" one element from the contract and one from the implementation space. Note that a relation may "connect" an element with itself, as in the case of the compatibility relation.

Each space may be further subdivided into *layers*. In the context of the SPEEDS project, only the layering of the contract space is relevant, because the main purpose of the HRC model is to represent contracts. However, a layering of the obligation and implementation spaces is also possible.

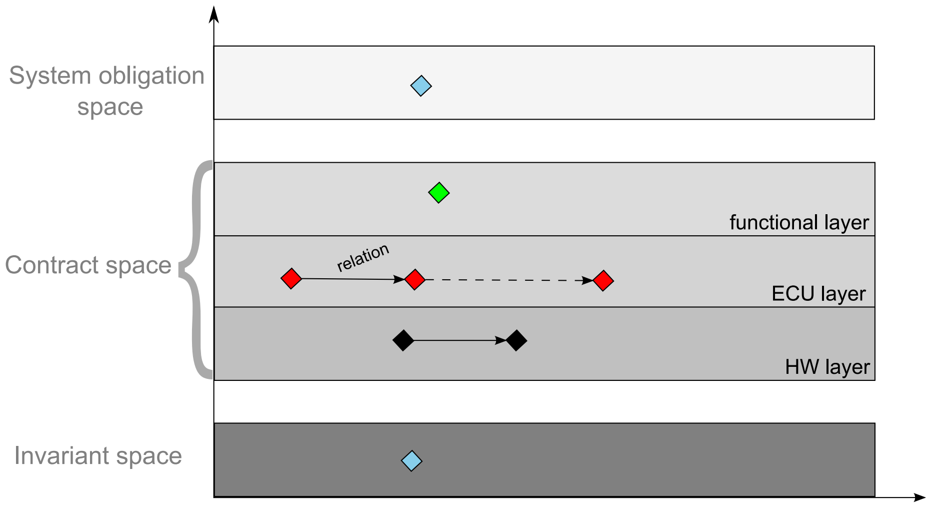


Fig. 3. Organization of spaces and layers

Figure 3 shows a possible layering of the contract space in the case of automotive applications. Here we identify three layers: 1) the functional layer, corresponding to describing the basic functional requirements and guarantees; 2) the Engine Control Unit (ECU) layer, that sets forth the high level timing and architecture assumptions and guarantees; and 3) the Hardware (HW) layer, corresponding to a more detailed description of the individual platform components. In addition, mapping an element from one layer to an element from another will create an element that does not belong to either of the operands' layers. Thus, we have three extra layers for all the possible mappings, as depicted in Figure 4; one layer for the mapping of Function to ECU, one for the mapping of ECU to HW and one for the mapping of all the layers.

4.2 Design Steps

A design step is an evolution of the development of the design, which can be seen also as the evolution of the design in time. We use the elements defined in the previous section to define the basic design steps that a user can follow during design. In principle, a design step is defined as a tuple of *source* and *target* rich components. Moreover, we specify a number of relations that are "required" between the elements of the rich components participating in the step.

A design step is said to be *validated* if the required relations hold. This validation is performed using high-level analysis services which are being developed within the SPEEDS project, and include tools that can check satisfaction, dominance, compatibility and consistency for a variety of models (from pure discrete to hybrid) using both formal and semi-formal (simulation with dynamic property checking) techniques. Moreover, we introduce the notion of *valid rich component*, that is a rich component whose contract is compatible, consistent, it is satisfied by its implementation and conforms to its obligations.

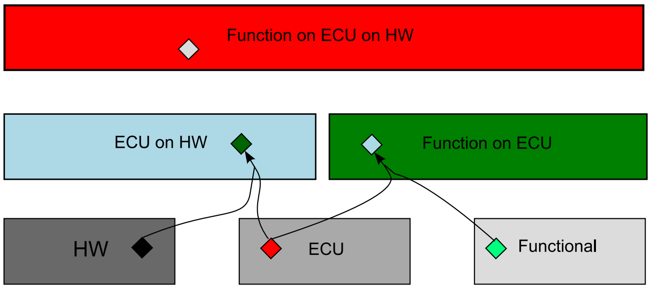


Fig. 4. Layers derived when mapping elements from different layers

When a design step is validated, and if the source rich components are valid, then also the target components are valid, which means that the resulting component can be used "safely" in place of the originating one, i.e., we can substitute without losing any verification and validation results obtained previously.

We subdivide the design steps in two categories:

1. Design steps on single rich components. The first category contains design steps which specify or modify only one element of a rich component, resulting in a new rich component where the remaining elements are unchanged.

Let $RC = \{B, C, M\}$ be the source and $RC' = \{B', C', M'\}$ the target rich components, where B, B' are the system obligations, C, C' the contracts and M, M' the implementation.[1] The design steps and the corresponding relations for their validation are described below.

System obligation modification design step is when $C = C'$ and $M = M'$, whereas $B \neq B'$. For the validation of this step there are two options:
- verify that $B' \subseteq B$
  or
- verify that C' *conforms* to $B \cup \neg B'$

Contract modification design step is when $B = B'$ and $M = M'$, whereas $C \neq C'$. For the validation of this step there are two options:
- verify that C' *strongly dominates* C
  or
- verify that C' is compatible, consistent, *conforms* to C and M' *satisfies* it

Implementation modification design step is when $B = B'$ and $C = C'$, whereas $M \neq M'$. For the validation of this step there are two options:
- verify that M' *refines* M
  or
- verify that M' *satisfies* C

The above steps are called "modifications" even though we may not have prior definition of the "modified" element, in which case, we consider the trivial element.

[1] Even though a rich component may have more than one contract or system obligation, their composition results into a unique one, and thus, without loss of generality, we consider this assumption for the rest of the document.

2. Design steps on multiple rich components. The second category contains those design steps having more than one source or more than one target rich components. Let $RC_n = \{B_n, C_n, M_n\}$ for $n \in [1..\kappa]$ be κ source and $RC'_m = \{B'_m, C'_m, M'_m\}$ for $m \in [1..\lambda]$ be λ target rich components.

Decomposition design step: For the decomposition we have $\kappa = 1$ and $\lambda \geq 2$.
Let $RC'' = \{B'', C'', M''\}$ be the parallel composition of all rich components RC'_m for $m \in [1..\lambda]$. For the validation of this step there are two options:
- verify that C'' *strongly dominates* C_1
  or
- verify that C'' is compatible, consistent, *conforms* to C_1 and M'' *satisfies* C_1

Composition design step: For the composition we have $\kappa \geq 2$ and $\lambda = 1$.
Since parallel composition preserves (strong) dominance, satisfaction and refinement, no verification task is needed for integration.

Mapping design step: Mapping is a composition (fusion) of design elements from different modeling layers and therefore we refer the reader to the discussion for composition.

Using the design steps. The above design steps are all possible actions that can advance the system design and are the "bricks" to build the design methodology. The design methodology that we follow uses these building blocks in a viewpoint centric approach. This means that we should not apply any contract prior to performing decomposition. In that way, and following Theorem 1, we retain a greater level of flexibility for the implementations that should satisfy the decomposed components. We can see this in Figure 5, where component RC, containing two contracts *Cr* and *Cf*, from the real time and functional viewpoints respectively, has two possible decompositions: rich

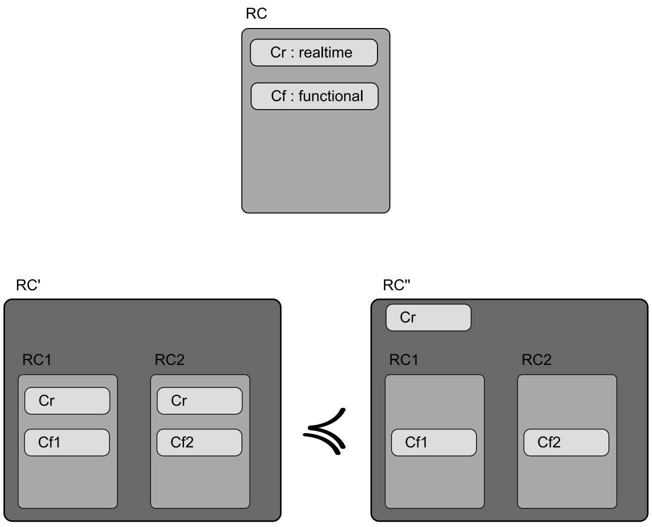


Fig. 5. Two possible decompositions of RC

components RC' and RC''. Therefore, we propose decomposition to RC'', where the real-time viewpoint is not applied (composed) to the decomposed components.

Note that the relations between the different elements of this decomposition should hold according to the definition of the *decomposition design step* above, if we want to have a valid design step. Thus, since we have no implementation or system obligation in our example, the following must hold: $Cf1 \parallel Cf2 \preceq Cf$.

5 Illustrative Example

Our approach aims at supporting component based development of heterogeneous embedded systems with multiple viewpoints, both functional and non-functional. The following simple example illustrates this for the case of functional, timed, and safety viewpoints. The top level view of our system is shown in Figure 6. It consists of a system controller that can let the underlying plant "start", "stop", or "work" (signals r, s, and w). The system controller promises that the mean amount of work performed by the plant does not exceed a maximum and that the work is not paused for too long. A human operator may decide to reinitialize the controller by sending the "reset" message $\mathbf{z}$. The system controller, which is the part of the system under design, must conform to the following obligations:

Protocol obligation: "work" requests can be sent only after a "start" and before a "stop". A "stop" must follow a "start" or a "work" request.

Longest idle time obligation: a "work" request must follow a "start" or a previous request at most after τ_{max} seconds.

Maximum mean work obligation: from the last operator "reset", the amount of "work" requests per unit of time must not be greater than $1/\xi$.

Figure 7 shows the automata that specify the obligations. For this example, the notation $[g]s/a$ denotes a transition. It is a triple consisting of a guard g, a triggering event s, and an action a. Action a may, in turn, assign some variables and/or emit some output(s). The idle time and mean work obligations are specified in terms of hybrid automata. These hybrid automata use a timer x bound to physical time, thus satisfying the differential equation $\dot{x} = 1$ (x increases with constant speed 1).

The system controller is decomposed into several components as shown on Figure 8. It consists of a simple controller that is responsible for sending the "start", "stop", and "work" signals to the underlying plant. The controller is deployed over a computing

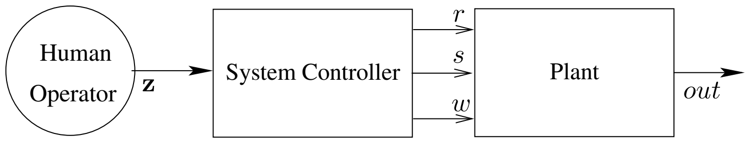


Fig. 6. System view

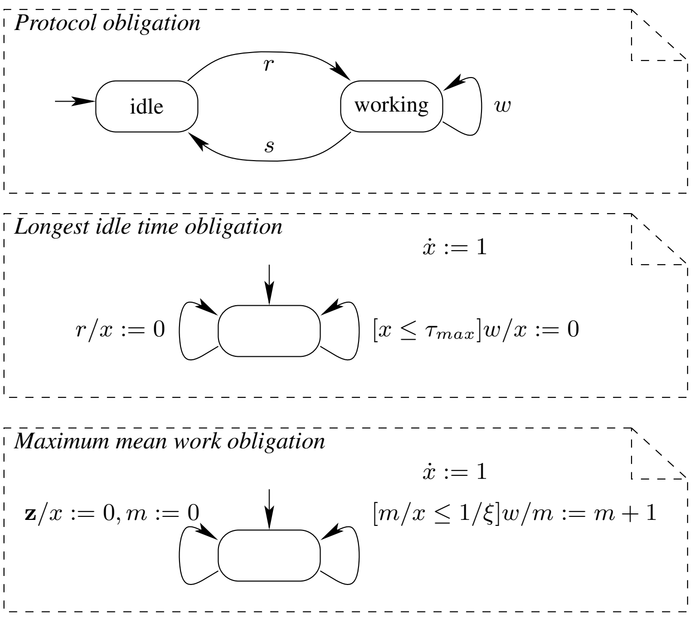

Protocol obligation
idle
working
r
s
w
Longest idle time obligation
$\dot{x} := 1$
$r/x := 0$
$[x \leq \tau_{max}]w/x := 0$
Maximum mean work obligation
$\dot{x} := 1$
$\mathbf{z}/x := 0, m := 0$
$[m/x \leq 1/\xi]w/m := m + 1$

Fig. 7. System obligations

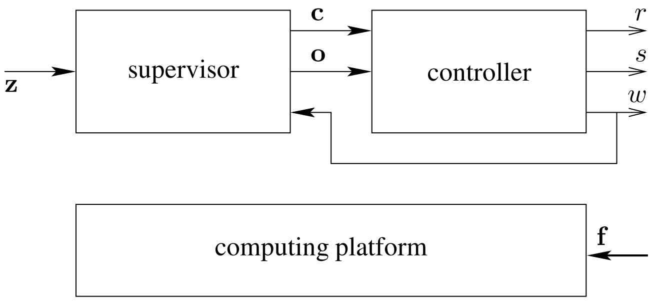
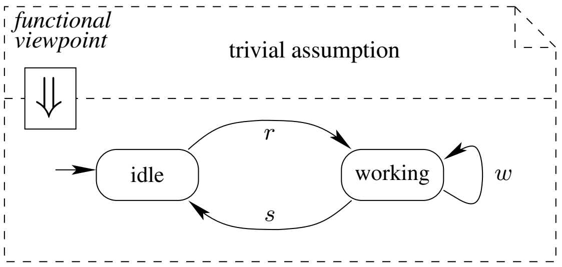


Fig. 8. The decomposition of the system controller

platform subject to "failure" $\mathbf{f}$. This component guarantees that the underlying plant receives "work" requests within a maximum amount of time. The supervisor component, instead, limits the "work" requests sent by the controller (the plant has limited capacity) by moving the controller into a "blocked" mode. This is achieved by mimicking the token bucket mechanism used for traffic shaping in communication networks: every unit of time, the supervisor accumulates a token for doing "work"; every request of "work" reduces the token amount by ξ. The supervisor monitors the flow of w's. When they get too frequent, i.e., no token is available, an "overloaded" message $\mathbf{o}$ is sent to the controller, stopping it from emitting further w requests. Only after an appropriate amount of time, long enough to let a token accumulate, does the supervisor emit a clear "c" message to the controller to enable the emitting of additional w requests. The supervisor resets the accumulated tokens when the human operator sends the "reset" message $\mathbf{z}$.

This system involves three viewpoints: functional, Quality of Service (QoS) of timed nature, and safety. The contracts for the different viewpoints are depicted in Figures 9–11. For each contract, we show its assumption (top) and promise (bottom). Assumptions are specified as observers, meaning that the corresponding diagrams define the negation



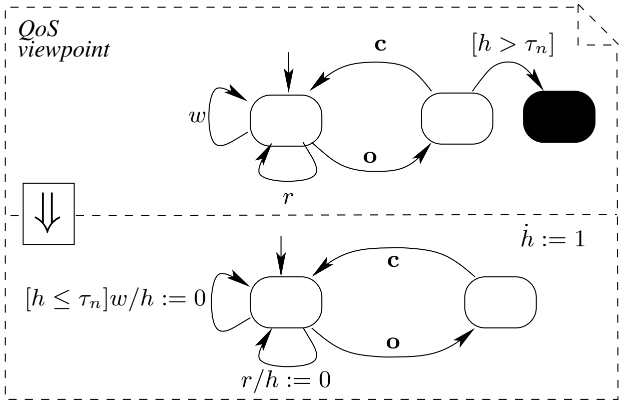


Fig. 9. The two contracts C_{funct} and C_{QoS} specifying the two viewpoints of the controller. The assumption is put on top of the promise and both are separated by the implication symbol $\boxed{\Downarrow}$.

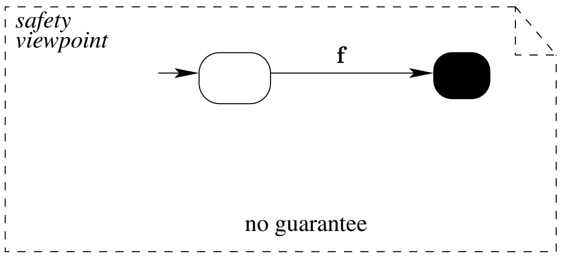


Fig. 10. The contract C_{safety} specifying the contract of the computing platform

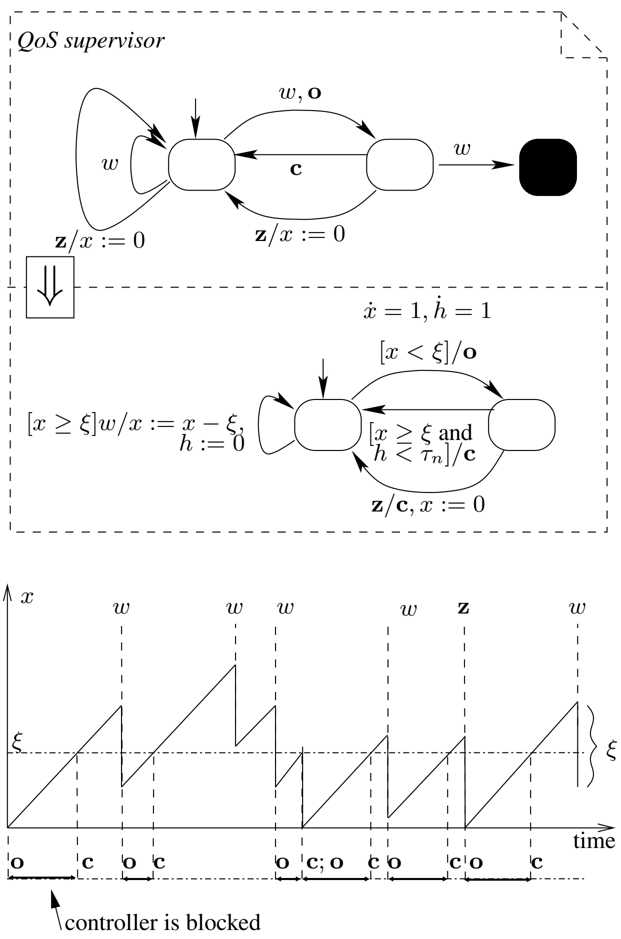


Fig. 11. Contract C_s of the supervisor and its behaviour

of these assumptions. In these diagrams, the circles filled in black denote not accepting states.

Figure 9 depicts the set of contracts associated to the controller. The first contract C_{funct} describes the functional aspect under trivial assumption. The promise of the contract C_{QoS} indicates that there exist two modes: nominal, corresponding to normal operation, and blocked, in which w's are not emitted. Contract C_{QoS} relates to timing. This contract assumes that, if the controller is blocked, it will move to the nominal mode (by receiving an event $\mathbf{c}$) at most τ_n seconds after the last "work" request. If not, the observer will move to a non accepting state. When in nominal mode, the controller guarantees that the time interval between two successive "work" requests is at most τ_n seconds. The timer h is dedicated to computing the elapsed time for both assumption and promise.

Contract C_{safety}, shown in Figure 10, is attached to the computing platform and asserts an assumption of no failure. The failure $\mathbf{f}$ is abstracted and considered as input to the platform itself. The promise is not provided and can be thought as "any" possible behavior (i.e., the "universe"). If a failure event arrives, then the assumption moves to a non accepting state, meaning that nothing could be guaranteed about the provided behavior of the platform. This kind of contracts is useful to introduce assumptions without altering the guarantees of the system.

Figure 11 depicts the QoS contract for the supervisor, which is in charge of avoiding system collapse that may occur when an excessive amount of "work" is supplied to the plant. The assumption says that no w must occur when the system is in the overloaded state. The promise is specified in terms of a hybrid automaton. This hybrid automaton uses two clocks x and h bound to physical time, thus satisfying the differential equation $\dot{x} = 1, \dot{h} = 1$. Timer x is used to implement the token bucket mechanism, while timer h is used to guarantee that a w request will be delayed by at most τ_n seconds. When action w occurs, timer x decreases and, if w occurs too frequently, in the long range x eventually reaches ξ, which causes the emission of message $\mathbf{o}$ and switches the mode to "blocked", where latency is at most the smallest between ξ and τ_n. At some point, when x is again greater than ξ, a cleaning message $\mathbf{c}$ is sent to the controller to switch to mode "nominal". It is guaranteed that the sending of $\mathbf{c}$ is at most τ_n seconds after the last "work" command. It is also guaranteed that, if the operator sends a reset by an event $\mathbf{z}$, then the system resets the timer value x and after ξ seconds the system turns back to its nominal mode.

Contract conformance to the system obligations

As introduced in Section 3.2, system obligations are compositional, i.e., if a contract C_1 conforms to a system obligation B_1 and a contract C_2 conforms to a system obligation B_2, then the parallel composition between C_1 and C_2 conforms to the system obligation $B_1 \cap B_2$. This property allows us to "allocate" system obligations (see Fig. 7) to components in order to check conformance of the corresponding contracts separately, thereby reducing the complexity of the verification task. In order for this check to be successful it is necessary that the system obligation's interface be part of the allocated component's interface. For example, the protocol obligation relates r, s and w, that are outputs of the controller component (see Fig.8). Hence, we can allocate the protocol obligation to the controller for the conformance check. Similarly, the longest idle time obligation relates r and w, so that it can also be allocated to the controller component for the conformance check.

Conversely, the maximum mean work obligation relates $\mathbf{z}$ and w. Hence, this obligation can either be allocated to the supervisor, because $\mathbf{z}$ and w are supervisor's inputs, or to the parallel composition of the supervisor with the controller, because w is also a controller's output. In the former case, the verification task is simpler, because it does not require the parallel composition of the supervisor and the controller. Nonetheless, the conformance check may fail in this case because w is controlled by the controller and not by the supervisor, so that the parallel composition might in the end be necessary to verify conformance to the system obligation.

To illustrate how the conformance check works, we show that the controller's contract conforms to the protocol and the longest idle time obligations. Let us first consider

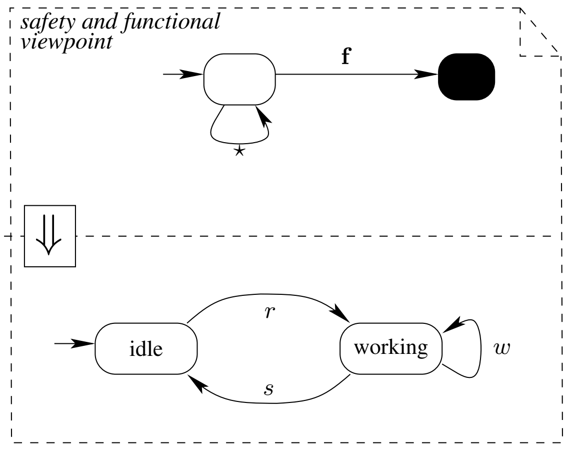


Fig. 12. Composed contract of the functional and safety viewpoints of the system controller

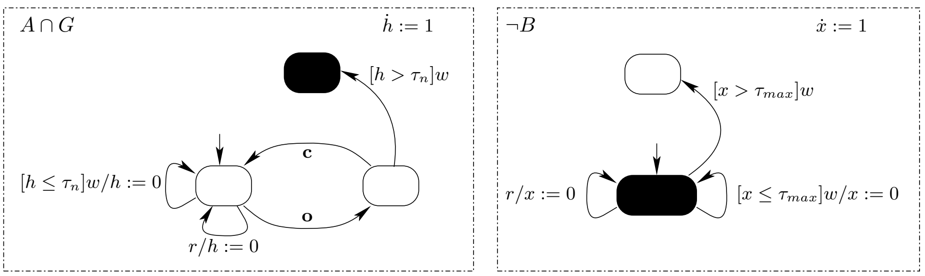


Fig. 13. Controller's contract obligation and negation of the corresponding system obligation

the protocol obligation. Observe that the controller's promise (Fig. 9) is equal to the protocol obligation. Conformance of a contract to a system obligation requires that the contract's promise subject to the contract's assumption (also called the contract obligation) is contained in the system obligation. In formulas: $A \cap G \subseteq B$, where (A, G) denotes the contract and B denotes the system obligation. Since the controller's promise is equal to the protocol obligation, then the controller's contract conforms to the protocol obligation for any assumptions. This shows that the controller's contract conforms to the protocol obligation even after composition with the safety viewpoint (Figs. 10 and 12).

Let us now consider the longest idle time obligation. The conformance check of the controller's contract to this obligation is not as trivial as in the case of the protocol obligation. To verify conformance in this case we need to compute the contract obligation $A \cap G$ and check the containment relation with the longest idle time obligation. To do so we can check that $A \cap G \cap \neg B = \emptyset$. The contract obligation and the negation of the longest idle time obligation are shown in Fig. 13. To compute the negation of the longest idle time obligation, we first complete the obligation's specification with its non-accepting states (not shown for clarity reasons). Since the resulting automaton is deterministic, its negation can be computed by exchanging accepting and non-accepting states. If we now take the intersection of the two automata shown in Fig. 13 we obtain an automaton that has no accepting states if we assume $\tau_n \leq \tau_{max}$, representing therefore the empty assertion. This proves that conformance is met.

6 Conclusion

We have presented mathematical foundations for a contract-based model for embedded systems design. Our generic mathematical model of contract supports "speculative design". This is achieved by focusing on component behavior, via compositions of contracts, with which diverse (functional and non-functional) aspects of the system can be expressed. This enabled a formalization of the whole process of component and multiple viewpoint composition through the general mechanism of contract fusion. A key contribution of our approach is that the incremental consideration of components and viewpoints can be handled with flexibility — whether through a component or a viewpoint centric methodology. The formalism and the design methodology has been illustrated through a multi viewpoint example.

Future work includes the development of effective algorithms to handle contracts, coping with the problems raised by complementation. Taking complements is a delicate issue: hybrid automata are not closed under complementation; in fact, no model class is closed under complementation beyond deterministic automata. To account for this fact, various countermeasures can be considered.

First, the designer has the choice to specify either E or its complement $\neg E$ (e.g., by considering observers). However, the parallel composition of contracts requires manipulating both E and its complement $\neg E$, which is the embarrasing case. To get compact formulas, our theory was developed using canonical forms for contracts, systematically. Not enforcing canonical forms provides room for flexibility in the representation of contracts, which can be used to avoid manipulating both E and $\neg E$ at the same time. A second idea is to redefine an assertion as a *pair* $(E, \bar{E})$, where $\bar{E}$ is an approximate complement of E, e.g., involving some abstraction. In doing so, one of the two characteristic properties of complements, namely $E \cap \bar{E} = \emptyset$ or $E \cup \bar{E} = \top$, do not hold. However, either necessary of sufficient conditions for contract dominance can be given. The above techniques are the subject of ongoing work and will be reported elsewhere.

Acknowledgments

The authors would like to acknowledge the entire SPEEDS team for their contribution to the project and to the ideas presented in this paper.

References

1. Damm, W.: Embedded system development for automotive applications: trends and challenges. In: Proceedings of the 6^{th} ACM & IEEE International conference on Embedded software (EMSOFT 2006), Seoul, Korea, October 22–25 (2006)

2. Butz, H.: The Airbus approach to open Integrated Modular Avionics (IMA): technology, functions, industrial processes and future development road map. In: International Workshop on Aircraft System Technologies, Hamburg (March 2007)
3. Sangiovanni-Vincentelli, A.: Reasoning about the trends and challenges of system level design. Proc. of the IEEE 95(3), 467–506 (2007)
4. Damm, W.: Controlling speculative design processes using rich component models. In: Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), St. Malo, France, June 6–9, pp. 118–119 (2005)
5. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
6. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
7. Lamport, L.: win and sin: Predicate transformers for concurrency. ACM Transactions on Programming Languages and Systems 12(3), 396–428 (1990)
8. Back, R.J., von Wright, J.: Contracts, games, and refinement. Information and communication 156, 25–45 (2000)
9. Back, R.J., von Wright, J.: Refinement Calculus: A systematic Introduction. Graduate Texts in Computer Science. Springer, Heidelberg (1998)
10. Dill, D.L.: Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits. ACM Distinguished Dissertations. MIT Press (1989)
11. Wolf, E.S.: Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution. PhD thesis, Department of Computer Science, Stanford University (October 1995)
12. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering, pp. 109–120. ACM Press, New York (2001)
13. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
14. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: Proceedings of the 13^{th} Annual Symposium on Foundations of Software Engineering (FSE 2005), pp. 31–40. ACM Press, New York (2005)
15. Negulescu, R.: Process spaces. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877. Springer, Heidelberg (2000)
16. Passerone, R.: Semantic Foundations for Heterogeneous Systems. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 (May 2004)
17. Burch, J., Passerone, R., Sangiovanni-Vincentelli, A.: Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In: Proceedings of the 2^{nd} International Conference on Application of Concurrency to System Design, Newcastle upon Tyne, UK, June 25–29 (2001)
18. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the Association for Computing Machinery 31(3), 560–599 (1984)
19. Engelfriet, J.: Determinacy $\rightarrow$ (observation equivalence = trace equivalence). Theoretical Computer Science 36, 21–25 (1985)
20. Brookes, S.D.: On the relationship of CCS and CSP. In: Díaz, J. (ed.) ICALP 1983. LNCS. vol. 154. Springer, Heidelberg (1983)
21. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 17(12), 1217–1229 (1998)