Contracts for Systems Design: Methodology and Application cases

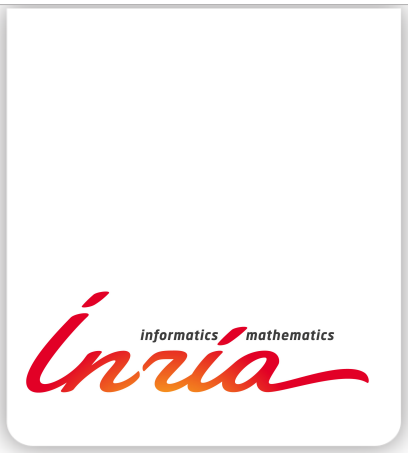
Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, Kim Guldstrand Larsen

Contracts for Systems Design: Methodology and Application cases

Albert Benveniste , Benoît Caillaud, Dejan Nickovic
Roberto Passerone, Jean-Baptiste Raclet
Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli
Werner Damm, Tom Henzinger, Kim Larsen

RESEARCH REPORT
N° 8760
July 2015
Project-Teams Hycomes

ISSN 0249-6399 ISRN INRIA/RR--8760--FR+ENG



Contracts for Systems Design: Methodology and Application cases

Albert Benveniste *, Benoît Caillaud, Dejan Nickovic
Roberto Passerone, Jean-Baptiste Raclet
Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli
Werner Damm, Tom Henzinger, Kim Larsen

Project-Teams Hycomes

Research Report n° 8760 — July 2015 — 63 pages

This work was funded in part by the European STREP-COMBEST project number 215543, the European projects CESAR of the ARTEMIS Joint Undertaking and the European IP DANSE, the Artist Design Network of Excellence number 214373, the TerraSwarm project funded by the STARnet phase of the Focus Center Research Program (FCRP) administered by the Semiconductor Research Corporation (SRC), the iCyPhy program sponsored by IBM and United Technology Corporation, the VKR Center of Excellence MT-LAB, and the German Innovation Alliance on Embedded Systems SPES2020.

Albert Benveniste and Benoît Caillaud are with INRIA, Rennes, France; Dejan Nickovic is with Austrian Institute of Technology (AIT), Roberto Passerone is with University of Trento, Italy, Jean-Baptiste Raclet is with IRIT-CNRS, Toulouse, France, Philipp Reinkemeier and Werner Damm are with Offis and University of Oldenburg, Alberto Sangiovanni-Vincentelli is with University of California at Berkeley, USA, Tom Henzinger is with IST Austria, Klosterneuburg, Kim G. Larsen is with Aalborg University, Danmark

* INRIA, Rennes, France. corresp. author: Albert.Benveniste@inria.fr

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Abstract: Recently, *contract based design* has been proposed as an "orthogonal" approach that can be applied to all methodologies proposed so far to cope with the complexity of system design. Contract based design provides a rigorous scaffolding for verification, analysis and abstraction/refinement. Companion paper [11] proposes a unified treatment of the topic that can help in putting contract-based design in perspective. This paper complements [11] by further discussing methodological aspects of system design with contracts in perspective and presenting two application cases.
The first application case illustrates the use of contracts in requirement engineering, an area of system design where formal methods were scarcely considered, yet are stringently needed. We focus in particular to the critical design step by which sub-contracts are generated for suppliers from a set of different viewpoints (specified as contracts) on the global system. We also discuss important issues regarding certification in requirement engineering, such as consistency, compatibility, and completeness of requirements.
The second example is developed in the context of the AUTOSAR methodology now widely advocated in the automotive sector. We propose a contract framework to support schedulability analysis, a key step in AUTOSAR methodology. Our aim differs from the many proposals for compositional schedulability analysis in that we aim at defining sub-contracts for suppliers, not just performing the analysis by parts—we know from companion paper [11] that sub-contracting to suppliers differs from a compositional analysis entirely performed by the OEM. We observe that the methodology advocated by AUTOSAR is in contradiction with contract based design in that some recommended design steps cannot be refinements. We show how to circumvent this difficulty by precisely bounding the risk at system integration phase. Another feature of this application case is the combination of manual reasoning for local properties and use of the formal contract algebra to lift a collection of local checks to a system wide analysis.

Key-words: system design, component based design, contract, interface.

Contrats pour la conception de systèmes: méthodologie et exemples d'application

Résumé : La conception de système constitue une étape clé pour la conception des avions, des trains, des voitures, etc. La complexité croissante des ces systèmes, largement due au logiciel, est source de retards et dépassements de coût. Les "bonnes pratiques" ne suffisent pas à régler ce problème et de nouvelles approches sont nécessaires. La conception fondée sur des modèles, complétée par la conception par niveaux et par composants, constituent un premier progrès. Récemment, une approche originale a été proposée, qui peut s'appliquer à toutes les méthodologies ci-dessus: la *conception par contrats.* De nombreux résultats existent dans ce domaine mais il manquait une vision unifiée qui mette en perspective des approches apparemment différentes telles que les contrats hypothèse/garantie ou les interfaces. L'article [11] propose une telle vision unifiée.

Le présent article accompagne l'article [11] de considérations méthodologiques approfondies et l'illustre par deux cas d'application.

Le premier porte sur l'utilisation de contrats pour l'ingéniérie des exigences, un domaine où les approches formelles n'ont pas encore vraiment trouvé leur place, mais sont cependant nécessaires. Notre cas d'application aprofondit en particulier l'étape difficile où l'on produit des sous-contrats pour les fournisseurs, à partir des exigences de niveau système. Nous étudions également comment utiliser les contrats pour formaliser des notions importantes dans le processus de certification, comme "consistence", "compatibilité", et "complétude" des exigences.

Le second exemple se situe dans le cadre d'AUTOSAR, un standard de conception largement utilisé dans le secteur de l'automobile. On propose un formalisme de contrats pour l'analyse d'ordonnancements, une étape critique et délicate de la méthodologie AUTOSAR.

Mots-clés : conception des systèmes, composant, contrat, interface.

Contents

1 Introduction

The reader is referred to companion paper [11] for a general introduction motivating contract based design. We only summarize in this brief introduction the contributions of this paper.

In Section 2 we complement the sections of companion paper [11] dealing with methodology. We review the existing answers to system complexity, such as layered and component based design, model based design and virtual integration, platform based design, and for each of them we briefly indicate how they would benefit from using contracts.

The first application case, presented in Section 3, illustrates the benefit of using contracts in requirement engineering. We develop and study requirements for a simple parking garage. Its top-level specification comprises several viewpoints, each one consisting of a requirement table. We pay attention to responsibilities by properly identifying assumptions regarding the context of use, and guarantees offered by the system if properly used. We then study the critical design step consisting in producing sub-contracts for each supplier, following an architecture of sub-systems that differs from the top-level architecture—a frequently encountered situation. We go beyond the state-of-the-art by proposing a *synthesis* method and algorithm, by which the sub-contracts are automatically derived, from the top-level contract and the (SysML-like) topological description of the sub-systems architecture. We discuss the use of contracts in formally establishing properties of the requirements such as consistency, compatibility, and completeness. Despite this is a simple example, it is yet much too complex to be dealt with by hand. A Proof of Concept tool was used to support our development. The contract framework used for this study is the Modal Interfaces, extensively developed in companion paper [11].

Our second application case, presented in Section 4, addresses a key part of the AUTOSAR development process—AUTOSAR is the methodology used today in the automotive industry. AUTOSAR advocates a design methodology by which the functions, structured into tasks, are first designed independently of the computing and communication infrastructure, assuming a virtual AUTOSAR run time environment. We study the key step by which time budgets are then allocated to tasks and computing resources are assigned. Lack of formal support in AUTOSAR methodology makes this step difficult today. We show the benefit of using contracts for this step. To this end, we develop an adaptation of the Assume/Guarantee contracts of companion paper [11] that we call *scheduling contracts*. Our study illustrates the semi-formal/semi-manual use of contracts.

2 Methodological issues in contract based design

The discussion in this section complements the two methodological sections 1 and 2 of our companion paper [11], by building on and extending the analysis of the methods, theories and tools covered there. We will first discuss in more details the role of the methodologies that have been introduced to cope with system complexity. Then, we will cover those methodological aspects which are most relevant to our case studies. Requirement engineering is the primary target of contract based design, so we discuss it in detail. Contracts are also useful in supporting virtual integration and deployment. One important point is that, by addressing early stages of system design, contracts must support domains in which automatic reasoning does not apply (e.g., due to issues

of decidability). Contract based design does not need to be fully automatic, however. Combining manual local reasoning with automatic lifting of local to system wide properties is sensible and well supported by contracts. These are the topics we are going to develop next.

2.1 Complexity of Systems and System-wide Optimization

As introduced in our companion paper [11], several approaches have been developed by research institutions and industry to cope with the exponential growth in systems complexity. Of particular interest to the development of embedded controllers and systems are layered design, component-based design, the V-model process, model-based development (MBD), virtual integration and Platform-Based Design (PBD). There are two basic principles followed by these methods: abstraction/refinement and composition/decomposition. Abstraction and refinement are processes that relate to the flow of design between different layers of abstraction (vertical process) while composition and decomposition operate at the same level of abstraction (horizontal process). Layered design, the V-model process, and model-based development focus on the vertical process while component-based design deals principally with the horizontal process. PBD combines the two aspects in a unified framework and hence subsumes them. It can be used to integrate the other methodologies. Contracts are ideal tools to solidify both vertical and horizontal processes providing the theoretical background to support formal methods.

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, e.g., the AUTOSAR standard[1] in the automotive sector, and the ARINC[2] standard in the avionic domain. As an example, consider the AUTOSAR standard. 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 and target hardware supports decoupling 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 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,

[1] http://www.autosar.org/
[2] http://www.aeec-amc-fsemc.com/standards/index.html

be used to match the boundaries of either organizational units within a company, or to define interfaces between different organizations in the supply chain.

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" characterized by 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. Another issue 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 some classes of contract models.

Layered and component-based design are well supported by the introduction of contracts. The theory discussed in the companion paper was deliberately construed to comply with the vertical and horizontal decomposition strategies, thus making the transition from a component-centric to a contract-centric approach seamless. Indeed, it has been shown how component models can easily be lifted to a contract model given sufficient compositionality properties [11, 8].

The V-shaped process model: 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 originally developed for defense applications by the German DoD.[3] Its characteristic V-shape splits the product development process into a *design* and an *integration* phase. Specifically, following product level requirement analysis, subsequent steps would first evolve a functional architecture supporting product level requirements. Sub-functions are then re-grouped taking into account re-use and product line requirements into a logical architecture, whose modules are typically developed by different subsystem suppliers. The realization of such modules often involves mechanical, hydraulic, electrical, and electronic system design. 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. 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.

Contracts provide an orthogonal added value with regard to the V-shaped process model. Contract based design does not impose any particular process. It only qualifies each design step with regard to system integration but does not impose any particular sequence of steps. It is therefore fully compliant with the V-shaped process model, as well as with other kinds of processes.

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

Model-Based Design: Model-based design (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 include SysML[4] [39] and/or AADL [40] for system level modeling, Catia and Modelica [26] for physical system modeling, Matlab-Simulink [31] for control-law design, and UML[5] [14, 37] Scade [13] and TargetLink for detailed software design. 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 Matlab-Simulink, model-based analysis including verification of compliance of requirements and specification models, model-based test-generation, rapid prototyping, and virtual integration testing.

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 [38] profile for real-time system analysis, the SPEEDS HRC metamodel [41] and the Metropolis semantic meta-model [7, 19]. 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 using a suitable algebra. 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.[6] 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. The Vanderbilt University group [32] has evolved an embedded software design methodology and a set of tools based on MDD. The generic modeling environment (GME) [32] provides a framework for model transformations enabling easy exchange of models between tools and offers sophisticated ways to support syntactic (but not semantic) heterogeneity.

Model-based development has reached significant maturity by covering nearly all aspects of the system (physics, functions, computing resources), albeit not yet in a fully integrated way. To offer a real added value, any newly proposed technology should be rich enough for encompassing all these aspects as well. Contract-based design offers, in large part, this desirable feature.

Virtual Integration: Rather than "physically" integrating a system from subsystems at integration stages, model-based design allows systems to be virtually integrated based on the models of their subsystem and the architecture specification of the system. 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. Such virtual integration thus allows detecting potential integration problems up front, in the early design phases. Virtual system integration is often a

[4] http://www.omg.org/spec/SysML/
[5] http://www.omg.org/spec/UML/
[6] www.cesarproject.eu

source of heterogeneous system models, such as when realizing an aircraft function through the combination of mechanical, hydraulic, and electronic systems. Heterogeneous composition of models with different semantics was originally addressed in Ptolemy [23], Metropolis [19, 15], and in the SPEEDS meta-model of heterogeneous rich components [18, 10, 12], albeit with different approaches. 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.

Virtual integration and virtual modeling is a step beyond basic model-based development, by offering an integrated view of all the above different aspects, e.g., physics + functions + performances. Contract-based design is well-suited to virtual integration as it supports the fusion of different systems aspects.

Platform Based Design: In Platform-Based design [19, 20, 44], 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 computation and communication library components, models of the components that represent a characterization in terms of performance and other non-functional parameters, and the rules that determine how the components can be assembled and how the functional and non-functional characteristics can be computed. Then, a platform represents a family of designs that satisfies a set of platform-specific constraints [6]. 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).

Contracts offer the theory and methodological support for both the successive refinement aspect, the composition aspect and the mapping process of PBD allowing formal analysis and synthesis processes.

2.2 Requirement capture and engineering

Requirements are the mean by which an OEM interacts with its supplier chain, on both a legal and technical perspective. For this reason, requirement engineering is the area of choice for contract based design. Therefore, we devote to this design activity a special discussion.

Requirements are nowadays typically expressed in the form of DOORS sheets.[7] Such sheets combine informal text statements with or without figures, formal or informal models, or semi-formal or formal statements from various specification languages.

Distinguishing and properly identifying the roles and foremost the *responsibilities* of the different actors in contributing to the system design is therefore essential in requirement engineering. This naturally leads to adopting *Assume/Guarantee* reasoning as a foundational paradigm. As we have seen in the companion paper [11], all known contract theories rely, either explicitly or implicitly, on a clean distinction between the systems guarantees and the assumptions about the environment specifying the legal context of use for the guarantees to hold. This identification of responsibilities, which lead to the A/G-paradigm, has a number of consequences for requirement engineering.

The meaning of requirements documents: Requirements generally aim at specifying the guarantees that are expected from the system. Defining these guarantees well is

[7] http://www-03.ibm.com/software/products/en/ratidoor

the primary focus of requirement management. Guarantees, however, generally go along with assumptions regarding the context of use. Such assumptions are most often left implicit, which is both a source of problems at system integration, and a source of dispute regarding liability between the OEM and its suppliers in case a problem occurs. Even if assumptions are carefully listed, no clean difference is generally made between how assumptions combine, versus how guarantees combine, in a requirements table. Clearly, guarantees must combine in a conjunctive way. This is indeed reflected by the common practice that "the system must pass all tests attached to the different requirements". What about assumptions? Assumptions should not get conjuncted: if the system is used in a way that violates some assumption, then the system is relieved from the set of guarantees that relied on this assumption. Other guarantees, however, remain. This reflects that assumptions must not be conjuncted. As we have seen in companion paper [11], contract theories offer the notion of *contract conjunction* to account for the difference in combining assumptions and guarantees in a requirements table. This very same concept is also valid to set the meaning of how the combination of different chapters or *viewpoints* of the systems requirements must be interpreted. Typical instances of wiewpoints are function, safety, energy, etc. These viewpoints rely on different modeling frameworks but nevertheless generally interact. This calls for supporting heterogeneous modeling in contract based design. Being generic, the meta-theory of contracts we have developed in Section 3 of the companion paper [11] is a useful step toward supporting heterogeneity.

Subsystems requirements for suppliers: Once the OEM has its system-level requirements at hand, it proceeds to defining the sets of requirements attached to the different subsystems it has identified in its architectural study. The natural question is then: does the *composition* of these subsystems requirements tables properly *refine* the system level requirements? The two words in italics are two concepts that must be properly clarified. Once more, there must be a difference in handling assumptions and guarantees while composing requirements tables attached to different subsystems. The conjunctive interpretation behind the statement "all tests must be passed" is clearly erroneous. It is acknowledged by skilled designers that part of the assumptions for a subsystem may be discharged by the guarantees offered by other subsystems. Unfortunately, this is not easily reflected in a simple aggregation of some requirements. How the subsystem's assumptions are discharged by the other subsystems actually requires the comprehensive notion of *contract composition* extensively studied in the companion paper [11]. In the same vein, assumptions and guarantees are handled differently in the notion of *contract refinement*, see the companion paper [11], which again rules out the naive "all tests must be passed" discipline when confronting subsystems requirements to system level requirements. See in particular the Section 3 of this paper where the parking garage example in requirement engineering is presented.

The added value of contracts in requirements engineering: We will illustrate this added value in our two case studies in two different ways:

- The correctness of subsystems requirements against system level requirements will be *verified* in the AUTOSAR case study, see Section 4, for the special viewpoint of task scheduling.
- The Parking Garage case study of Section 3 will propose a more prospective *synthesis* technique, by which subsystems contracts are formally derived from system level contracts and an architecture specification (à la SysML).

2.3 From informal to semi-formal to formal contracts

Unlike model checking and formal verification in general, contract based design does not need to be fully automatic. Indeed, the following is sensible and useful in practice:

1. Performing *manual* proofs or analyses on local parts of the system—due to issues of undecidability or excessive computational complexity. This of course implies that the designer is willing to take responsibility for their validity.

2. Lifting a large collection of local contract properties to system-wide guarantees and assumptions, *automatically*.

The point is that step 2 can exhibit a large combinatorial complexity, which deserves assistance from some automatic engine. In contrast, manual reasoning on "local subsystems" (step 1) is acceptable and can be reasonably trusted and argued in certification processes. The AUTOSAR case study of Section 4 is an illustration of this.

3 Contracts in requirement engineering

In this section, we discuss the use of contracts for requirement engineering. We illustrate our purpose by the means of a small example, a car parking system, that is representative of early requirements capture. The following issues arise in requirement capture. For each of them, we briefly indicate how our example illustrates them:

1. The top-level system specification is captured in a table or document collecting different kinds of requirements expressed using different formalisms (constrained natural language, boilerplate requirements [4], all sorts of automata theoretic formalisms, scenario languages, logics). Different formalisms may be used for different kinds of requirements in a same specification.

   In our example, we illustrate this by blending textual requirements written in constrained English with tiny automata, expressing elementary behavioural properties.

2. The requirements document is often structured into chapters describing various aspects of the system.

   In our example of car parking system, there is a clear separation between the specification of how the gates should behave, and of how the payment subsystem should proceed.

3. The current practice is that assumptions are often implicit, and even when they are explicitly stated, the pairing with the guarantees is missing. We clarify all of this in our application example. Some requirements are under the responsibility of the system under development; they contribute to specifying the *guarantees* that the system offers. Other requirements are not under the responsibility of the system; they contribute to defining the *assumptions* regarding the context in which the system should operate as specified. Requirement documents should be structured in such a way that each guarantee is paired with a subset of assumptions explaining the operational context under which this guarantee is expected to hold.

In our example, both assumptions and guarantees are handled as first class citizens and the pairing between them is explicitly stated.

We now move to the presentation of the example.

3.1 The car parking system, informal presentation

We begin with the top-level requirements and then we discuss how to map the requirements onto an architecture and sub-contract them to several independent suppliers.

3.1.1 Top-level requirements

The system under specification is a car parking subject to payment of a fee before exiting. At its most abstract level, the requirements document comprises three chapters **gate**, **payment**, and **supervisor**, see Table 1. The **gate** chapter collects the common requirements regarding one or several entry and exit gates. These common requirements will then be instantiated to entry and exit gates.

gateforall $g \in \{\text{entry}, \text{exit}\}$
$R_{g,1}$: *"vehicles shall not pass when gate is closed", see Fig. 1*
$R_{g,2}$: *?vehicle_pass is forbidden next to ?vehicle_pass*
$R_{g,3}$: !gate_open is forbidden next to !gate_open, and
!gate_close is forbidden next to !gate_close
$(R_{g,1}, R_{g,2}) \longrightarrow R_{g,3}$
payment
supervisor

Table 1: The top-level specification, with chapter **gate** expanded; requirements written in roman are guarantees and requirements written in *italics* are assumptions. The last line $(R_{g,1}, R_{g,2}) \longrightarrow R_{g,3}$ indicates the pairing between guarantees and associated assumptions. Quotes indicate requirements informally expressed in natural language and formalized as automata.

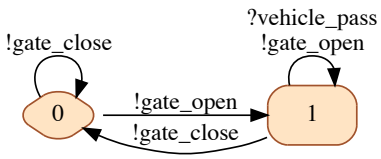


Figure 1: i/o-automaton formalizing requirement $R_{g.1}$. Prefix "?" indicates an input and prefix "!" indicates an output.

Focus on the "**gate**" chapter. It consists of the three requirements shown on Table 1. Requirement $R_{g.1}$ is best described by means of an i/o-automaton, shown in Figure 1—In Table 1, we only provide a specification in natural language. Suppose that some requirement says: "?gate_open never occurs". This is expressed by having no mention of ?gate_open in the corresponding i/o-automaton—this way of doing assumes that the alphabet of actions of the i/o-automaton is explicitly given. The other two requirements

are written using constrained natural language, which can be seen as a boilerplate style of specification. Prefix "?" indicates an input and prefix "!" indicates an output.

The first two requirements are not under the responsibility of the system, since they rather concern the car driver. Thus it does not make sense to include them as part of the guarantees offered by the system. Should we remove them? This would be problematic. If drivers behave the wrong way unexpected things can occur for sure. The conclusion is that 1) we should keep requirements $R_{g.1}$ and $R_{g.2}$, and 2) we should handle them differently than $R_{g.3}$, which is a guarantee offered by the system. Indeed, $R_{g.1}$ and $R_{g.2}$ can only be seen as assumptions and $R_{g.3}$ is the guarantee stated under the operational context of $R_{g.1}$ and $R_{g.2}$. This pairing between a guarantee and its related assumptions is denoted using the graph notation in the last line of Table 1.[8] So far we have specified **gate** as a list of requirements. Requirement $R_{g.1}$ specified as an i/o-automaton can be considered formal. Requirements $R_{g.2}$ and $R_{g.3}$ are formulated in constrained natural language and are ready for subsequent formalization (e.g., using i/o-automata too).

Are we done? Not yet! We need to formalize what it means to have a collection of requirements, and what it means to distinguish assumptions from guarantees. Similarly, we must formalize what it means to combine different chapters of a requirement document. The answer was announced in point 3 of the discussion at the beginning of this Section 3: requirement documents are conjunctions of *causal pairs*, consisting of a guarantee together with the set of assumptions required for this guarantee to be in force. It turns out that this is realized by using the conjunction of the contracts encoding the causal pairs {guarantee, related assumptions}, as we shall see in Section 3.2.2. Thus, if $\mathscr{C}$ is the top-level specification of the car parking system, we have $\mathscr{C} = \mathscr{C}_{\mathbf{gate}} \wedge \mathscr{C}_{\mathbf{payment}} \wedge \mathscr{C}_{\mathbf{supervisor}}$. The following issues then arise regarding this top-level specification. First of all, since a conjunction operator is involved in the construction of $\mathscr{C}$, there is a risk of formulating *inconsistent* (i.e., contradicting) requirements. Second, are we sure that the top-level specification $\mathscr{C}$ is *complete*, i.e., precise enough to rule out undesirable implementations? One good way of checking for completeness would consist in being able to execute or simulate this top-level specification $\mathscr{C}$ and to observe if unexpected behaviors can occur. All of this is developed in Section 3.2.4.

3.1.2 Sub-contracting

Having the top-level specification $\mathscr{C}$ at hand, the designer then specifies an architecture *à la* SysML, as shown on Figure 2. Some comments are in order regarding this architecture.

The considered instance of car parking system consists of one entry gate, one exit gate, and one payment machine. Compare with the top-level specification of Table 1. The latter comprises a generic gate, a payment machine, and a supervisor, each one with its set of requirements. In contrast, the architecture of Figure 2 involves no supervisor—the latter is meant to be distributed among the two gates. So the system architecture does not match the structure of the top-level specification—this is a typical situation encountered in practice.

The next step in the design consists in sub-contracting the development of each of the three sub-systems of the architecture of Figure 2. This amounts to specifying three

[8] Such a pairing can be easily implemented as hyperlinks in requirement engineering tools.

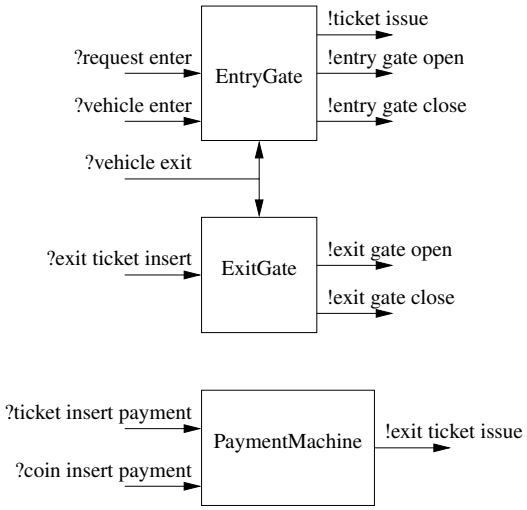


Figure 2: System architecture as specified by the designer.

sub-contracts $\mathscr{C}_{\mathbf{EntryGate}}$, $\mathscr{C}_{\mathbf{ExitGate}}$, and $\mathscr{C}_{\mathbf{PaymentMachine}}$, such that[9]:

$$\mathscr{C}_{\mathbf{EntryGate}} \otimes \mathscr{C}_{\mathbf{ExitGate}} \otimes \mathscr{C}_{\mathbf{PaymentMachine}} \preceq \mathscr{C} \tag{1}$$

Recall that refinement $\preceq$ in (1) means that any implementation of the left hand side is also a valid implementation of the top-level $\mathscr{C}$ and any legal operational use (we call it: environment) of $\mathscr{C}$ is also legal for the left hand side. Then, the contract composition operator $\otimes$ ensures that each supplier can develop its sub-system based on its own sub-contract only, and, still, integrating the so designed sub-systems yields a correct implementation of the top-level specification. Guessing such sub-contracts is a difficult task. This is investigated in the next section.

3.2 Formalization using contracts

Despite being simple and small, the car parking system example quickly becomes complex for reasons that are intrinsic to the formal management of requirements. The MICA[10] tool was used to develop it [17]. In this development, requirements were written in constrained English language and then translated into Modal Interfaces—while the presented translation is manual, automatic translation could be envisioned.[11] Once this is completed, contracts are formally defined and the apparatus of contracts can be used. In particular, important properties regarding certification can be formally defined and checked, e.g., consistency, compatibility, correctness, and completeness. In addition, support is provided for turning top-level requirements into an architecture of sub-systems, each one equipped with its own requirements. The latter can then be

[9] We refer readers not well acquainted with contract-based reasoning or these notations to section 2 of the companion paper [11].

[10] http://www.irisa.fr/s4/tools/mica/

[11] In fact, the contract specification languages proposed in the projects SPEEDS [10] and CESAR (http://www.cesarproject.eu/) are examples of translations from a constrained English language to a formal models of contracts similar to Modal Interfaces.

submitted to independent suppliers for further development. We begin with the presentation of the contract framework we will be using, namely Modal Interfaces.

3.2.1 The contract framework

We use Modal Interfaces (with variable alphabet) as developed in Section 5.3 of the companion paper [11] and subsequent ones. There are three main reasons for this choice:

1. By offering the *may* and *must* modalities, Modal Interfaces are well suited to express mandatory and optional behaviors in the specification, which we consider important for requirements engineering.

2. Being large sets of requirements structured into chapters, requirements documents are a very fragmented style of specification. Only Modal Interfaces offer the needed support for an accurate translation of concepts such as "set of requirements", "set of chapters", together with a qualification of who is responsible for each requirement (the considered component or sub-system versus its environment).

3. As we shall see, at the top-level, conjunction prevails. However, as soon as the designer refines the top-level requirements into an architecture of sub-systems, composition enters the game. Turning a conjunction of top-level requirements into a composition of sub-systems specifications thus becomes a central task. Only Modal Interfaces provide significant assistance for this.

Overall, the problem considered in the above claim 3 can be stated as follows. The designer begins with some system-level contract $\mathscr{C}$, which is typically specified as a conjunction of viewpoints and/or requirements. The designer guesses some topological architecture by decomposing the alphabet of actions of $\mathscr{C}$ as

$$\Sigma \quad = \quad \bigcup_{i\in I} \Sigma_i \quad , \quad \Sigma_i \quad = \quad \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}} \tag{2}$$

such that composability conditions regarding inputs and outputs hold. Once this is done, we expect our contract framework to provide help in generating a decomposition of $\mathscr{C}$ as

$$\bigotimes_{i\in I} \mathscr{C}_i \quad \preceq \quad \mathscr{C} \tag{3}$$

where sub-contract $\mathscr{C}_i$ has alphabet $\Sigma_i = \Sigma_i^{\text{in}} \uplus \Sigma_i^{\text{out}}$. Guessing architectural decomposition (2) relies on the designer's understanding of the system and how it should naturally decompose—this typically is the world of SysML. Finding decomposition (3) is, however, technically difficult in that it involves behaviors. The algorithmic means that were presented in Section 5.5 of the companion paper [11] provide the due answer. In this car parking system, we use the special operation of *restriction* that was developed in that section.

3.2.2 Top-level requirements

We first explain how the specification of "**gate**" in Table 1 translates into Modal Interfaces. Observe that each requirement $R_{g.j}$ of Table 1 is a sentence that can be formalized as an i/o-automaton, see Figure 1 for such a formalization of requirement $R_{g.1}$. In general, each (chapter of a) requirement document D collects requirements (assumptions

or guarantees) for which the input/output status of the different actions is consistent, meaning that no action exists in D that is an input in some requirement and an output in another one. Furthermore, document D takes the form of a set of *causal pairs* $\mathbf{A}_i \rightarrow \mathbf{G}_i$:

$$\mathrm{D} \quad = \quad \{\mathbf{A}_i \rightarrow \mathbf{G}_i \,|\, \mathbf{G} = \uplus_{i\in I}\mathbf{G}_i \text{ and } \mathbf{A}_i \subseteq \mathbf{A}\} \tag{4}$$

where the subset $\mathbf{G}$ of guarantees of D decomposes as $\mathbf{G} = \uplus_{i\in I}\mathbf{G}_i$ and $\mathbf{A}_i \subseteq \mathbf{A}$ is the subset of the assumptions of D on which each guaranty belonging to $\mathbf{G}_i$ relies. Referring to Table 1 for **gate**, there is only one guarantee which requires the two assumptions to hold: $\{\mathrm{R}_{g,1}, \mathrm{R}_{g,2}\} = \mathbf{A} \rightarrow \mathbf{G} = \{\mathrm{R}_{g,3}\}$. The translation of document D specified as in (4) is explained next, where $G_i = \{\mathrm{R}_{\mathrm{ij}} \mid j{\in}J_i\}$ and $A_i = \{\mathrm{R}_{\mathrm{ik}} \mid k{\in}K_i\}$.

Rules 1 (translating individual guarantees) *For each causal pair $\mathbf{A}_i \rightarrow \mathbf{G}_i$, we start from a description of each guarantee $\mathrm{R}_{\mathrm{ij}} \in \mathbf{G}_i$ as an i/o-automaton. This i/o-automaton is translated to a Modal Interface by applying the following rules:*

- $\mathbf{R}_1^{\mathrm{G}}$ *: Unless otherwise explicitly stated, transitions labeled by an output action are given a may modality. The rationale is that the default semantics for guarantees is "best effort". The only exception is when the requirement specifies that an output action is mandatory, e.g., by having a "must" in the sentence.*
- $\mathbf{R}_2^{\mathrm{G}}$ *: Transitions labeled by an input action of the considered system are given a must modality. The rationale is that implementations may not refuse this input action in this state.*

Applying these rules to R_{ij} yields a modal interface called G_{ij}. □

Performing this for the single guarantee $\mathrm{R}_{\mathrm{g.3}}$ of **gate** yields the Modal Interface shown in Figure 3.

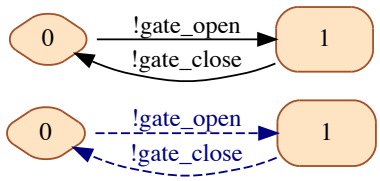


Figure 3: Translating the guarantee $\mathrm{R}_{\mathrm{g.3}}$ of **gate** as an i/o-automaton (top) and then as a Modal Interface $G_{\mathbf{gate}}$ (bottom) using Rules 1.

Rules 2 (translating individual assumptions) *For each causal pair $\mathbf{A}_i \rightarrow \mathbf{G}_i$, we start from a description of each assumption $\mathrm{R}_{\mathrm{ik}} \in \mathbf{A}_i$ as an i/o-automaton. This i/o-automaton is translated to a Modal Interface by applying the following sequence of rules:*

1. *We complement the status input/output in every assumption R_{ik}, thus taking the point of view of the environment; we call the result $\overline{\mathrm{R}}_{ik}$;*
2. *Having done this we apply Rules 1 to each $\overline{\mathrm{R}}_{ik}$. So far this yields, for each $\overline{\mathrm{R}}_{ik}$, a Modal Interface $\overline{A}_{ik}$ that must be satisfied by every environment; complementing backward the status input/output of each action yields a modal interface A_{ik}.* □

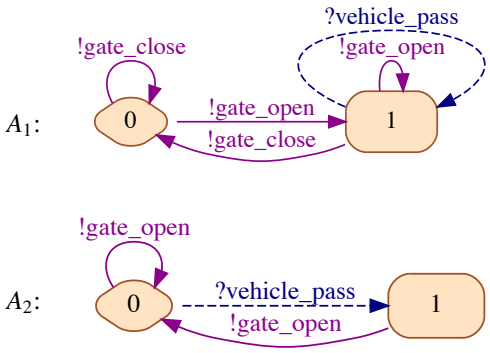


Figure 4: Translating the assumptions of **gate** using Rules 2.

Performing this for the assumptions $R_{g.1}$ and $R_{g.2}$ of **gate** yields the Modal Interfaces A_1 and A_2 shown in Figure 4.

Rules 3 (combining) *The modal interface $\mathscr{C}_i$ representing the causal pair $\mathbf{A}_i \rightarrow \mathbf{G}_i$ is then computed as indicated in Section 5.7 of the companion paper* [11]. *Finally, the top-level contract $\mathscr{C}$ is the conjunction of the contracts associated to each causal pair: $\mathscr{C} = \bigwedge_{i \in I} \mathscr{C}(\mathbf{A}_i, \mathbf{G}_i)$.*

Performing this for the whole chapter **gate** yields the Modal Interface shown in Figure 5.

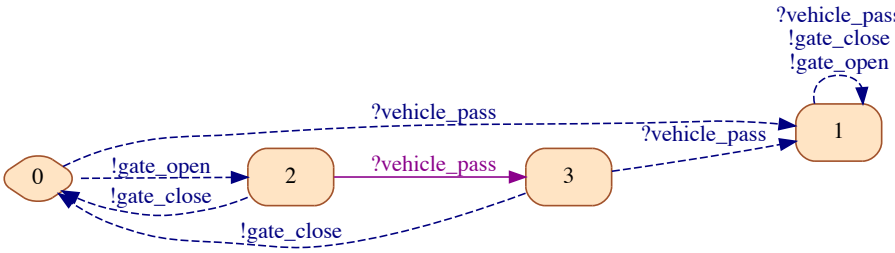


Figure 5: Chapter **gate** of the top-level requirements document translated into a Modal Interface $\mathscr{C}_{\mathbf{gate}}$. Remark state 1 is universal, meaning that any behavior may be allowed after reaching this state.

Some comments are in order:

- *Regarding the guarantees offered by the component:* Allowed outputs possess a *may* modality, which reflects that Guarantees specify what the component may deliver. Other actions are forbidden.
- *Regarding the context of operation:* Legal inputs to the **gate** (e.g., vehicle_pass when exiting state "2") have a *must* modality. This complies with the intuition that the component should not refuse legal stimuli from its environment. Violation of the contract by its environment occurs when an illegal input is submitted by the environment (vehicle_through when exiting state 0 or state 3). As a consequence, the whole contract gets relaxed by moving to the special state "1" from

which any action is allowed—such a state is often called a "top" state. This top state resulted from computing the quotient.

The same procedure applies to all chapters **gate**, **payment**, and **supervisor**, of the top-level textual specification, shown in Table 2 (it is an expansion of Table 1).

gate(x) where $x \in$ {entry, exit}
$R_{g.1}(x)$: *"vehicles shall not pass when x_gate is closed", see Fig. 1*
$R_{g.2}(x)$: *after ?vehicle_pass ?vehicle_pass is forbidden*
$R_{g.3}(x)$: after !x_gate_open !x_gate_open is forbidden and
after !x_gate_close !x_gate_close is forbidden
$(R_{g.1}(x), R_{g.2}(x)) \longrightarrow R_{g.3}(x)$
payment
$R_{p.1}$: *"user inserts a coin every time a ticket is inserted and only then", Fig. omitted*
$R_{p.2}$: *"user may insert a ticket only initially or after an exit ticket has been issued", Fig. omitted*
$R_{p.3}$: "exit ticket is issued after ticket is inserted and payment is made and only then", Fig. omitted
$(R_{p.1}, R_{p.2}) \longrightarrow R_{p.3}$
supervisor
$R_{g.1}$(entry) (*assumption borrowed from* **gate**(entry))
$R_{g.1}$(exit) (*assumption borrowed from* **gate**(exit))
$R_{g.2}$(entry) (*assumption borrowed from* **gate**(entry))
$R_{g.2}$(exit) (*assumption borrowed from* **gate**(exit))
$R_{s.1}$: initially and after !entry_gate close !entry_gate open is forbidden
$R_{s.2}$: after !ticket_issued !entry_gate open must be enabled
$R_{s.3}$: "at most one ticket is issued per vehicle entering the parking and tickets can be issued only if requested and ticket is issued only if the parking is not full", see Fig 7
$R_{s.4}$: "when the entry gate is closed, the entry gate may not open unless a ticket has been issued", Fig. omitted
$R_{s.5}$: "the entry gate must open when a ticket is issued", Fig. omitted
$R_{s.6}$: "exit gate must open after an exit ticket is inserted and only then", Fig. omitted
$R_{s.7}$: "exit gate closes only after vehicle has exited parking", Fig. omitted
$(R_{g.1}(\text{entry}), R_{g.1}(\text{exit}), R_{g.2}(\text{entry}), R_{g.2}(\text{exit})) \longrightarrow (R_{s.1}, \ldots, R_{s.7})$
supervisor: unformalized requirements
$R_{s.8}$: "*the ticket inserted in the* **gate** *must be physically the same as the one issued by the* **payment** *machine*"
$R_{s.9}$: "a vehicle cannot exit without having paid a ticket"
$(R_{g.1}(\text{entry}), \ldots, R_{g.2}(\text{exit}), R_{s.8}) \longrightarrow R_{s.9}$

Table 2: Top-level requirements. *Assumptions* and Guarantees are written in italics and roman, respectively. The last line of each chapter specifies the causal pairs following (4)—for this example, all assumptions are required for every guarantee. Quoted requirements are written in natural language; the corresponding formalization as a modal interface is omitted. The additional chapter, in blue, collects requirements not supported by our formalization.

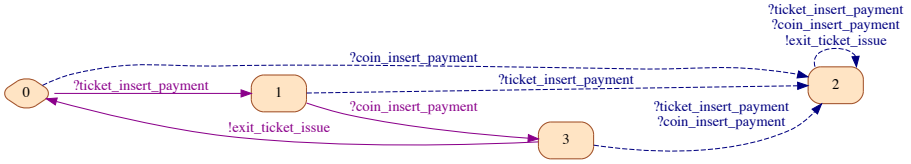


Figure 6: Chapter **payment** of the top-level requirements document translated into a Modal Interface $\mathscr{C}_{\textbf{payment}}$.

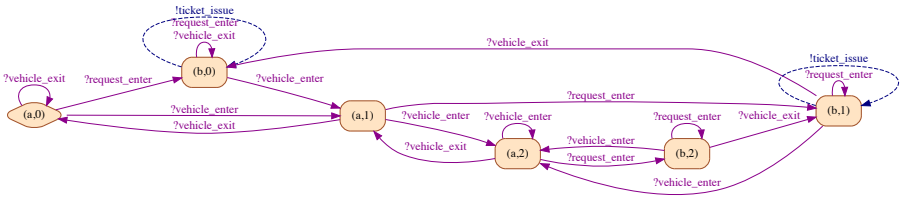


Figure 7: Modal Interface for $R_{s,3}$

Comment 1 (Requirements not supported by our formalization) This table involves a supplementary chapter for the **supervisor**, written in blue, which collects requirements that are not supported by our formalization. The reason for this lack of formal support is that infinite domains of data are beyond the reasoning capabilities of the MICA tool we are using, whereas tickets are infinitely many and must be characterized by a unique identifier. As a consequence, guarantee $R_{s,9}$ is outside the scope of our formal analysis. It is indeed a typical situation in practice, that only a subset of the requirements can be formally supported—requirements address all aspects of system specification, be they within or outside the scope of formalization. For the case of Table 2, requirement $R_{s,9}$ must be validated against the designed system, either manually, or via observer techniques. □

The Modal Interfaces encoding chapters **payment** and **supervisor** of the top-level are displayed in Figures 6 to 8. Figure 8 showing the contract associated to the **supervisor** is unreadable and the reader may wonder why we decided to put it here. We indeed wanted to show that, when contract design is performed formally and carefully, top-level contracts rapidly become complex, even for modest sets of requirements. So the formal management of requirements and their translation into formal contracts must be tool-assisted.

Finally, the whole top-level contract $\mathscr{C}$ is the conjunction of the contracts representing chapters **gate**, **payment**, and **supervisor**, of the top-level requirements document:

$$\mathscr{C} \quad = \quad \mathscr{C}_{\textbf{gate}} \wedge \mathscr{C}_{\textbf{payment}} \wedge \mathscr{C}_{\textbf{supervisor}} \tag{5}$$

Owing to the complexity of $\mathscr{C}_{\textbf{supervisor}}$ shown in Figure 8, we do not show the Modal Interface $\mathscr{C}$ formalizing the full document. Nevertheless, the latter was generated and can then be exploited as we develop next. The above specification only covers the functional viewpoint of the system. Other viewpoints might be of interest as well, e.g., regarding timing behavior and energy consumption. They would be developed with the same method and combined to the above contract $\mathscr{C}$ using conjunction.

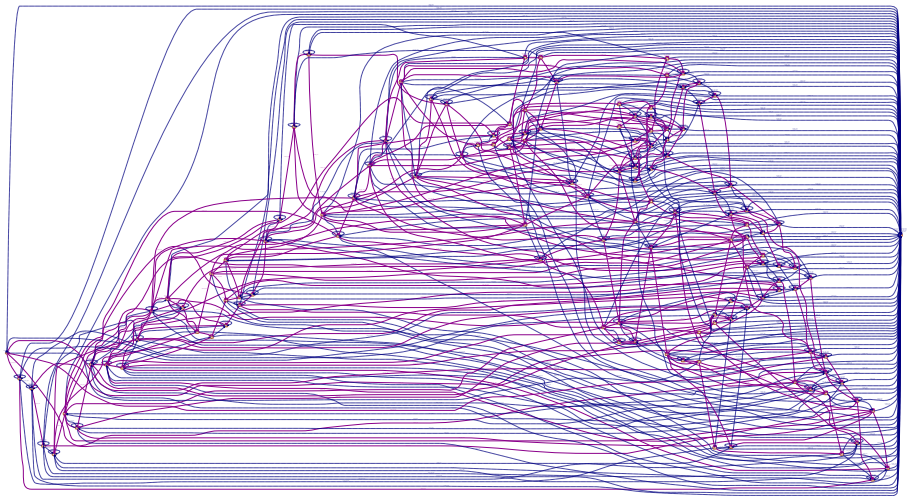


Figure 8: Chapter **supervisor** of the top-level requirements document translated into a Modal Interface $\mathscr{C}_{\textbf{supervisor}}$, for a capacity of two for the parking garage.

3.2.3 Sub-contracting

In this section, we apply the technique developed in Section 5.5 of the companion paper [11] for generating an architecture of sub-systems with their associated sets of requirements. Each sub-system can then be submitted for independent development to a different supplier. The next duty of the designer is, thus, to specify an architecture *à la* SysML shown on Figure 2.

Comment 2 (Mismatch requirements architecture vs. system architecture) Observe that the considered instance of the parking garage consists of one entry gate, one exit gate, and one payment machine. Compare with the top-level specification of Table 2. The latter comprises a generic gate, a payment machine, and a supervisor, each one with its set of requirements. In contrast, the architecture of Figure 2 involves no supervisor. The supervisor is meant to be distributed among the two gates. This mismatch between requirements architecture and system architecture is representative of real situations: it is the purpose of this application case to propose solutions for it. □

In Figure 9 we show the result of applying, to the architecture of Figure 2, the Algorithm 1 developed in Section 5.5 of companion paper [11], which yields by construction a refinement of the top-level contract $\mathscr{C}$ by a decomposition into local contracts:

$$\mathscr{C} \quad \geq \quad \mathscr{C}(\Sigma_{\text{EntryGate}}) \otimes \mathscr{C}(\Sigma_{\text{ExitGate}}) \otimes \mathscr{C}(\Sigma_{\text{PaymentMachine}}) \tag{6}$$

Local contract $\mathscr{C}(\Sigma_{\text{EntryGate}})$ is the more complex one because it involves counting. For the sake of readability, we have assumed a capacity of two. Small capacities less than 20 can be handled with enumerated methods. For larger capacities, symbolic methods must be used. Remarkably enough, *the decomposition* (6) *involves small sub-systems compared to* $\mathscr{C}_{\textbf{supervisor}}$ (Fig. 8) *and the global contract* $\mathscr{C}$*; the restriction operation is to be acknowledged for this strong reduction in size.* The main reasons is that the components have few synchronizations and that the decomposition method revealed the parallelism that is hidden in $\mathscr{C}$.

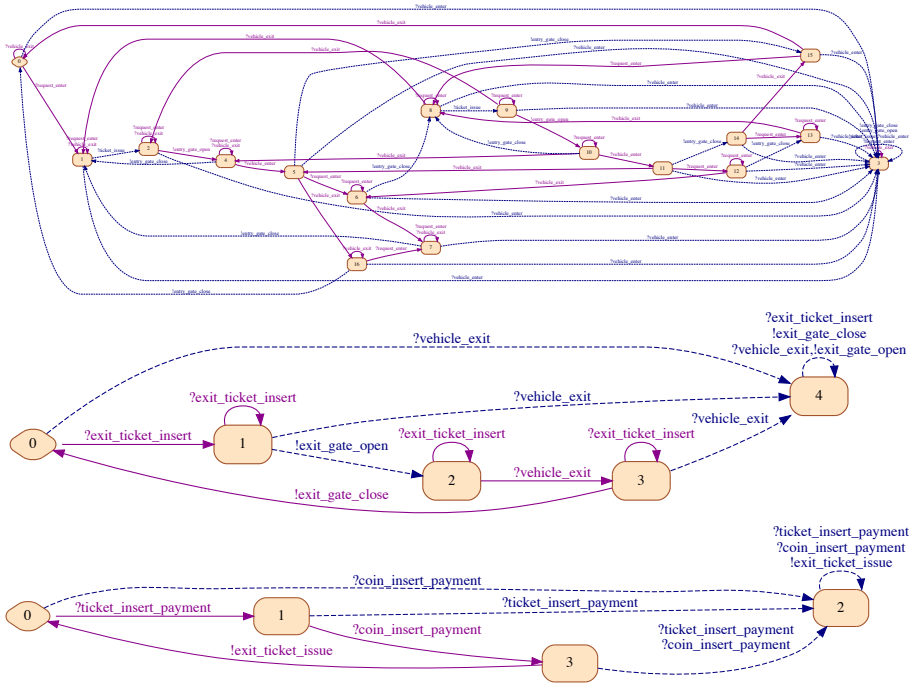


Figure 9: The three restrictions of the global contract $\mathscr{C}$ for the three sub-systems EntryGate (top), ExitGate (mid), and PaymentMachine (bottom).

Comment 3 (Comment 1, cont'd) Referring to Table 2, recall that requirement R_{s_9} must be validated against the designed system, either manually, or via observer techniques. How can this piece of manual work be combined with our formal analysis? Call $\mathscr{C}_{\textbf{supervisor\_informal}}$ the contract corresponding to the supplementary chapter of Table 2. The total top-level contract is in fact given by $\mathscr{C} \wedge \mathscr{C}_{\textbf{supervisor\_informal}}$. To check a design against this contract it is enough to check it against $\mathscr{C}$ and against $\mathscr{C}_{\textbf{supervisor\_informal}}$. The former check involves a complex contract but is tool supported. In contrast, the latter check must rely on observers but involves a simple contract. The bottom line is that contract based requirement engineering supports the combination of tool based and manual checks well. □

3.2.4 Consistency, Compatibility, Correctness, Completeness

Requirements capture and management are important matters for discussion with certification bodies. These bodies would typically assess a number of quality criteria from, e.g., the following list elaborated by INCOSE [30]: Accuracy, Affordability, Boundedness, Complexity, Completeness, Conciseness, Conformance, Consistency, Correctness, Criticality, Level, Orthogonality, Priority, Risk, Unambiguousness, and Verifiability. In this section we focus on four quality criteria that are considered central by certification authorities and are relevant to contracts, namely: Completeness, Correctness, Consistency, and Compatibility.

Consistency & Compatibility. Consistency and compatibility have been formally defined in the meta-theory, see Section 3 of companion paper [11] and Table 2 therein. In particular, those formal definitions can be formally checked. Thus, the question arises whether these formal definitions suitably reflect the common sense meaning of these terms.

According to the common meaning, a set of requirements is *consistent* if it is not self-contradicting. The intent is that there is no point in trying to implement an inconsistent set of requirements, as no such implementation is going to exist. It turns out that the existence of implementations is the formal definition of consistency according to the meta-theory, which meets the common sense interpretation of this term.

We illustrate consistency on the top-level specification of Table 2. Referring to this table, let us investigate the consistency of the set of requirements $\{R_{g.3}(\text{exit}), R_{s.6}, R_{s.7}\}$ under assumptions $\{R_{g.1}(\text{exit}), R_{g.2}(\text{exit})\}$. The following scenario is *may*-reachable for this modal interface:

1) exit_ticket_insert; 2) exit_gate_open; 3) exit_ticket_insert

After this scenario, event exit_gate_open has modality *must* in interface $R_{s.6} \wedge R_{s.7}$, whereas it has modality *cannot* in interface $R_{g.3}(\text{exit})$. Thus, the state reached after this scenario is inconsistent. Two options are possible:

1. *Return to the designer the message that the set of requirements is inconsistent, since it possesses inconsistent states.* This has the merit of leaving the designer with the entire responsibility of avoiding hidden contradictions in its requirements. On the other hand, this puts a heavy burden on the shoulders of the designer since she must have detailed understanding of its specification. Alternatively, one can

2. *Reduce the resulting interface by pruning away the inconsistent states, hoping that not all states become inconsistent.* (See Lemma 9 of Section 5.3 in companion paper [11] regarding interface reduction.) This is the option we have followed in this application example. It is supported by our tool MICA [17].

According to the common meaning, an architecture of sub-systems, as characterized by their respective specifications, is *compatible* if these sub-systems "match together", in that they can be composed and the resulting system can interact with the environment as expected—use cases can be operated as wished. As explained in Table 2 of Section 3 of companion paper [11], this is the formal definition of compatibility in the meta-theory. Again, the formal definition of compatibility meets its common sense interpretation.

Correctness. Correctness can only be defined with reference to another specification. In contract based design, correctness is not a known concept. We thus propose to specialize "correctness" as one of the following properties, depending on the design step performed (see Table 2 of Section 3 of companion paper [11] for the notations):

- "correctness" specializes as: "is a correct implementation of", written $\models^{M}$;
- "correctness" specializes as: "is a correct environment of", written $\models^{E}$;
- "correctness" specializes as: "refines", written $\preceq$.

Completeness. Completeness raises a difficulty. Although the term "completeness" speaks for itself, it cannot be formally defined what it means to be complete, for a top-level specification in the form of a set of requirements. The reason is that we lack a reference against which completeness could be checked. Hence, the only way to inspect a top-level specification for completeness is to explore it manually. The best help for doing this is to execute the specification. Thus, specifications must be *executable*. Fortunately, Modal Interfaces are executable and, this way, undesirable behaviors can be revealed.

We illustrate this on the top-level specification of Table 2, in which we change requirement $R_{s.6}$ to: "exit gate must open after an exit ticket is inserted" (by omitting "and only then"). Lack of completeness is revealed by simulation. The following scenario can occur:

1) exit_ticket_insert; 2) exit_gate_open; 3) vehicle_exit; 4) exit_gate_close; 5) exit_gate_open

where step 5) conforms the specification, due to prior step 1). This scenario shows that vehicles may exit without having inserted an exit ticket, an unwanted behavior. This reveals that the specification was not tight enough, i.e., incomplete. So far for completeness of the top-level specification.

In contrast, completeness can be formally defined when a reference $\mathscr{C}$ is available. We propose to say that $\mathscr{C}'$ is *incomplete* with reference to $\mathscr{C}$, if

1. $\mathscr{C}'$ does not refine $\mathscr{C}$, but
2. there exists $\mathscr{C}'' \preceq \mathscr{C}'$ such that $\mathscr{C}''$ is consistent, compatible, and refines $\mathscr{C}$.

The rationale for this definition is that $\mathscr{C}'$ is not precise enough but can be made so by adding some more requirements. Note that $\mathscr{C}'$ is incomplete with reference to $\mathscr{C}$ if and only if $\mathscr{C}' \wedge \mathscr{C}$ is consistent and compatible. This criterion is of particular relevance when $\mathscr{C}' = \bigotimes_{i \in I} \mathscr{C}_i$ is an architecture of sub-contracts, where $\mathscr{C}_i$ is to be submitted to supplier i for independent development.

3.3 Discussion

Requirements engineering is considered very difficult. Requirements are typically numerous and very difficult to structure. Requirements concern all aspects of the system: function, performance, energy, reliability/safety. Hence, systems engineers generally use several frameworks when expressing requirements. The framework of contracts expressed as Modal Interfaces that we have proposed here improves the situation in a number of aspects:

- It encompasses a large part of the requirements.[12]
- It can accommodate different concrete formalisms. In our example, we have blend textual requirements with requirements specified as state machines. Richer formalisms such as Stateflow diagrams can be accommodated in combination with abstract interpretation techniques—this is not developed here.
- We have shown how to offer formal support to important properties during the process of certification.

[12] According to figures that were given to us by industrials, 70-80% of the requirements can be expressed using the style we have developed here. Other requirements typically involve physical characteristics of the system or define the range for some parameters.

- We have proposed a correct-by-construction approach to the difficult step of moving from the top-level specification in the form of a requirements document, to an architecture of sub-contracts for the suppliers.[13]
- Not all requirements can be supported by a formal approach—be it contract-based or different in nature. It is the merit of our contract-based approach to allow for a semi-automatic/semi-manual handling of requirements.

4 Contracts for deployment and mapping in the context of AUTOSAR

AUTOSAR[14] is a world-wide development partnership including almost all players in the automotive domain electronics supply chain. It has been created with the purpose of developing an open industry standard for automotive software architectures. To achieve the technical goals of modularity, scalability, transferability and reusability of functions, AUTOSAR provides a common software component model and a common infrastructure based on standardized interfaces for the different layers. The AUTOSAR project has focused on the objectives of resource independence, standardization of interfaces and portability of code. While these goals are clearly of paramount importance, their achievement may not be sufficient for improving the quality of software systems and ensuring safe system integration.

As for most other embedded system, the design of car electronics involves functional as well as non-functional properties, assumptions and constraints [28]. In the AUTOSAR design flow, a large part of the effort is devoted to non-functional aspects combining:

- latencies and throughputs, which are critical in computer controlled systems, and
- the sharing of communication and computing resources and the conflicts that can result.

Getting a proper scheduling of the software components is a key step in meeting such specifications. The case study we develop in this section addresses the above issues—the detail of the functions, however, is not considered.

4.1 An illustrative design scenario

The focus of the used contract framework is the integration phase of a design process, where software components are allocated to a hardware platform. More specifically, we consider scenarios like the following: The bottom part of Figure 10 shows a target platform that is envisioned by, say, an Original Equipment Manufacturer (OEM). It consists of two processing nodes (CPU_1 and CPU_2). Suppose the OEM wants to implement two applications, characterized by contracts $\mathscr{C}_1$ and $\mathscr{C}_2$, on this architecture and delegates their actual implementation to two different suppliers. Both applications share a subset of the resources of the target platform, e.g. tasks τ_2 and τ_4 are executed

[13] The framework of Assume/Guarantee contracts that is used in Section 4 does not offer this, because local alphabets are not properly handled. In contrast, Assume/Guarantee contracts are very permissive in how they can be expressed. In particular, dataflow diagrams (Simulink) can be used to express assumptions and guarantees.

[14] http://www.autosar.org/

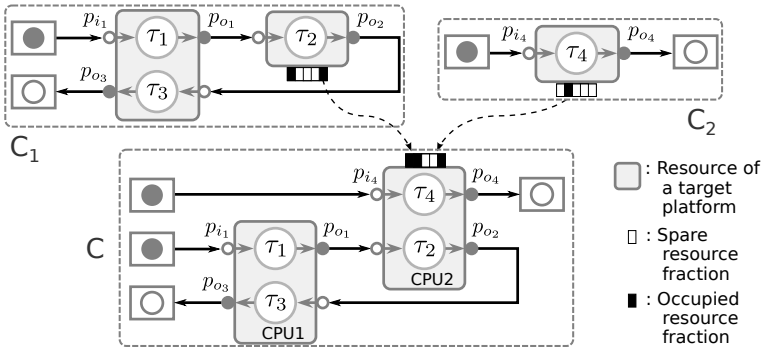


Figure 10: Exemplary Integration Scenario using Resource Segregation

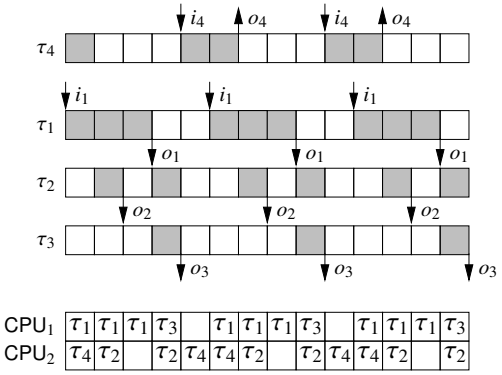


Figure 11: A sample trace showing the joint histories of the four tasks τ_1–τ_4 (top). Observe that the additional histories of the two CPU (bottom) are redundant if we know the resource allocation for each task. We show them nevertheless to highlight that no conflict occurs in this trace.

on CPU_2 after integration. Furthermore, we assume the system specification $\mathscr{C}$ shown in Figure 10 to be available from previous design phases. While some components together with their (local) contracts may also be known (e.g. in case of reuse), the OEM generally has to negotiate proper specifications with the suppliers, in our case the two contracts $\mathscr{C}_1$ and $\mathscr{C}_2$. At this point, the designer is faced with the following two issues:

1. The functions performed by the two subsystems must integrate correctly and their integration must satisfy the top-level function specification;

2. The scheduling of the software components delegated to each supplier must yield a satisfactory scheduling at system integration, meaning that timing constraints are met given the performance characteristics of the computing and communication resources, despite the two designs compete for shared resources.

For these two aspects, the design method must support independent development by each supplier while guaranteeing safe and correct integration, provided that the subsystems are correctly implemented. In this case study *we concentrate on issue 2, leaving aside issue 1* (the latter was addressed by the parking garage case study of Section 3).

Example 1 Suppose task τ_1 on the system depicted at the bottom of Figure 10 is a periodic task with period $p = 5$ and execution time $c = 3$. The two tasks τ_2 and τ_3 also have period $p = 5$. Task τ_2 depends on τ_1, i.e. is activated by τ_1, and has an execution time $c_2 = 2$. Task τ_3 depends on τ_2 and has an execution time $c_3 = 1$. Task τ_4 is also a periodic task with period $p_4 = 5$ and $c_4 = 2$. Suppose both CPUs are scheduled using a fixed priority preemptive policy, where tasks τ_1 and τ_4 have high priority on their CPU. The delay of the task-chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ depends on the activation-pattern of τ_4 and its execution time. This is illustrated in Figure 11. Once τ_1 completes execution it activates (via port p_{o_1}) τ_2, which in turn might be preempted by τ_4. Finally, τ_3, activated by τ_2, could be preempted by a subsequent instance of τ_1 resulting from another event i_1 of the periodic event stream. An excerpt of a possible trace in L is shown in Figure 11, which corresponds to the discussed scheduling scenario. Indeed, while composing the two subsystems and deploying them on the computing architecture of Figure 10, bottom, the main problem is the coupling due to the sharing of CPU_2. Observe that every port and resource has its own event tape in the component. Note that we omitted input ports connected to some output port: the intuition is that the synchronization of the subsystems' behaviors is by unifying ports with identical names. This intuition for the composition of subsystems will be indeed formalized by our forthcoming notion of abstract scheduling component, see Definition 4. □

Objectives of this application case: *In this work, we assume that a procedure performing global scheduling analysis is available. Such procedures exist for various classes of scheduling problems. Our aim is to lift such procedures to a contract framework supporting compositionality and independent development.* In the next sections we develop our contract framework for this application case. Then, we develop the application.

4.2 Scheduling Components

In this section we prepare the material for the framework of "scheduling contracts" we will be using for this application. This contract framework is a mild adaptation of the Assume/Guarantee contracts (A/G-contracts for short) considered in Section 4 of companion paper [11].

Recall that Assume/Guarantee contracts (A/G-contracts) are pairs of assumption and guarantees: $\mathscr{C} = (A, G)$. In the basic A/G-contract framework [10], A and G are assertions, i.e., sets of traces for system variables.[15] Components capturing legal implementations or environments of contracts are also modeled by assertions. Component E is a legal environment for $\mathscr{C}$ if $E \subseteq A$ and component M is an implementation for $\mathscr{C}$ if $A \times M \subseteq G$. In this writing, $\subseteq$ is simply set inclusion and component composition $\times$ is by intersection of sets of traces (assuming that the underlying set of system variables is universal and thus fixed): $A \times M =_{\text{def}} A \cap M$. One difficulty of A/G-contracts is the important notion of *saturation*: contracts (A, G) and $(A, G \cup \neg A)$, where $\neg$ is set complement, possess identical sets of legal environments and implementations, so we consider them equivalent. The second one is called *saturated* and is a canonical form for the class of equivalent contracts. Also, $M_{\mathscr{C}} = G \cup \neg A$ is the maximal implementation for this contract. Thus, we need the operations $\cup$ and $\neg$, or at least we need the operation $(A, G) \rightarrow G \cup \neg A$, which is to be interpreted as "A entails G".

[15] These are typically specified using modeling tools such as Simulink/Stateflow.

Thus, as a first step, we need the counterpart of assertions for our component framework, with the associated algebra. Ingredients of scheduling problems are: *tasks* with their *precedence conditions* reflecting data dependencies and *resource allocation*. The sets of *timed traces* we are interested in are those satisfying the scheduling constraints, plus extra quantitative properties such as period, deadline conditions, etc. Call *concrete scheduling components* the resulting model. Unfortunately, no rich algebra with the above requested operators $\subseteq, \times, \cup\neg$ exists for concrete scheduling components.

By abstracting away part of the description of task activities in traces, we slightly abstract concrete scheduling components to so-called *abstract* ones. The idea is that we keep only what is essential for capturing interactions of scheduling problems, namely: 1) trigger and release events for tasks, and 2) busyness of resources. The abstraction map binds each concrete scheduling component to its abstraction and we will show that this binding is faithful. The framework of abstract scheduling components is simple enough so we manage to equip it with the wanted operations $\subseteq, \times, \cap, \cup, \neg$. A/G-contracts for abstract scheduling components follow then easily. The rest of this section is devoted to the introduction of concrete and abstract scheduling components. Then we study the relation between them.

4.2.1 Concrete Scheduling Components

For our model of scheduling components we assume the following:

- A slotted model of real-time, in which the real line $\mathbb{R}_+$ is divided into successive discrete time slots of equal duration. Successive slots are thus indexed by using natural numbers $1, 2, 3, \ldots, n, \ldots \in \mathbb{N}$, with 0 indexing the initial conditions. In the sequel, the term "*date*" will refer to the index of the time slot in consideration.
- An underlying set $\mathcal{T}$ of *tasks*, generically denoted by the symbol τ. To describe events of interest for tasks, we consider the following alphabets:
  - the *control alphabet* $\Sigma_c = \{i, o, io, aw, sl\}$ collects the *trigger*, *completion*, *trigger-and-completion*, *awake*, and *sleeping* events, for a task; this alphabet describes the triggering and completion of tasks; since we follow a slotted model of time, triggering and completion can occur within the same slot, which is indicated by the event *io*;[16]
  - the *busyness alphabet* $\Sigma_b = \{*, \bot\}$ collecting the *busy* and *idle* events; this alphabet indicates, for a task, its status busy/idle at a given time slot.

  On top of these alphabets, we build:

$$\Sigma \quad =_{\text{def}} \quad \{(c, b) \in \Sigma_c \times \Sigma_b \mid c{=}sl \;\Rightarrow\; b{=}\bot\} \tag{7}$$

  reflecting that task τ can only be busy when it is not sleeping. The status of each task in each time slot is expressed by using alphabet Σ. This is illustrated on Figure 12.

  In addition, each task τ comes equipped with a pair $(p^t(\tau), p^c(\tau)) \in \mathcal{P}{\times}\mathcal{P}$ of *trigger* and *completion ports*, where $\mathcal{P}$ is an underlying set of *ports.* For T a set of tasks, we will consider the set $P_T =_{\text{def}} \{p^t(\tau), p^c(\tau) \mid \tau \in T\}$.

[16] Strictly speaking, statuses *aw* and *sl* add no useful information about the history of a task; these two statuses are only here for technical convenience. They are used in (7) to build our structured alphabets and they facilitate the formulation of the condition 1 characterizing behaviors in Definition 2.

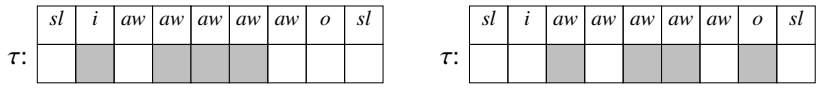


Figure 12: Two executions for task τ, with their successive time slots. The top/bottom rows illustrate the history of Σ_c and Σ_b in each execution; blank and grey stand for $\perp$ and $*$.

- An underlying set $\mathcal{R}$ of *resources*, generically denoted by the symbol r. A resource can be either available or busy with executing a given task at a given time slot. Resources can run in parallel. Each resource $r \in \mathcal{R}$ is assigned the alphabet $\Sigma_r \subseteq \mathcal{T} \cup \{0\}$ of the tasks it can run, where the special symbol 0 indicates that r is idle.

Definition 1 *A* concrete scheduling component *is a tuple* $\mathbf{M} = (K, L)$*, where:*

- $K = (T, R, \rho)$ *is the* sort *of* $\mathbf{M}$*, where:* $T \subseteq \mathcal{T}$ *is the set of* tasks*,* $R \subseteq \mathcal{R}$ *is the set of* resources*, and* $\rho : T \to R$*, the* resource allocation map*, is a partial function satisfying* $\tau \in \Sigma_{\rho(\tau)}$*. Say that tasks* τ_1 *and* τ_2 *are* non-conflicting *if they do not use the same resource:*

$$\tau_1 \|_K \tau_2 \quad \textit{if} \quad \left\{ \begin{array}{rl} \textit{either} & \rho(\tau_1) \textit{ is undefined} \\ \textit{or} & \rho(\tau_2) \textit{ is undefined} \\ \textit{or} & \rho(\tau_1) \neq \rho(\tau_2) \end{array} \right. \tag{8}$$

- *For* $\tau_1, \tau_2 \in T$*, say that* τ_1 precedes *task* τ_2*, written*

$$\tau_1 \multimap \tau_2, \tag{9}$$

if the completion port of τ_1 *coincides with the start port of* τ_2*:* $p^c(\tau_1) = p^t(\tau_2)$*. We require that this relation is circuit free and we denote by* $\preceq$ *the partial order on* T *obtained by taking the transitive closure of* $\multimap$ *and we call* $\preceq$ *the* precedence order*. The dual order between ports will also be needed: for* $p_1, p_2 \in P_T$*, say that* p_1 precedes p_2*, written*

$$p_1 \boxminus\!\!\rightarrow p_2, \tag{10}$$

if there exists $\tau \in T$ *such that* $p_1 = p^t(\tau)$ *and* $p^c(\tau) = p_2$*; relation* $\boxminus\!\!\rightarrow$ *is circuitfree if so was* $\multimap$ *and, with no risk of confusion, we also denote by* $\preceq$ *the* precedence order *on* P_T *generated by* $\boxminus\!\!\rightarrow$*.*

- $L \subseteq \Sigma_T^\omega$ *is the* language *of* $\mathbf{M}$*, where* $\Sigma_T =_{\text{def}} T \to \Sigma_\tau$ *and* A^ω *denotes the set of all infinite words over alphabet* A*. Due to the decomposition (7) of* Σ*, every word* $w \in L$ *can be equivalently seen as a pair of words* $w = (w_c, w_b)$ *describing the control and busyness history of* w*.* □

Since a word $w \in L$ yields a history for each task, it induces, by picking the resource running that task, a corresponding *resource word* w^R, such that

$$\begin{array}{l} w^R \text{ is the tuple collecting the } w^r \text{ for } r \in R, \text{ such that, for every slot } n\text{:} \\ w^r(n) = \{\tau \in T \mid \rho(\tau) = r \text{ and } w(\tau, n) = (c, b) \text{ satisfies } b = *\} \end{array} \tag{11}$$

i.e., $w^r(n)$ returns the set of tasks that resource r runs at slot n. Note that this set is not a singleton if and only if a conflict occurs at slot n regarding resource r. A possible word w of L was shown in Figure 11, together with its corresponding (non-conflicting) resource extension.

Notations: Whenever convenient, we will denote by T_K or T^K the set of tasks of sort K, and similarly for the other constituents of a sort. The events of a task τ will be denoted by i_τ, o_τ, etc. For w a word of Σ_K^ω, $T' \subseteq T$, and $R' \subseteq R$, we denote by

$$w^{T'} \text{ the } T'\text{-word of } w, \quad \text{and by} \quad w^{R'} \text{ the } R'\text{-word of } w, \tag{12}$$

obtained by projecting w to the subalphabet $\Sigma_{T'}$ and projecting the induced word w^R to the subalphabet $\Sigma_{R'}$, respectively. If $T' = \{\tau\}$ is a singleton, we simply write w^τ and similarly for R'. The reader is referred to Figure 11 for an illustration of this—occurrences of a 0 are figured by a blank. □

Not all words of L are compliant with the rules of scheduling. We characterize those compliant words in the following definition, where $\mathbf{M} = (K, L)$ denotes a concrete scheduling component:

Definition 2 (semantics of concrete scheduling component) *A* behavior *of sort K is any infinite word $w \in \Sigma_T^\omega$ satisfying the following three* scheduling conditions*:*

1. *For each task $\tau \in T$, the control word w_c belongs to the language $(sl^*.(io + i.aw^*.o))^\omega$, where $a^* =_{\text{def}} \epsilon + a + a.a + a.a.a + \ldots$ is the Kleene closure starting at the empty word. Informally, the two events i and o alternate in w, with i occurring first; io is interpreted as the immediate succession of two i and o events at the same time slot. Call nth* epoch *of τ in w the n-th occurrence of a pattern of w belonging to $(io + i.aw^*.o)$.*

2. *$\tau_1 \preceq \tau_2$ implies the following: for every $n \geq 1$, the nth occurrence of event o_{τ_1} must have occured in w strictly before i_{τ_2}—in words, τ_2 can only start after τ_1 has been completed;*

3. *w is* non-conflicting*, meaning that, for any two conflicting tasks τ_1 and τ_2 belonging to T (cf. (8)), it never happens that w^{τ_1} and w^{τ_2} are non-idle at the same time slot.*

The semantics of $\mathbf{M}$ *is the sublanguage $[\![\mathbf{M}]\!] \subseteq L$ consisting of all behaviors of K belonging to L. Say that $\mathbf{M}$ is* schedulable *if $[\![\mathbf{M}]\!] \neq \emptyset$.* □

Observe that, due to the above Condition 2, tasks related by precedence conditions possess identical logical clocks—consequently, if they are specified to be periodic, their periods must be equal. This is not required for tasks not related by precedence conditions.

Comment 4 (roles of K and L) The pair $\mathbf{M} = (K, L)$ can be seen as the specification of a global scheduling problem. The sort K fixes the set of tasks and their precedence conditions, the set of resources, and the allocation of tasks to resources. The language L can serve to specify additional aspects of this scheduling problem, including task durations and/or minimum time interval between successive activation calls for a task.

Comment 5 (role of $[\![\mathbf{M}]\!]$) Semantics $[\![\mathbf{M}]\!]$ can be seen as the maximally permissive solution of the scheduling problem stated by $\mathbf{M}$. *Recall that, in this work, we assume that the procedure for computing the semantics of a scheduling component is available—such procedures exist for various classes of scheduling problems. Our aim is to lift such procedures to a contract framework supporting compositionality and independent development.* □

The class of concrete scheduling components is easily equipped with a parallel composition:

Definition 3 (composition of concrete scheduling components) *Say that* $\mathbf{M}_1$ *and* $\mathbf{M}_2$ *are* composable *if their allocation maps* ρ_1 *and* ρ_2 *coincide on* $T_1 \cap T_2$ *and the relation* $\multimap_1 \cup \multimap_2$ *on* $T_1 \cup T_2$ *is cycle free. If* $\mathbf{M}_1$ *and* $\mathbf{M}_2$ *are composable, their* composition $\mathbf{M}_1 \underline{\times} \mathbf{M}_2 =_{\text{def}} ((T, R, \rho), L)$ *is defined as follows:*

$$\begin{array}{rcl} T & = & T_1 \cup T_2 \\ R & = & R_1 \cup R_2 \\ \forall \tau \in T : \rho(\tau) & = & \textbf{if } \tau \in T_1 \textbf{ then } \rho_1(\tau) \textbf{ else } \rho_2(\tau) \\ L & = & \mathbf{pr}^{-1}_{T \to T_1}(L_1) \cap \mathbf{pr}^{-1}_{T \to T_2}(L_2) \end{array}$$

where $\mathbf{pr}_{T \to T_i}()\,, i = 1, 2$*, denotes the projection from* T *to* T_i *and* $\mathbf{pr}^{-1}$ *is its inverse.* □

Of course, the key to understand the meaning of composition $\underline{\times}$ is the construction of the semantics $[\![\mathbf{M}_1 \underline{\times} \mathbf{M}_2]\!]$, where the scheduling problem is solved. To each scheduling component $\mathbf{M} = ((T, R, \rho), L)$, we associate the following scheduling component where L is replaced by the semantics $[\![\mathbf{M}]\!]$ of $\mathbf{M}$:

$$[\mathbf{M}] \quad =_{\text{def}} \quad ((T, R, \rho), [\![\mathbf{M}]\!]) \tag{13}$$

The following result holds:

Lemma 1 *If* $\mathbf{M}_1$ *and* $\mathbf{M}_2$ *are composable, then* $[\![\mathbf{M}_1 \underline{\times} \mathbf{M}_2]\!] = [\![\, [\mathbf{M}_1] \underline{\times} [\mathbf{M}_2] \,]\!]$.

Proof: Since sorts are unchanged, from $\mathbf{M}_i$ to $[\mathbf{M}_i]$, the right hand side is well defined. For the same reason, the conditions listed in Definition 2 are the same when selecting the behaviors of $\mathbf{M}_1 \underline{\times} \mathbf{M}_2$ and of $[\mathbf{M}_1] \underline{\times} [\mathbf{M}_2]$.
To construct the semantics of $\mathbf{M}_1 \underline{\times} \mathbf{M}_2$:

1. pick all pairs $(w_1, w_2) \in L_1 \times L_2$ such that $\mathbf{pr}_{T_1 \to T_1 \cap T_2}(w_1) = \mathbf{pr}_{T_2 \to T_1 \cap T_2}(w_2)$;
2. for such a pair (w_1, w_2) and $\tau \in T_i, i = 1, 2$, set $w(\tau) = w_i(\tau)$; this define a word $w \in \Sigma_T^\omega$;
3. keep only the words w that are behaviors of sort (T, R, ρ).

To construct the semantics of $[\mathbf{M}_1] \underline{\times} [\mathbf{M}_2]$:

1. pick all pairs $(w_1, w_2) \in L_1 \times L_2$;
2. keep only the pairs (w_1, w_2) such that w_i is a behavior of sort (T_i, R_i, ρ_i);
3. keep only the pairs $(w_1, w_2) \in L_1 \times L_2$ such that $\mathbf{pr}_{T_1 \to T_1 \cap T_2}(w_1) = \mathbf{pr}_{T_2 \to T_1 \cap T_2}(w_2)$;
4. for such a pair (w_1, w_2) and $\tau \in T_i, i = 1, 2$, set $w(\tau) = w_i(\tau)$; this define a word $w \in \Sigma_T^\omega$;

5. keep only the words w that are behaviors of sort (T, R, ρ).

Since precedence relations $\multimap_1 \cup \multimap_2$ and $\multimap$ coincide, the combination of steps 2 and 5 of the second procedure coincides with step 3 of the first procedure. □

Comment 6 (the need for a more abstract mathematical framework) As announced in the introductory discussion of Section 4.2, the model of concrete scheduling component is too complex and detailed as a model of component on top of which contracts can be built. In particular, the consideration of sorts raises a difficult typing problem and is an obstruction against getting the constructs required for a universe of components. Moreover, scheduling component $\mathbf{M}$ and its semantics $[\![\mathbf{M}]\!]$ are related in a complex way, through Definition 2. This makes it difficult to define the operations we need on components, particularly $\subseteq$ and $\cup\neg$ (in turn, parallel composition $\times$ was easy to define as we have seen). The notion of *abstract scheduling component* we develop in the forthcoming section will overcome these difficulties. Abstract scheduling components capture the architecture aspect of Figure 10, namely: ports carrying start and completion events of tasks, and resources—tasks themselves are, however, ignored. □

4.2.2 Abstract Scheduling Components

Definition 4 (abstract scheduling component) *An* abstract scheduling component *is a language*

$$M \subseteq \mathcal{V}^\omega\,, \quad \textit{where} \quad \mathcal{V} =_{\text{def}} \left(\{0,1\}^{\mathcal{P}}\right) \times \left(\prod_{r\in\mathcal{R}} \Sigma_r\right)$$

Recall that Σ_r is the alphabet of tasks that can be executed by resource r, see the beginning of Section 4.2.1. We will freely interpret $\{0, 1\}$ as the Boolean domain and symbol "1" indicates the occurrence of an event. Abstract scheduling components come equipped with the following algebra:

- The Boolean algebra $\cap, \cup, \neg$ (set complement), and the inclusion $\subseteq$ on sets;
- A *parallel composition* by intersection: $M_1 \times M_2 =_{\text{def}} M_1 \cap M_2$.

Thus, abstract scheduling components offer all the algebra required for a universe of components on top of which A/G-contracts can be built. It is therefore interesting to map concrete to abstract scheduling components.

4.2.3 Mapping Concrete to Abstract Scheduling Components

Recall that, for $K = (T, R, \rho)$ a sort, we denote by $P_T =_{\text{def}} p^t(T) \uplus p^c(T) \subseteq \mathcal{P}$ the set of ports used by T, see the beginning of Section 4.2.1. Then, we set

$$\mathcal{V}_K \;=_{\text{def}}\; \left(\{0,1\}^{P_T}\right) \times \left(\prod_{r\in R} \Sigma_r\right) \tag{14}$$

Definition 5 (Mapping concrete to abstract scheduling components) *Each concrete scheduling component* $\mathbf{M} = (K, L)$ *is mapped to a unique abstract scheduling component* $[\![\mathbf{M}]\!]^{\mathsf{A}}$ *called its* abstract semantics, *defined as follows:*

1. *Pick any* $w \in [\![\mathbf{M}]\!]$, *see Definition 2;*
2. *Denote by* $\pi_T(w)$ *the word over* $\{0,1\}^{P_T}$ *obtained from* w *as follows:*

   $$\forall p \in P_T,\ \textit{define}\ {}^\bullet p = \{\tau \in T \mid p^c(\tau) = p\} \quad \textit{and} \quad p^\bullet = \{\tau \in T \mid p^t(\tau) = p\},$$

   the sets of anterior and posterior tasks of p. *Put the* n*th event of* p, nondeterministically*:*

- *after the $n-1$st event of p,*
- *when or after every task belonging to ${}^\bullet p$ has completed for the nth time in w, and*
- *strictly before every task belonging to $p^\bullet$ has started for the nth time in w.*

If ${}^\bullet p = \emptyset$, then the first condition is not considered and similarly if $p^\bullet = \emptyset$.

3. *Denote by $\pi_R(w)$ the word over alphabet $\prod_{r\in R} \Sigma_r$ defined as follows:*

   (a) *For every time slot n and every resource $r \in R$, set*

   $$\pi_R(w)(r,n) = \tau \text{ if and only if } w(\tau,n) = (c,b) \text{ satisfies } b = * \text{ and } \rho(\tau) = r.$$

   This part of word $\pi_R(w)$ represents the "positive history" of w, i.e., the use of the resources belonging to R by tasks belonging to T;

   (b) *We complement $\pi_R(w)$ by describing the "negative history" of w, consisting of a description of all the possibilities left, for tasks not belonging to T, in using resources from R:*

   in all slots of $\pi_R(w)(r,n)$ that are kept idle after step 3a, we set $\pi_R(w)(r,n) = \tau'$ where $\tau' \in \Sigma_r, \tau' \notin T$ is chosen nondeterministically.

Then, with reference to the sort $K = (T,R,\rho)$ of $\mathbf{M}$, we set:

$$\eta_K(w) \quad =_{\text{def}} \quad (\pi_T(w), \pi_R(w)) \ \in \ \mathcal{V}_K^\omega \tag{15}$$

4. *Finally, we define*

$$[\![\mathbf{M}]\!]^{\mathsf{A}} \ =_{\text{def}} \ \mathbf{pr}^{-1}_{\mathcal{V}^\omega \to \mathcal{V}_K^\omega}\Big(\big\{\eta_K(w) \,\big|\, w \in [\![\mathbf{M}]\!]\big\}\Big) \ \subseteq \ \mathcal{V}^\omega$$

where the quantification ranges over $w \in [\![\mathbf{M}]\!]$ and all instances of nondeterministic choices in step 2, and $\mathbf{pr}_{\mathcal{V}^\omega \to \mathcal{V}_K^\omega}$ denotes the projection, from $\mathcal{V}^\omega$ to $\mathcal{V}_K^\omega$.

Step 2 of this construction is sound since w is a behavior in the sense of Definition 2. Step 2 is the key step since it transforms a max-plus type of parallel composition (every task waits for all its preceding tasks having completed before starting) into a dataflow connection where data are communicated through the shared ports. The data communicated are the events carried by the ports. These events occur nondeterministically after all preceding tasks have completed for the nth time and before all succeeding tasks start for the nth time.

Step 3 complements the actual history of each task of $\mathbf{M}$ by an explicit description of all possibilities that are left to other scheduling components in using resources shared with $\mathbf{M}$. The reason for doing this is that this allows to capture the interleaved use of shared resources by different components, by a simple parallel composition by intersection.

The above construction is illustrated in Figure 13. When hiding the tasks sitting inside the boxes, the architecture shown on Figure 10 is a dataflow representation of $[\![\mathbf{M}]\!]^{\mathsf{A}}$: in interpreting this figure, one should consider that each task is free to start any time after it has received its triggering event, and free to wait for some time before emitting its completion event.

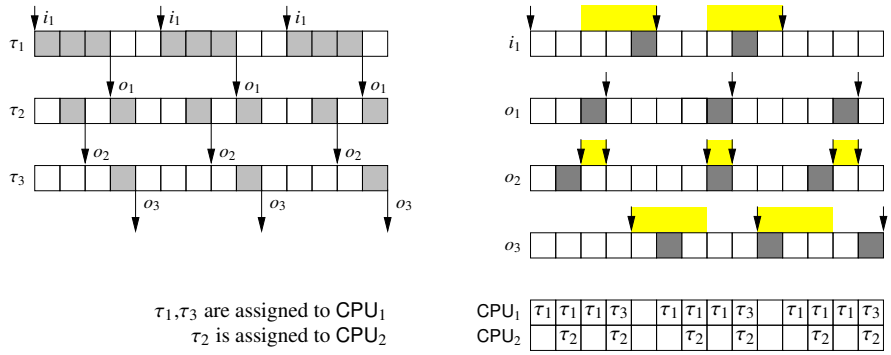


Figure 13: Showing a concrete behavior of $\mathbf{M}$ (left, with reference to Figure 11) and a corresponding abstract behavior of $[\![\mathbf{M}]\!]^{\mathsf{A}}$ (right), by using $\mathcal{P} = \{i_1, o_1, o_2, o_3\}$ as underlying alphabet of ports. On the second diagram, blanks figure the slots left free for any external task to run on the referred resource. The yellow rectangles indicate the room for nondeterministic choices; bounds of these rooms are figured by pointing arrows; where such arrow is missing, the corresponding rectangle is unbounded on that side.

Lemma 2 *The mapping* $\mathbf{M} \to [\![\mathbf{M}]\!]^{\mathsf{A}}$ *satisfies the following properties:*

1. *Schedulability is preserved in that* $[\![\mathbf{M}]\!] \neq \emptyset$ *if and only if* $[\![\mathbf{M}]\!]^{\mathsf{A}} \neq \emptyset$*;*
2. *For every* $r \notin R$*, the set* $\{v(r) \mid v \in [\![\mathbf{M}]\!]^{\mathsf{A}}\}$ *is the free language* $(\mathcal{T} - T)^{\omega}$*.*

The special property 2 is not preserved under the Boolean set algebra. Therefore, the mapping $\mathbf{M} \to [\![\mathbf{M}]\!]^{\mathsf{A}}$ is not surjective.

4.2.4 Faithfulness of the mapping

Consider two concrete scheduling components $\mathbf{M}_1$ and $\mathbf{M}_2$, where $\mathbf{M}_i = (K_i, L_i)$ and $K_i = (T_i, R_i, \rho_i)$. Checking the inclusion $[\![\mathbf{M}_1]\!]^{\mathsf{A}} \subseteq [\![\mathbf{M}_2]\!]^{\mathsf{A}}$ requires computing the abstract semantics $[\![\mathbf{M}_i]\!]^{\mathsf{A}}$, which may be costly. In this section we provide sufficient conditions for this inclusion, to be checked directly on the concrete scheduling components $\mathbf{M}_1$ and $\mathbf{M}_2$.

Lemma 3 *The following conditions on the pair* $(\mathbf{M}', \mathbf{M})$ *imply* $[\![\mathbf{M}']\!]^{\mathsf{A}} \subseteq [\![\mathbf{M}]\!]^{\mathsf{A}}$*:*

1. *There exists a surjective total map* $\psi : T' \to T$*, such that:*

   (a) *For every* $\tau \in T$*:*

   $$p^t(\tau) = \min_{\psi(\tau')=\tau} p^t(\tau') \qquad \text{and} \qquad p^c(\tau) = \max_{\psi(\tau')=\tau} p^c(\tau') \tag{16}$$

   where min and max refer to the order $\preceq'$ *generated by the precedence relation* (10) *on ports of* $\mathbf{M}'$*;*

   (b) *The following holds, for every 4-tuple of tasks* $(\tau'_1, \tau'_2, \tau_1, \tau_2) \in T'^2 \times T^2$*:*

   $$\left[\psi(\tau'_1){=}\tau_1 \text{ and } \psi(\tau'_2){=}\tau_2\right] \implies \left[\tau'_1 \parallel' \tau'_2 \Rightarrow \tau_1 \parallel \tau_2\right] \tag{17}$$

2. *For each task $\tau \in T$, there exists an injective total map $\chi_\tau : \Sigma_\tau \to \biguplus_{\tau' \in \psi^{-1}(\tau)} \Sigma_{\tau'}$, where the $\Sigma_{\tau'}$ are copies, for each referred task τ', of the alphabet Σ defined in (7); set $\chi =_{\text{def}} \biguplus_{\tau \in T} \chi_\tau$. The language L is defined through some temporal property* Timing_Prop *on the events from alphabet $\biguplus_{\tau \in T} \Sigma_\tau$, and, replacing, in* Timing_Prop*, every event e by its image $\chi(e)$ defines a language L'' such that $L'' \supseteq L'$.*

Say that $\mathbf{M}' \sqsubseteq \mathbf{M}$ when the above three conditions hold. Furthermore, the following condition implies the above Condition 1b:

$$\rho'(\tau') \quad = \quad \rho(\psi(\tau'))\,. \tag{18}$$

Observe that Conditions 1 involve only the sorts K_1 and K_2 of $\mathbf{M}_1$ and $\mathbf{M}_2$. Condition 2 formalizes the situation in which the language L_2 is specified through timing properties relating certain events of interest for tasks of $\mathbf{M}_2$ (duration between trigger and completion, end-to-end duration when traversing a set of successive tasks, etc.). The considered events are then mapped to some events of $\mathbf{M}_1$ and the timing property remains the same or is strenghtened. *Proof:* The additional statement is obvious. Note that (18) implies $R' = R$. So, we focus on the main statement. By Condition 2, we only need to prove that the scheduling conditions associated to K_1 are stronger than those associated to K_2. By definition of the precedence order, see (9), Condition 1a implies that, for every 4-tuple of tasks $(\tau'_1, \tau'_2, \tau_1, \tau_2) \in T'^2 \times T^2$,

$$\Big[\, \psi(\tau'_1){=}\tau_1 \;\text{ and }\; \psi(\tau'_2){=}\tau_2 \,\Big] \quad \Longrightarrow \quad \Big[\; \tau_1 \prec \tau_2 \quad \Rightarrow \quad \tau'_1 \prec' \tau'_2 \;\Big]\,. \tag{19}$$

Set $\Psi = (\psi, \psi) : T_1^2 \to T_2^2$. By (16), $\mathbf{M}$ involves a subset of the ports of $\mathbf{M}'$. By (17), we have $\|' \subseteq \Psi^{-1}(\|)$, and, by (19), we have $\prec' \supseteq \Psi^{-1}(\prec)$. Consequently, the ports involved in $\mathbf{M}$ are less sequentialized and more concurrent than the same ports in $\mathbf{M}'$. Furthermore, additional ports only involved in $\mathbf{M}'$ may be subject to precedence constraints and access conflicts. The inclusion $[\![\mathbf{M}']\!]^{\mathsf{A}} \subseteq [\![\mathbf{M}]\!]^{\mathsf{A}}$ follows. □ The following result expresses that abstract semantics is faithful with respect to concrete scheduling components equipped with the composition $\between$ introduced in Definition 3:

Lemma 4 *If $\mathbf{M}_1$ and $\mathbf{M}_2$ are composable, then $[\![\mathbf{M}_1 \between \mathbf{M}_2]\!]^{\mathsf{A}} = [\![\mathbf{M}_1]\!]^{\mathsf{A}} \times [\![\mathbf{M}_2]\!]^{\mathsf{A}}$.*

Proof: To construct $[\![\mathbf{M}_1 \between \mathbf{M}_2]\!]^{\mathsf{A}}$ the following steps are performed:

1. pick all pairs $(w_1, w_2) \in L_1 \times L_2$;
2. keep only the pairs (w_1, w_2) that agree on $T_1 \cap T_2$; for such a pair (w_1, w_2), fuse w_1 and w_2 by setting for $\tau \in T_i, i = 1, 2$: $w(\tau) := w_i(\tau)$;
3. keep only the words w that are behaviors of sort $K = (T, R, \rho)$, thus obtaining $[\![\mathbf{M}_1 \between \mathbf{M}_2]\!]$; for each remaining word w, following steps 2 and 3 of Definition 5, generate non-deterministically $v_K =_{\text{def}} \eta_K(w)$;
4. expand v_K to all ports and resources by applying the inverse projection of step 4 of Definition 5.

To construct $[\![\mathbf{M}_1]\!]^{\mathsf{A}} \times [\![\mathbf{M}_2]\!]^{\mathsf{A}}$ the following steps are performed:

1. pick all pairs $(w_1, w_2) \in L_1 \times L_2$;

2. keep only the pairs (w_1, w_2) such that w_i is a behavior of sort (T_i, R_i, ρ_i), for $i = 1, 2$; for each remaining word w_i, following steps 2 and 3 of Definition 5, generate non-deterministically $v_{K_i} =_{\text{def}} \eta_{K_i}(w_i)$;

3. keep only the pairs (v_{K_1}, v_{K_2}) that agree on $(P_{T_1} \cap P_{T_2}) \uplus (R_1 \cap R_2)$; for such a pair, fuse v_{K_1} and v_{K_2} by setting, for $p \in P_{T_i}$, $r \in R_j$ for $i, j = 1, 2$: $v_K(p) = v_{K_i}(p)$ and $v_K(r) = v_{K_j}(r)$;

4. expand v_K to all ports and resources by applying the inverse projection of step 4 of Definition 5.

The first and last steps of these two procedures are identical. On the other hand, the mapping $\mathbf{M} \mapsto [\![\mathbf{M}]\!]^{\mathsf{A}}$ specified in step 2 of Definition 5 is indeed designed so that the second and third steps of the above two procedures yield identical results. □ For $(\mathbf{A}, \mathbf{G})$ a composable pair of concrete scheduling components, we will sometimes need to consider the expression $[\![\mathbf{G}]\!]^{\mathsf{A}} \cup \neg[\![\mathbf{A}]\!]^{\mathsf{A}}$. This is addressed in the following lemma, where the items of $\mathbf{A}$ possess "$_{\mathbf{A}}$" as a subscript, and similarly for $\mathbf{G}$:

Lemma 5 *Let $(\mathbf{A}, \mathbf{G})$ be a composable pair of concrete scheduling components such that $R_{\mathbf{A}} = \emptyset$. Then, the following formulas define a concrete scheduling component $\mathbf{M} = ((T, R, \rho), L)$ such that $[\![\mathbf{M}]\!]^{\mathsf{A}} = [\![\mathbf{G}]\!]^{\mathsf{A}} \cup \neg[\![\mathbf{A}]\!]^{\mathsf{A}}$:*

$$T = T_{\mathbf{A}} \cup T_{\mathbf{G}}\ ,\ R = R_{\mathbf{G}}\ ,\ \rho(\tau) = \textbf{if } \tau \in T_{\mathbf{A}} \textbf{ then } \rho_{\mathbf{A}}(\tau), \tag{20}$$

$$L = \mathbf{pr}^{-1}_{T \to T_{\mathbf{G}}}(L_{\mathbf{G}}) \cup \mathbf{pr}^{-1}_{T \to T_{\mathbf{A}}}(\neg L_{\mathbf{A}}) \tag{21}$$

where $\mathbf{pr}^{-1}_{T \to T_{\mathbf{G}}}()$ and $\mathbf{pr}^{-1}_{T \to T_{\mathbf{A}}}()$ denotes the referred inverse projections.

Proof: This results from the fact that, thanks to the assumption that $R_{\mathbf{A}} = \emptyset$, the characterization (2) for the abstract scheduling components that are image of a concrete scheduling component is satisfied. □

4.3 Scheduling Contracts

4.3.1 Scheduling Components and Scheduling Contracts

In this section we reuse Section 4 of companion paper [11]. As recomended in that reference, we first define what components are for this theory, and then we define contracts. Regarding components, the notations used here refer to the operations $\subseteq, \cap, \cup, \neg, \times$ introduced for abstract scheduling components in Section 4.2.2.

Definition 6 (scheduling components and scheduling contracts) *A* scheduling contract *is a pair $\mathscr{C} = (A, G)$ of abstract scheduling components, called its* assumptions *and* guarantees*, respectively.*

The set $\mathcal{E}_{\mathscr{C}}$ of the legal environments for $\mathscr{C}$ collects all abstract scheduling components E with non-empty semantics and such that $E \subseteq A$. The set $\mathcal{M}_{\mathscr{C}}$ of all implementations of $\mathscr{C}$ collects all abstract scheduling components M with non-empty semantics and such that $A \times M \subseteq G$.

Each scheduling contract can be put in its equivalent saturated form *$\mathscr{C} = (A, G \cup \neg A)$, possessing the same sets of legal environments and implementations. Scheduling contract $\mathscr{C}$ is compatible if and only if $A \neq \emptyset$ and consistent if and only if $G \cup \neg A \neq \emptyset$. Say that scheduling contract $\mathscr{C} = (A, G)$ is* schedulable *if $A \cap G \neq \emptyset$.* □

The material of Section 4.1 of companion paper [11] applies verbatim and the reader is referred ot it. Note that $A \cap G = A \cap (G \cup \neg A)$, hence checking schedulability does not require the contract to be saturated. The justification of this notion of schedulability for contracts is given in the next section.

In practice the designer will specify scheduling contracts via pairs $(\mathbf{A}, \mathbf{G})$ of composable concrete scheduling components, called its assumption and guarantee (see Definition 3 for the notion of composability). Thus, define:

Definition 7 *Call* concrete *scheduling contract (or* concrete *contract) a pair* $\mathbf{C} = (\mathbf{A}, \mathbf{G})$ *of composable concrete scheduling components called its* assumptions *and* guarantees.

To contrast with concrete contracts, we will sometimes call *abstract scheduling contracts*, or *abstract contracts*, the scheduling contracts of Definition 6. The mapping from concrete to abstract scheduling components developed in Section 4.2.3 allows mapping concrete scheduling contracts to abstract ones:

$$\mathbf{C} = (\mathbf{A}, \mathbf{G}) \quad \mapsto \quad \mathscr{C}_{(\mathbf{A},\mathbf{G})} =_{\text{def}} (\llbracket \mathbf{A} \rrbracket^{\mathsf{A}}, \llbracket \mathbf{G} \rrbracket^{\mathsf{A}}) \tag{22}$$

Say that $\mathbf{C}$ is *consistent, compatible, schedulable,* or *in saturated form,* if so is $\mathscr{C}_{(\mathbf{A},\mathbf{G})}$. Regarding schedulability, the following holds:

Lemma 6 *If* $\mathbf{C}$ *is schedulable, then it has* $\mathbf{G}$ *as a concrete implementation that remains schedulable (in the sense of Definition 2) when put in the context of* $\mathbf{A}$. □

Proof: By Definition 6 regarding schedulability of abstract contracts, we have $\llbracket \mathbf{A} \rrbracket^{\mathsf{A}} \times \llbracket \mathbf{G} \rrbracket^{\mathsf{A}} = \llbracket \mathbf{A} \rrbracket^{\mathsf{A}} \cap \llbracket \mathbf{G} \rrbracket^{\mathsf{A}} \neq \emptyset$. By Lemma 4, we have $\llbracket \mathbf{A} \rrbracket^{\mathsf{A}} \times \llbracket \mathbf{G} \rrbracket^{\mathsf{A}} = \llbracket \mathbf{A} \underline{\times} \mathbf{G} \rrbracket^{\mathsf{A}}$. By Statement 1 of Lemma 2, $\llbracket \mathbf{A} \underline{\times} \mathbf{G} \rrbracket^{\mathsf{A}} \neq \emptyset$ if and only if $\llbracket \mathbf{A} \underline{\times} \mathbf{G} \rrbracket \neq \emptyset$, which is the implication stated in the lemma. □ The reason for considering mapping (22) is that only abstract scheduling contracts, not concrete ones, are equipped with the contract algebra. Observe that contract $\mathscr{C}_{(\mathbf{A},\mathbf{G})}$ may not be in *saturated form*. To prove contract properties, we need a few criteria that are expressed in terms of concrete scheduling components and use only the algebra available for them.

By notational convention and unless confusion can result, we will simply denote by $\mathscr{C}$ the abstract contract associated to concrete contract $\mathbf{C}$ through the mapping (22).

4.3.2 A toolbox of sufficient conditions in terms of concrete contracts

These criteria will only be sufficient conditions, stronger than the verification of the same properties expressed in the true contract domain, i.e., by using abstract scheduling components (recall that the latter are not convenient for practical specification).

Lemma 7 (Checking for implementation and environment relations) *The following conditions imply that* $\llbracket \mathbf{E} \rrbracket^{\mathsf{A}}$ *is an environment and* $\llbracket \mathbf{M} \rrbracket^{\mathsf{A}}$ *is an implementation for* $\mathscr{C}_{(\mathbf{A},\mathbf{G})}$*:*

$$\mathbf{E} \sqsubseteq \mathbf{A} \quad \text{and} \quad \mathbf{M} \text{ is composable with } \mathbf{A} \text{ and } \mathbf{A} \underline{\times} \mathbf{M} \sqsubseteq \mathbf{G} \tag{23}$$

Proof: This follows from Lemmas 3 and 4. □

Lemma 8 (Checking for contract refinement) *The following conditions imply refinement* $\mathscr{C}_{(\mathbf{A},\mathbf{G})} \preceq \left(\bigwedge_{j \in J} \mathscr{C}_{(\mathbf{A}_j,\mathbf{G}_j)}\right)$*:*

$$\forall j \in J \quad \Longrightarrow \quad \mathbf{A}_j \text{ is composable with } \mathbf{G} \text{ and } \begin{cases} \mathbf{A}_j \underline{\times} \mathbf{G} & \sqsubseteq & \mathbf{G}_j \\ \mathbf{A}_j & \sqsubseteq & \mathbf{A} \end{cases} \tag{24}$$

Proof: that (24) implies the desired refinement: by Lemmas 3 and 4, Condition (24) implies its abstract counterpart:

$$\forall j \in J \quad \Longrightarrow \quad \left\{ \begin{array}{rcl} [\![\mathbf{A}_j]\!]^{\mathsf{A}} \times [\![\mathbf{G}]\!]^{\mathsf{A}} & \subseteq & [\![\mathbf{G}_j]\!]^{\mathsf{A}} \\ [\![\mathbf{A}_j]\!]^{\mathsf{A}} & \subseteq & [\![\mathbf{A}]\!]^{\mathsf{A}} \end{array} \right. \tag{25}$$

Focus first on environments. Pick any abstract scheduling component E such that $E \subseteq [\![\mathbf{A}_j]\!]^{\mathsf{A}}$ for some j. Using the second inclusion of (25) we deduce that $E \subseteq [\![\mathbf{A}]\!]^{\mathsf{A}}$. Consider next implementations. Pick any abstract scheduling component M such that $[\![\mathbf{A}]\!]^{\mathsf{A}} \times M \subseteq [\![\mathbf{G}]\!]^{\mathsf{A}}$. Since $[\![\mathbf{A}_j]\!]^{\mathsf{A}} \subseteq [\![\mathbf{A}]\!]^{\mathsf{A}}$, we deduce $[\![\mathbf{A}_j]\!]^{\mathsf{A}} \times M \subseteq [\![\mathbf{A}_j]\!]^{\mathsf{A}} \times [\![\mathbf{G}]\!]^{\mathsf{A}}$, and thus (25) implies $[\![\mathbf{A}_j]\!]^{\mathsf{A}} \times M \subseteq [\![\mathbf{G}_j]\!]^{\mathsf{A}}$. This proves that Conditions (25) imply the desired refinement, hence so do Conditions (24). □

Formula (24) supports refinement checking for pairs of concrete assumptions and guarantees that do not induce contracts in saturated form.

Checking for contract refinement and composition: For $(\mathbf{A}, \mathbf{G})$ a concrete system-level contract and $(\mathbf{A}_i, \mathbf{G}_i)$, $i \in I$ a set of concrete sub-contracts assigned to each subsystem in an architectural decomposition of the global system, we typically want to check if refinement $\otimes_{i\in I} \mathscr{C}_{(\mathbf{A}_i,\mathbf{G}_i)} \preceq \mathscr{C}_{(\mathbf{A},\mathbf{G})}$ holds. Unfortunately, checking this requires having these contracts in saturated form, see Section 4 of companion paper [11]. To this end we can use Lemma 5, which provides a concrete formula for representing the saturated form of a contract, assuming that assumptions of this contract have no resource assigned to them. If this is not possible (e.g., because assumptions involve resources), we will need to work directly in the domain of (abstract) scheduling contracts.

Comment 7 (verification vs. synthesis of contracts) We now have the material at hand for: 1) verifying that successive refinement steps proposed by the designer are correct and, 2) checking for implementation and environment relations. We can also synthesize the conjunction and composition of abstract contracts, but we have no way to reverse engineer the results back to concrete scheduling components. To summarize, our contract framework supports verification of independent designs but is not powerful enough to synthesize them—as it was for instance performed for the parking garage example of Section 3.

4.3.3 Getting sub-contracts in the AUTOSAR development process

In this section we develop techniques in support of the following design steps, which are advocated by AUTOSAR:

Process 1 (AUTOSAR development process)

1. Start with a top-level, system wide, contract. At this level, only functions are considered while computing resources are ignored. Functions are abstracted as systems of tasks with their precedence constraints. The top-level contract may be the conjunction of several viewpoints, and/or it may be specified by means of requirement tables.

2. To prepare for subcontracting to different suppliers, decompose this functional top-level contract into functional sub-contracts. So far computing resources are not considered.

3. At this step the computing infrastructure is now taken into account. Perform system wide (global) task scheduling, thus inferring resource budgets.

4. Derive resource aware sub-contracts and submit them to the supplier, for implementation. (The supplier may request a negotiation in case resource budgeting is too tight for him to meet the sub-contract.) □

This process is rather informal. It is thus tempting to interpret the above tasks as refinement steps, for scheduling contracts. With this in mind, Steps 1 and 2 exhibit no particular difficulty. Step 3, however, raises a problem. Adding the consideration of resources to a resourceless contract cannot be a refinement step. This can be seen from Lemma 3, which gives sufficient conditions for concrete contract refinement: referring to this lemma, there is no way that the resulting contracts $\mathbf{C}' = (\mathbf{A}', \mathbf{G}')$ can refine $\mathbf{C}$ since $[\![\mathbf{A}']\!]^{\mathsf{A}} \supseteq [\![\mathbf{A}]\!]^{\mathsf{A}}$ is not possible when resources are added, from $\mathbf{A}$ to $\mathbf{A}'$. This is no surprise in fact, since one cannot independently add *shared* resources to different contracts, and at the same time expect to be able to develop and implement them independently.

Of course, from a theoretical standpoint, there is an easy solution to this problem. One could argue that not considering resources and budgeting them from the very beginning is a mistake and cannot work. Following this argument we would need to consider resources already in the top-level contracts, and address budgeting right from the beginning. Unfortunately, this is in total disagreement with the AUTOSAR approach, which advocates at early stages the specification of software architectures consisting of software components, regardless of resources.

To overcome this difficulty, our approach is: 1) to precisely characterize the "illegal" development steps we perform that violate contract refinement, and 2) to precisely identify the resulting risks for later system integration. We will need the following notion of "port-refinement" for concrete contracts, which is an approximation of refinement in which only ports are taken into consideration.

Port-refinement of contracts: Decompose the alphabet $\mathcal{V}$ introduced in Definition 4:

$$\mathcal{V} \;=\; (\,\{0,1\}^{\mathcal{P}}\,) \times (\,\textstyle\prod_{r\in\mathcal{R}} \Sigma_r\,) \;=\; \mathcal{V}^{\mathcal{P}} \times \mathcal{V}^{\mathcal{R}} \tag{26}$$

For $\mathbf{M} = ((T, R, \rho), L)$ a concrete scheduling component, define

$$[\![\mathbf{M}]\!]^{\mathcal{P}} \quad =_{\text{def}} \quad \mathbf{pr}_{\mathcal{V}^{\mathcal{P}}}\Big([\![\mathbf{M}_{\rho/\epsilon}]\!]^{\mathsf{A}}\Big), \text{ where } \mathbf{M}_{\rho/\epsilon} =_{\text{def}} ((T, \emptyset, \epsilon), L), \tag{27}$$

and ϵ is the allocation map with empty domain. In words, we first ignore the possible conflicts due to shared resources (replacing $\mathbf{M}$ by $\mathbf{M}_{\rho/\epsilon}$), we then take the abstract semantics $[\![\mathbf{M}_{\rho/\epsilon}]\!]^{\mathsf{A}}$, and we finally project the resulting abstract semantics over the ports only (taking $\mathbf{pr}_{\mathcal{V}^{\mathcal{P}}}(...)$). $[\![\mathbf{M}]\!]^{\mathcal{P}}$ captures the scheduling aspect of $\mathbf{M}$ while discarding the resource aspect of it. Observe that $[\![\mathbf{M}]\!]^{\mathcal{P}}$ contains the language obtained by projecting $[\![\mathbf{M}]\!]^{\mathsf{A}}$ over the ports; this inclusion is generally strict. If, however, $\mathbf{M} = ((T, \emptyset, \epsilon), M)$ is resourceless, then $[\![\mathbf{M}]\!]^{\mathcal{P}} = \mathbf{pr}_{\mathcal{V}^{\mathcal{P}}}\Big([\![\mathbf{M}]\!]^{\mathsf{A}}\Big)$. For $\mathbf{C} = (\mathbf{A}, \mathbf{G})$ a concrete scheduling contract, define

$$[\![\mathbf{C}]\!]^{\mathcal{P}} \quad =_{\text{def}} \quad ([\![\mathbf{A}]\!]^{\mathcal{P}}, [\![\mathbf{G}]\!]^{\mathcal{P}}), \text{ the } \textit{port-contract} \text{ associated with } \mathbf{C}. \tag{28}$$

Despite the boldface notation used, port-contracts are abstract contracts. For $\mathbf{C}$ and $\mathbf{C}'$ two concrete contracts, say that

$$\mathbf{C}' \textit{ port-refines } \mathbf{C}, \text{ written } \mathbf{C}' \preceq_{\mathcal{P}} \mathbf{C} \quad \text{ if } \quad [\![\mathbf{C}']\!]^{\mathcal{P}} \leq [\![\mathbf{C}]\!]^{\mathcal{P}}. \tag{29}$$

T-closed contracts and illegal development steps: We restrict these steps to the following situation, which does not contradict the AUTOSAR methodology. Assume, from

early design stages on, prior knowledge of the following property about a given set T of tasks—this does not require detailed knowledge of the computing resources:

Definition 8 (T-closed contracts) *Say that a set $T \subset \mathcal{T}$ of tasks is* segregated *if the set $\mathcal{R}$ of all resources partitions as follows:*

$$\mathcal{R} = \mathcal{R}_T \cup \overline{\mathcal{R}}_T, \mathcal{R}_T \cap \overline{\mathcal{R}}_T = \emptyset, \quad \textit{and} \quad \left\{ \begin{array}{rcl} T & \subseteq & \Sigma_{\mathcal{R}_T} \\ \mathcal{T} - T & \subseteq & \Sigma_{\overline{\mathcal{R}}_T} \end{array} \right. \tag{30}$$

For any segregated set of tasks T, say that concrete contract $\mathbf{C} = (\mathbf{A}, \mathbf{G})$ is T-closed *if $T_{\mathbf{G}} \subseteq T$ and $T_{\mathbf{A}} \cap T = \emptyset$.* □

An instance of T-closed set of tasks will naturally occur in our application case. If $\mathbf{C} = (\mathbf{A}, \mathbf{G})$ is T-closed, then $\rho_{\mathbf{G}}(T_{\mathbf{G}}) \cap \rho_{\mathbf{A}}(T_{\mathbf{A}}) = \emptyset$ holds. *Illegal steps* are performed on T-closed contracts only. An illegal step consists in replacing T-closed contract $\mathbf{C}$ by another T-closed contract $\mathbf{C}'$ port-refining it: $\mathbf{C}' \preceq_{\mathcal{P}} \mathbf{C}$.

The resulting risks at system integration: Port-refinement being not a refinement, replacing $\mathbf{C}$ by $\mathbf{C}'$ *won't* ensure that any implementation of $\mathbf{C}'$ will meet the guarantees of $\mathbf{C}$ under any legal environment for $\mathbf{C}'$—it should ensure this if it was a true refinement. Still, the following result holds, which precisely bounds the risks at system integration. In this lemma, we generically denote by M the abstract scheduling component associated to $\mathbf{M}$.

Lemma 9 *Let be $\mathbf{C}' \preceq_{\mathcal{P}} \mathbf{C}$ satisfying the following conditions:*

1. *$\mathbf{C}$ and $\mathbf{C}'$ are T-closed for a same segregated set T of tasks,*
2. *$\mathbf{C}'$ is schedulable,*
3. *$[\![\mathbf{A}']\!]^{\mathcal{P}} = [\![\mathbf{A}]\!]^{\mathcal{P}}$, $\mathbf{A}$ and $\mathbf{A}'$ both have their tasks pairwise non-conflicting, and*
4. *$\mathbf{G}$ is resourceless.*

Then, the following holds: $\emptyset \neq \mathbf{A} \underline{\times} \mathbf{G}' \sqsubseteq \mathbf{G}$. □

Proof: Property $\emptyset \neq \mathbf{A} \underline{\times} \mathbf{G}'$ follows from Condition 3 and the assumption that $\mathbf{C}'$ is schedulable. Finally, since $\mathbf{C}' \preceq_{\mathcal{P}} \mathbf{C}$, we have $[\![\mathbf{A}]\!]^{\mathcal{P}} \times [\![\mathbf{G}']\!]^{\mathcal{P}} \subseteq [\![\mathbf{G}]\!]^{\mathcal{P}}$, which implies $\mathbf{A} \underline{\times} \mathbf{G}' \sqsubseteq \mathbf{G}$ since $\mathbf{G}$ is resourceless. □

Comment 8 Lemma 9 expresses that $\mathbf{G}'$ is an implementation of $\mathbf{C}'$ that, when put in the context of the most permissive environment of $\mathbf{C}$, meets the guarantee $\mathbf{G}$ and is schedulable. That $\mathbf{G}$ is met will remain valid for any legal environment of $\mathbf{C}$ and any implementation of $\mathbf{C}'$. Schedulability, however, is only ensured by the most permissive environment of $\mathbf{C}$ and implementation of $\mathbf{C}'$. This restriction is not surprising since schedulability is a liveness property whereas A/G-contracts support only safety properties. □

The Autosar *development process made safe*: We are now ready to explain how the Autosar development process (Process 1) can be made safe by implementing the illegal development steps safely.

Process 2 (Autosar development process made safe) We assume a segregated subset T of tasks.

1. Start with a top-level, T-closed, contract $\mathbf{C}^{\text{func}}_{\text{top}} = (A_{\text{top}}, G_{\text{top}})$. At this level, only functions are considered while computing resources are ignored. Functions are abstracted as systems of tasks with their precedence constraints. The top-level contract may be the conjunction of several viewpoints, and/or it may be specified by means of requirement tables.

   - *Comment*: No change with respect to Process 1 besides T-closedness.

2. To prepare for subcontracting to different suppliers, decompose the above functional contract $\mathbf{C}_{\text{top}}$ into functional, resource agnostic, sub-contracts in such a way that

$$\mathbf{C}^{\text{func}}_{\text{ref}} = (\mathbf{A}_{\text{ref}}, \mathbf{G}_{\text{ref}}) = \ \underline{\times}_{i\in I}\, \mathbf{C}_i \quad \text{satisfies} \quad \begin{cases} \mathbf{A}_{\text{top}} \underline{\times} \mathbf{G}_{\text{ref}} & \sqsubseteq & \mathbf{G}_{\text{top}} \\ \mathbf{A}_{\text{ref}} & \sqsupseteq & \mathbf{A}_{\text{top}} \end{cases} \tag{31}$$

   where the $\mathbf{C}_i$ are T-closed subcontracts for the different suppliers. In addition, we require that $\mathbf{A}_{\text{ref}}$ and $\mathbf{A}_{\text{top}}$ possess identical sets of tasks, i.e., map ψ of Lemma 3 is the identity. By Lemma 8, (31) ensures $\mathscr{C}^{\text{func}}_{\text{ref}} \preceq \mathscr{C}^{\text{func}}_{\text{top}}$. So far resources were not considered.

   - *Comment*: No change so far, with respect to Process 1, besides naming contracts and making refinement step precise through (31). The first two steps make no reference to semantics, meaning that no scheduling analysis is required, cf. Comment 5. From the next step on, this process deviates from Process 1.

3. At this step the computing resources are now taken into account. Allocate a resource to each task of $\mathbf{A}_{\text{ref}}$ and $\mathbf{G}_{\text{ref}}$, in such a way that all tasks of $\mathbf{A}_{\text{ref}}$ are pairwise non-conflicting, see Definition 1. Precedence constraints between tasks are not modified. This yields a resource aware T-closed contract $\mathbf{C}^{\text{res}}_{\text{ref}}$ such that

$$\mathbf{C}^{\text{res}}_{\text{ref}} = (\mathbf{A}^{\text{res}}_{\text{ref}}, \mathbf{G}^{\text{res}}_{\text{ref}}) \quad \preceq_{\mathcal{P}} \quad \mathbf{C}^{\text{func}}_{\text{ref}} \tag{32}$$

   Since $\mathbf{A}^{\text{res}}_{\text{ref}}$ is free of conflict, only $\mathbf{G}^{\text{res}}_{\text{ref}}$ requires a non-trivial scheduling analysis, which result is specified through the semantics $[\![\mathbf{G}^{\text{res}}_{\text{ref}}]\!]$, cf. Comment 5. At this point, resources have been globally budgeted and scheduling analysis globally performed.

   - *Comment*: This is the illegal step, which is protected by Lemma 9.

4. Continue by decomposing contract $\mathbf{C}^{\text{res}}_{\text{ref}}$ into resource aware sub-contracts $\mathbf{C}^{\text{res}}_i$, following the architecture specified at Step 2, in such a way that $\bigotimes_{i\in I} \mathscr{C}^{\text{res}}_i \preceq \mathscr{C}^{\text{res}}_{\text{ref}}$. The results of the next section can be used for this.

4.3.4 Dealing with non staturated contracts

The operation of contract saturation is algorithmically complex since it requires taking complements. In this section we discuss direct sufficient conditions to support independent development without the need for saturating contracts.

Lemma 10 *Let $\mathscr{C} = (A, G)$ be a (possibly nonsaturated) contract such that $A, G \subseteq D^* \cup D^\omega$, where X is some set of variables with domain D_x and $D =_{\text{def}} \prod_{x\in X} D_x$. Assume $X = X_1 \cup X_2$ and set $D_i =_{\text{def}} \prod_{x\in X_i} D_x$. Assume a decomposition*

$$G = G_1 \cap G_2 \tag{33}$$

where $G_i = \mathbf{pr}^{-1}_{D\to D_i}(H_i)$ *for some* $H_i \subseteq D_i^* \cup D_i^\omega$*, meaning that* G_i *involves only variables belonging to* X_i*. Then, the following conditions on the pair* (A_1, A_2) *ensure that the two contracts* $\mathscr{C}_1 = (A_1, G_1)$ *and* $\mathscr{C}_2 = (A_2, G_2)$ *satisfy* $\mathscr{C}_1 \otimes \mathscr{C}_2 \preceq \mathscr{C}$*:*

$$\begin{array}{rcl} A_1 \cup A_2 & \supseteq & A \cap \neg G \\ \neg G_1 \cup A_2 & \supseteq & A \cap \neg G \\ A_1 \cup \neg G_2 & \supseteq & A \cap \neg G \\ (A_1 \cap A_2) \cup \neg G & \supseteq & A \end{array} \tag{34}$$

How to satisfy (33) *in practice*: Suppose that the set of behaviors G is defined by some finite set $\mathbb{E}$ of equations involving the variables of X. We associate to $\mathbb{E}$ its *incidence graph* $\mathscr{G}_\mathbb{E}$, which is a non directed bipartite graph with $\mathbb{E} \uplus X$ as set of vertices. $\mathscr{G}_\mathbb{E}$ has a branch $(x, E) \in X \times \mathbb{E}$ if and only if equation E involves the variable x. Recalling that $X = X_1 \cup X_2$, set $\mathbb{E}_1 =_{\text{def}} \{E \mid (x, E) \in \mathscr{G}_\mathbb{E} \text{ and } x \in X_1\}$ and similarly for $\mathbb{E}_2$. We have $\mathbb{E} = \mathbb{E}_1 \cup \mathbb{E}_2$ (the two subsets overlap in general), which induces $G = G_1 \cap G_2$. *Proof:* of the lemma. None of the considered contracts is saturated. So we first saturate them, that is, we redefine $\mathscr{C} = (A, (G \cup \neg A))$ and $\mathscr{C}_i = (A_i, (G_i \cup \neg A_i))$. Setting $G' =_{\text{def}} G \cup \neg A$ and $G'_i =_{\text{def}} G_i \cup \neg A_i$, we have, regarding the guarantees:

$$\begin{array}{rcl} G'_1 \cap G'_2 & = & (G_1 \cup \neg A_1) \cap (G_2 \cup \neg A_2) \\ & = & G \cup (G_1 \cap \neg A_2) \cup (G_2 \cap \neg A_1) \cup \neg(A_1 \cup A_2) \\ \text{(by the first three lines of (34))} & \subseteq & G \cup \neg A \end{array}$$

which implies $G'_1 \cap G'_2 \subseteq G'$. Regarding the assumptions, we have:

$$\begin{array}{rcl} (A_1 \cap A_2) \cup \neg(G'_1 \cap G'_2) & \supseteq & (A_1 \cap A_2) \cup \neg(G \cup \neg A) \\ & = & (A_1 \cap A_2) \cup (\neg G \cap A) \\ \text{(by the last line of (34))} & \supseteq & A \end{array}$$

This shows the lemma. □

4.4 Modeling methodology and extensions used in the case study

In this section we discuss the use of our framework in practice. We first discuss how scheduling components capture scheduling problems in practice. Then, in order to capture read and write actions we propose an extension of our existing set of pure events carried by ports.

4.4.1 Capturing scheduling problems with our framework

Scheduling problems are captured by our notion of Concrete Scheduling Component $\mathbf{M}=(K, L)$. Sort K identifies the set of tasks together with their ports, induced precedence conditions, and the resource allocation map. Language L allows expressing various dynamical constraints such as, for instance:

1. Bounds on the execution time of tasks, i.e., the number of busy slots for each epoch;
2. Bounds on the duration of intervals $[i_\tau, o_\tau]$, from start to completion events;
3. Bounds on the intervals $[p^t(\tau), i_\tau]$ from release times to start times of tasks;
4. Minimum inter-arrival time between two successive triggers $p^t(\tau)$ for a task;

5. Bounds on the response time interval $[p^t(\tau), p^o(\tau)]$ of tasks for each output $p^o(\tau) \in P^o(\tau)$ produced by the task (this captures deadlines);

6. When combining the consideration of K and L it is possible, for two tasks τ_1 and τ_2 such that $\tau_1 \preceq \tau_2$, to express in L end-to-end bounds for the interval $[i_{\tau_1}, o_{\tau_2}]$.

Observe that bounds 1–3 cannot be expressed using abstract scheduling components, but other above listed properties 4–6 can.

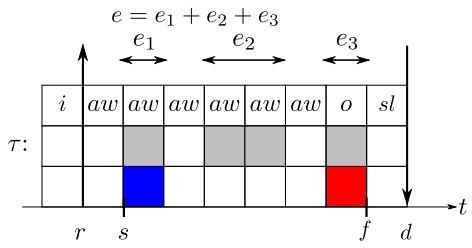


Figure 14: Typical real-time task parameters + timing constraints on data paths

In the literature about real-time scheduling analysis [34, 16, 47], there exist a common understanding about typical task parameters, which are important for 1) defining timing constraints and 2) defining algorithms solving scheduling problems like checking *feasibility* of a schedule or the *schedulability* of a task set. Figure 14 puts these task parameters in the context of our model of concrete scheduling components. For a task τ, the following parameters are typically of interest:

- r: The release or activation time of a task. This is the point in time when a new job of a task is created and becomes ready for execution.

- s: The start time of a task. This is the point in time when the task starts executing its job. It may coincide with the release time or delayed for example by a higher priority task that is executing.

- f: The finishing time of a task. This is the point in time when the task finished its current job and terminates.

- R: The response time of a task. This is the difference between the finishing time the the release time of a task: $R = f - r$.

- d: A relative deadline of a task. This is the point in time when the task must have finished its current job. Usually this is seen relative to r. A task meets its deadline if $R \leq d$.

- e: The execution time of a task. This is the time a task needs for execution on its processor without being preempted.

So far we discussed the specification of monolithic scheduling problems. Compositionality is supported by contracts, which we discuss next.

Our model of Definitions 1 and 2 carries the essence of real-time scheduling, namely: resources, tasks, precedence conditions between tasks, and the language L that can be used to express various timing assumptions or guarantees. Now, this model does not offer the expressiveness we need for our application case developed in Section 4.5.

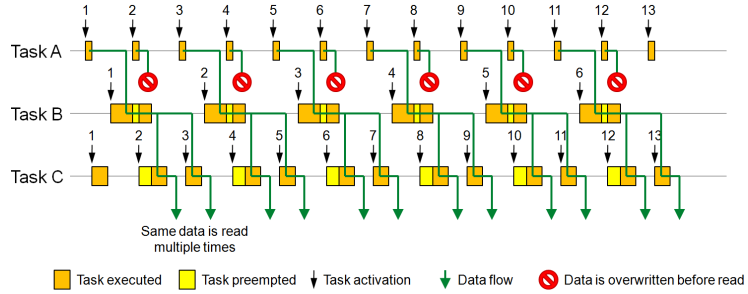


Figure 15: Loss and duplication of data due to under- and oversampling [21]

4.4.2 Extensions used in practice

To avoid complicating our theoretical development, we only give informal explanations. So far in our current framework of Definitions 1 and 2, each task τ possesses one trigger and one completion port. In fact, our practical application requires considering additional ports.

Figure 15 motivates the extension with regard to reads/writes of variables. Identification of activation or input ports with output or completion ports of other tasks represent data dependencies within a task set. In [34, Section 3.4] such dependencies are discussed in terms of a *task graph*, which is distinquished from a *precedence graph*, with the former being an extension of the precedence graph by adding different kinds of edges between tasks denoting for example data dependencies. Such data paths may involve under- and oversampling effects like depicted in Figure 15. Though these dependencies do not necessarily coincide with task predences, it is typical in control engineering to require input values to not exceed a certain age (called *AgeConstraint* in [21]). At the same time, data loss due to undersampling is acceptable. This discussion motivates the following extensions related to ports:

Input and Output ports: In addition to its trigger and completion ports $p^t(\tau)$ and $p^c(\tau)$, each task τ possesses two sets of *input ports* $P^i(\tau) = \{p^i(\tau, k) \mid k \in K_\tau\}$ and *output ports* $P^o(\tau) = \{p^o(\tau, \ell) \mid \ell \in L_\tau\}$. As part of the specification of the considered concrete scheduling component $\mathbf{M}$, the relation $\dashv\!\boxminus\!\rightarrow$ between ports introduced in (10) can be extended to the whole set of ports associated to task τ, namely $\{p^t(\tau)\} \cup P^i(\tau) \cup P^o(\tau) \cup \{p^c(\tau)\}$, with the condition that the trigger port and completion port remain minimal and maximal in this extended order. Since ports can be shared between tasks, the precedence relation $\dashv\!\boxminus\!\rightarrow$ extends to the set of all ports of the concrete scheduling component $\mathbf{M}$. The reason for this extension is that AUTOSAR designs generally involve tasks having several control points from which certain "posterior" tasks can be launched.

Read and Write actions: To each input port $p^i(\tau, k)$ we associate some *read action* occurring at some busy slot of the task. Similarly, to each output port $p^o(\tau, \ell)$ we associate some *write action* occurring at some busy slot of the task.

Semantics: The semantics $[\![\mathbf{M}]\!]$ is obtained by adapting Definition 2 to the refined ordering between start and completion of tasks and the various read and write actions, and meeting the conflict freeness condition.

Extending Lemma 3 regarding inclusion: In this lemma tasks get refined by using the surjective total map $\psi : T_1 \to T_2$, where T_1 is the set of tasks of the refined concrete scheduling component. Since the pair of trigger and completion ports, which was associated to each task, is now replaced by a pair of sets of ports, we need to replace the map ψ acting on tasks by a surjective map $\Psi : P_1 \to P_2$ acting on ports. As a counterpart of Condition 1 of Lemma 3, we require that $\Psi(\preceq_1) = \preceq_2$, where $\preceq_i$ is the precedence order on P_i, for $i = 1, 2$. Conditions 1b and 2 of Lemma 3 remain unchanged.

In the sequel, we will be using the above extension of our contract framework. We now move to our application case.

4.5 AUTOSAR compliant development of an Exterior Light Management System

To illustrate the practical use of the framework of scheduling contracts in AUTOSAR, we consider as an example an excerpt of an exterior light management system for an automobile.[17] We focus on the parts responsible for sensing driver inputs and actuating the rear direction indicator lights and brake lights. With this example we show how typical timing requirements can be expressed in the contract framework and discuss the added value of these contracts for negotiations between OEM and suppliers. Also we reconsider the AUTOSAR methodology by applying development Process 2. All of this allows us to establish contractual specifications along the supply chain on firm bases, even when electronic control units (ECUs) are going to be shared by different suppliers, who independently implement software components for them. Throughout the case-study we assume discrete time slots to have a fixed duration of 1μs.

Regarding modeling methodology and notations, we will be using both concrete contracts (for the specification of contracts at early steps of the design) and abstract scheduling contracts (when using the contract algebra). We will use the symbols $\mathbf{C}$ and $\mathscr{C}$ to distinguish between them.

Step 1 of Process 2: top-level specification of the Virtual Functional Bus

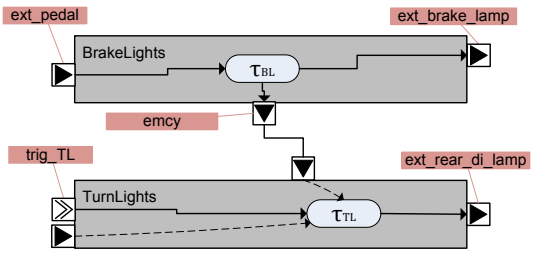


Figure 16: Virtual Functional Bus (*VFB*) architecture

We begin our study by showing how timing requirements of AUTOSAR components can be captured by means of the contract framework. Figure 16 shows the Virtual Functional Bus (*VFB*) architecture handling the exterior light management. It specifies

[17] A case-study from the German SPES2020 project

the interfaces exposed by the components to the sensors delivering driver inputs (brake pedal, warn light button, turn signal lever)[18] and to the actuators controlling the lights (rear direction indicator lights and brake lights). The system shall control the brake lights in accordance with the driver pressing the brake pedal. The TurnLights component controls the direction indicator lights according to the position of the turn signal lever and the warn lights button. The system shall also implement an emergency stop signal, where warn lights flash when the vehicle is braked severely.

The graphical notations in Figure 16 distinguishes pure data flows (the dashed lines) and control flows (the solid lines), where the latter ones may also carry data items. For example the task τ_{TL} is only triggered whenever an event occurs on port trig_TL. When executing, it will read the data item from port emcy. In our context, pure data flow wires induce no precedence constraint: reads from the wire occur independently from corresponding writes.[19]

The requirements document of the system contains two timing-related requirements R_1 and R_2 shown on Table 3, top. We formalize these two requirements as scheduling contracts in the sequel of Table 3. To this end, we first capture the *VFB* architecture through the sort of its component K^G_{VFB}: each component is represented by means of a task; there are not further tasks since the two blocks BrakeLights and TurnLights are considered black-box at this stage; for the same reason, no resource is allocated; we only specify part of the ports for tasks; the orders $\multimap$ and $\boxminus$ and precedence order $\preceq$, follow as explained in (9) and (10). For the two contracts $\mathbf{C}_{R_i}, i = 1, 2$, the sort of their assumption $\mathbf{A}_i$ is specified by K^A_{VFB}; note that $\mathbf{A}_i$ is composable with $\mathbf{G}_i$ as requested for a contract; the languages L_{A_i} and L_{G_i} are specified by using a pattern-based contract specification language (terms in bold-face are keywords). $\mathbf{A}_1$ and $\mathbf{A}_2$ reflect and make explicit an assumption about the frequency of sensors samples of the brake pedal position.[20] The guarantees specify an interval for the latency when new values have been computed and are sent at output ports. Using conjunction we obtain the contract expressing the behavior required by the system requirements R_1 and R_2: $\mathscr{C}_{\text{top\_level}} = \mathscr{C}_{R_1} \wedge \mathscr{C}_{R_2}$, where $\mathscr{C}_{R_i}$ is the abstract counterpart of $\mathbf{C}_{R_i}$—we must switch to abstract contracts to apply the conjunction.

[18] We do not name some ports of the TurnLights component, as they are not relevant for the case study.

[19] This communication mode, in which writers and readers act quasi-periodically but with no synchronization, was referred to as *Communication by Sampling* in [9].

[20] Note that these two assumptions were not part of the requirements. It is actually not uncommon that some critical assumptions are implicit in requirements documents, which may, at times, become a problem.

Req:

R_1	The delay between brake sensing and activation of the brake lights must not be greater than $25ms$.
R_2	The delay between brake sensing and flashing of the warn lights in case of an emergency brake situation must not be greater than $60ms$.

K^G_{VFB}:

T	τ_{BL}, τ_{TL}
P_T	$P^i(\tau_{BL}) = p^t(\tau_{BL}) =$ ext_pedal, $P^o(\tau_{BL}) =$ {emcy,ext_brake_lamp}, $P^i(\tau_{TL}) =$ {trig_TL,emcy}, $p^t(\tau_{TL}) =$ trig_TL, $P^o(\tau_{TL}) =$ {ext_rear_di_lamp}
R	undefined

K^A_{VFB}:

T	$\tau^{A1}_{VFB}, \tau^{A2}_{VFB}$
P_T	$P^i(\tau^{A1}_{VFB}) = p^t(\tau^{A1}_{VFB}) =$ undefined, $P^o(\tau^{A1}_{VFB}) =$ {ext_pedal}, $P^i(\tau^{A2}_{VFB}) = p^t(\tau^{A2}_{VFB}) =$ undefined, $P^o(\tau^{A2}_{VFB}) =$ {trig_TL}
R	undefined

$\mathbf{C}_{R_1}$:

K_{A_1}	import K^A_{VFB}
L_{A_1}	ext_pedal **occurs each** 20ms **and** trig_TL **occurs each** 20ms
K_{G_1}	import K^G_{VFB}
L_{G_1}	**delay between** ext_pedal **and** ext_brake_lamp **within** [0ms,25ms]

$\mathbf{C}_{R_2}$:

K_{A_2}	import K^A_{VFB}
L_{A_2}	ext_pedal **occurs each** 20ms **and** trig_TL **occurs each** 20ms
K_{G_2}	import K^G_{VFB}
L_{G_2}	**delay between** ext_pedal **and** ext_rear_di_lamp **within** [0ms,60ms]

Table 3: "Req" (on top) displays informally the top-level requirements for *VFB*, as a set $\{R_1, R_2\}$ of requirements. The duration of the time slot is 1μs. K^G_{VFB} is a formalization of the system architecture displayed on Figure 16 as a component sort. The two concrete contracts $\mathbf{C}_{R_i}, i = 1, 2$ import K^G_{VFB} for their guarantee G_i. Assumption A_i has a different sort sharing ports ext_pedal and trig_TL. Since their behavior is assumed to be independent, they are driven by two different tasks τ^{A1}_{VFB}, τ^{A2}_{VFB} respectively, which represent the environment of the *VFB*. Observe that $L_{A_1} = L_{A_2} =_{\text{def}} L_A$. The top-level contract, which formalizes the requirements (under the added assumptions) is $\mathscr{C}_{\text{top\_level}} =_{\text{def}} \mathscr{C}_{R_1} \wedge \mathscr{C}_{R_2}$, where $\mathscr{C}_{R_i}$ is the abstract counterpart of $\mathbf{C}_{R_i}$.

Contracts in saturated form: The calculus of A/G-contracts developed in Section 4 of companion paper [11] requires that contracts are in *saturated form*. Thus, if we write $\mathscr{C}_{\text{top\_level}} = (A, G)$, for A and G two abstract scheduling components, we must work with its saturated form $(A, G \cup \neg A)$. For the case where assumptions do not involve resources, we can use Lemma 5. For our top-level contract $\mathscr{C}_{\text{top\_level}}$ this amounts to replacing the conjunction $L_{G_1} \cap L_{G_2}$ derived from Table 3 by the following saturated one: $(L_{G_1} \cap L_{G_2}) \cup \neg L_A$. We express it by using our pattern language, see Table 4. To avoid heavy notations such as in Table 4, the saturation of contracts will not be

ext_pedal **occurs each** 20ms ; trig_TL **occurs each** 20ms
$\Longrightarrow$
delay between ext_pedal **and** ext_brake_lamp **within** [0ms,25ms] ; **delay between** ext_pedal **and** ext_rear_di_lamp **within** [0ms,60ms]

Table 4: Getting saturated contracts in practice

explicitly performed in the specifications to follow. But we will have to take care of it when invoking the algebra of contracts.

Notational convention: In the sequel, we will not mention, in the description of a sort, its undefined items. For instance, referring to Table 3, we will not mention the item R, nor the undefined input ports $P^i(\tau)$ or trigger port $p^t(\tau)$. Further, we will not explicitly link assumptions and guarantees to their sorts. Instead we take as generic convention that a guarantee of contract $\mathscr{C}_X$ is a language of a scheduling component with sort K_X^G and the sort of the assumption is K_X^A. These conventions are used in the sequel unless otherwise specified.

Step 2 of Process 2: resource agnostic sub-contracts for the VFB

Assuming components BrakeLights and TurnLights shall be implemented by two different suppliers, we now propose sub-contracts specifying a time budgeting for them, regardless of resource allocation. These two contracts are depicted on Table 5. For ports P_T and their association to tasks, the reader is referred to Figure 16. Note that, for these two contracts, assumption and guarantee are composable. To ensure that $\mathscr{C}_{BL}$ and $\mathscr{C}_{TL}$ are correct with respect to the top-level requirements, we must prove the refinement

$$\mathscr{C}_{VFB} \quad \preceq \quad \mathscr{C}_{R_1} \wedge \mathscr{C}_{R_2}, \text{ where } \mathscr{C}_{VFB} =_{\text{def}} \mathscr{C}_{BL} \otimes \mathscr{C}_{TL} \tag{35}$$

Observe that (35) involves abstract contracts. In the sequel, we write $\mathscr{C}_{BL} = (A_{BL}, G_{BL})$ and similarly for other abstract contracts. For $\mathscr{C} = (A, G)$ a contract, we set $\overline{G} =_{\text{def}} G \cup \neg A$ (interpreted as: G in the context of A), so that the saturated form for this contract is $(A, \overline{G})$. Using these notations and Lemma 8, to prove (35) it is enough to prove the following:

$$A \cap \overline{G}_{BL} \cap \overline{G}_{TL} \quad \subseteq \quad G_{R_i}, i = 1, 2 \tag{36}$$

$$A \quad \subseteq \quad (A_{BL} \cap A_{TL}) \cup \neg(\overline{G}_{BL} \cap \overline{G}_{TL}) \tag{37}$$

K^G_{BL}:	T	τ_{BL}
	P_T	$P^i(\tau_{BL}) = p^t(\tau_{BL}) =$ ext_pedal, $P^o(\tau_{BL}) =$ {emcy,ext_brake_lamp}
K^A_{BL}:	T	τ^{A1}_{VFB}
	P_T	$P^o(\tau^{A1}_{VFB}) =$ {ext_pedal}
$\mathbf{C}_{BL}$:	L_A	ext_pedal **occurs each** 20ms
	L_G	**delay between** ext_pedal **and** ext_brake_lamp **within** [0ms,25ms] **and** **delay between** ext_pedal **and** emcy **within** [0ms,5ms]
K^G_{TL}:	T	τ_{TL}
	P_T	$P^i(\tau_{TL}) =$ {trig_TL,emcy}, $p^t(\tau_{TL}) =$ trig_TL, $P^o(\tau_{TL}) =$ {ext_rear_di_lamp}
K^A_{TL}:	T	τ^{A2}_{VFB}, τ_{BL}
	P_T	$P^o(\tau^{A2}_{VFB}) =$ {trig_TL}, $P^i(\tau_{BL}) = p^t(\tau_{BL}) =$ ext_pedal, $P^o(\tau_{BL}) =$ {emcy}
$\mathbf{C}_{TL}$:	L_A	emcy **occurs each** 20ms **with jitter** 5ms **and** trig_TL **occurs each** 20ms
	L_G	**delay between** emcy **and** ext_rear_di_lamp **within** [0ms,50ms]

Table 5: Concrete contracts $\mathbf{C}_{BL}$ and $\mathbf{C}_{TL}$.

where $A =_{\text{def}} A_{R_1} = A_{R_2}$. So far none of the above contracts refer to resources, hence Lemma 5 applies. Returning to the mapping from concrete to abstract scheduling components developed in Definition 5, we see that, if no resource is involved, the mapping $\mathbf{M} \mapsto [\![\mathbf{M}]\!]^{\mathsf{A}}$ is entirely determined by

1. the precedence order between tasks (or ports), and
2. the language L specified as part of $\mathbf{M} = (K, L)$.

Since the precedence order on tasks is unchanged when moving from $\mathbf{C}_{\text{top\_level}}$ shown on Table 3 to $\mathbf{C}_{VFB} =_{\text{def}} \mathbf{C}_{BL} \underline{\times} \mathbf{C}_{TL}$ introduced in Table 5 and (35), it is enough to reason using the pattern-based expressions shown in the tables. Wo do this without further notice in the sequel. Denote by $L^1_{A,TL}$ the first statement of L_A in $\mathbf{C}_{TL}$. We first observe that

$$A \cap \overline{G}_{BL} \cap \overline{G}_{TL} = A \cap G_{BL} \cap (G_{TL} \cup \neg L^1_{A,TL}).$$

Reasoning on latencies, jitter, and periods, allows to infer that $L^1_{A,TL}$ is discharged by $A \cap G_{BL}$:

$$A \cap G_{BL} \quad \subseteq \quad L^1_{A,TL}. \tag{38}$$

More precisely, (38) follows from the implication shown on Table 6. Thus,

$$A \cap \overline{G}_{BL} \cap \overline{G}_{TL} = A \cap G_{BL} \cap G_{TL}$$

ext_pedal **occurs each** 20ms ; trig_TL **occurs each** 20ms ; **delay between** ext_pedal **and** ext_brake_lamp **within** [0ms,25ms] ; **delay between** ext_pedal **and** emcy **within** [0ms,5ms]
$\Longrightarrow$
emcy **occurs each** 20ms **with jitter** 5ms

Table 6: Proving (38).

follows. Again, reasoning on latencies, jitter, and periods, allows to prove that $A \cap G_{BL} \cap G_{TL} \subseteq G_{R_i}$ holds for $i = 1, 2$, which shows (36). Focus next on (37). We have $A_{BL} \cap A_{TL} = A \cap L^1_{A,TL}$. Hence,

$$\begin{aligned}(A_{BL} \cap A_{TL}) \cup \neg(\overline{G}_{BL} \cap \overline{G}_{TL}) &= (A \cap L^1_{A,TL}) \cup \neg(\overline{G}_{BL} \cap \overline{G}_{TL}) \\ &\supseteq (A \cap L^1_{A,TL}) \cup \neg\overline{G}_{BL} \\ \text{(by (38))} \quad &\supseteq (A \cap L^1_{A,TL}) \cup (A \cap \neg L^1_{A,TL}) = A\end{aligned}$$

which proves (37), and thus also (35).

Discussion: The above step of our contract based design methodology leads to the following important observations:

- Verification tools exist that would make the above reasoning automatic. Our purpose, however, in this use case, is to demonstrate that reasoning with contracts does not need to be performed fully automatically. Here, local properties are considered easy for the human and are thus verified manually. In contrast, lifting local properties of contracts to system wide properties is error prone (risk of circular reasoning) and thus relies on the formal algebra of contracts. How assumptions get discharged when composing contracts must be handled carefully, as our analysis of refinement (35) showed.
- Manual reasoning can be complemented by the use of observers. Again, it is important that the use of observers when component composition and viewpoint combination both occur is performed correctly. Our development in Section 3.6 of companion paper [11] provides the formal support for this. For the pattern based language used here, a framework for checking refinement of contracts using an observer based strategy is described in [27].

So far we did not consider target architecture for computing, that is, we did not consider resources. If tasks get allocated to resources, the need to schedule tasks sharing a same resource may be a cause of failure to meet the top-level contract. In the next section, we further refine our contracts to account for resources.

Steps 3 and 4 of Process 2: budgeting resources using contracts

Considering resources in contracts occurs at a next stage of the design process, at which 1) the target platform and its provided resources are known and 2) software components

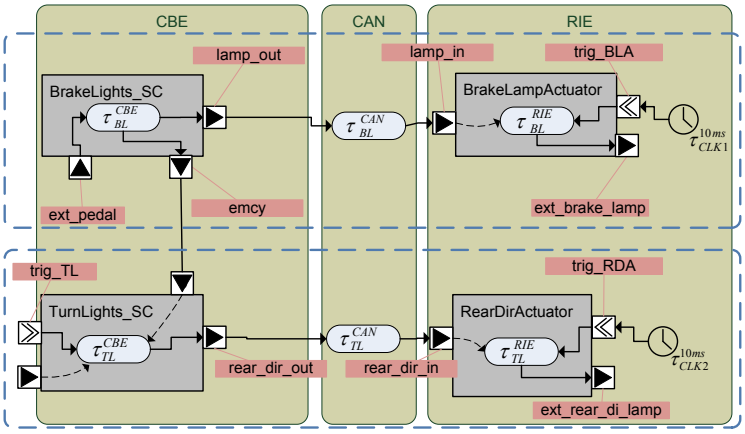


Figure 17: Deploying *VFB* on computing and communication resources.

have been decomposed such that they can be mapped to platform resources.[21]

For our case study we assume a simple target architecture consisting of two ECUs, Central Body ECU (CBE) and Rear Indicator ECU (RIE), connected by a CAN bus. The VFB architecture shown in Figure 16 is further refined and then deployed as the architecture shown in Figure 17. Observe that the deployment architecture does not match the high-level architecture of Figure 16. Instead, the deployment is driven by the separation of the sensing and control parts from the actuation part. The first parts should be deployed on CBE and the latter part on RIE. The blue boxes denote the previous components BrakeLights and TurnLights from the architecture shown on Figure 16. The objective here is to define sub-contracts $\mathbf{C}_{BL}^{\text{res}}$ and $\mathbf{C}_{\text{TL}}^{\text{res}}$, with the architecture shown in Figure 17. We know this will not yield a contract refinement. Nevertheless, we will comply with the steps 3 and 4 of the safe Process 2.

Resource budget: The OEM provides a resource budget for ECUs CBE and RIE and the CAN bus. The resource reservation for the BrakeLights component is modelled by the tasks shown at the top of Figure 17. At the bottom of that figure the tasks of the TurnLights components are depicted. Allocation of tasks to resources is indicated by the brown boxes. The OEM specfies that with a period of $20ms$ a time window of 2–$3ms$ is available for component BrakeLight_SC for sensing and control of the brake light part, and additional $2ms$ for the calculation of a possible emergency brake situation. This is modeled by the task τ_{BL}^{CBE} in Figure 17, which has an execution time interval of $[4ms, 5ms]$ assigned. Further, task τ_{BL}^{CBE} triggers task τ_{BL}^{CAN}, which has an execution time of $250\mu s$. The task τ_{BL}^{RIE} is periodically triggered each $10ms$ by a timer of the operating system (represented by τ_{CLK1}^{10ms}), reading the value of lamp_in and sending control values to the actuator via port ext_brake_lamp. Similar execution time budgets are assigned to the remaining tasks.

Performing Step 3 of Process 2: To perform Step 3 of Process 2 we proceed as follows. We first consider the top-level contract in saturated form $\mathscr{C}_{\text{top}} = (A; (G1 \cap G2) \cup \neg A)$, where requirement R_1 is represented by contract (A, G_1) and similarly for R_2. Since

[21] In fact, Step 2 of Process 2 was not mandatory. The designer can directly move from Step 1 to Step 3 of Process 2. Step 2 was developed for illustration purposes, to show the difference when resources get considered.

it is a resource agnostic contract, Lemma 5 applies and provides us with a concrete form for this contract, denoted by $\mathbf{C}_{\text{top}} = (\mathbf{A}_{\text{top}}, \mathbf{G}_{\text{top}})$. This top-level contract is then modified as follows:

Regarding assumptions: Tasks relevant to the assumption $\mathbf{A}_{\text{top}}$ remain unchanged, namely τ_{VFB}^{A1} and τ_{VFB}^{A2}. We assign them resources that are different, and different from the resources shown on Figure 17. We thus comply with Condition 3 of Lemma 9.

Regarding guarantees: We split tasks τ_{BL} and τ_{TL} as shown on Figure 17, which results in a change of the sort of guarantee $\mathbf{G}_{\text{top}}$ that complies with the Conditions 1 of Lemma 3. This splitting results in the creation of new ports. In addition, we want this new set of tasks to be scheduled according to fixed priorities. The additional information provided when specifying the resource budget (see above) is translated to constraints on the language of the guarantees, which, again, complies with the Conditions 1 of Lemma 3.

Performing this results in the resource aware contract $\mathbf{C}_{\text{top}}^{\text{res}} = (\mathbf{A}_{\text{top}}^{\text{res}}, \mathbf{G}_{\text{top}}^{\text{res}})$ shown in Table 7 that complies with the conditions of Lemma 9. At this point we have implemented Step 3 of Process 2.

Performing Step 4 of Process 2: We now proceed to Step 4, which consists in decomposing this resource aware global contract into sub-contracts.

To this end we first map concrete contract $\mathbf{C}_{\text{top}}^{\text{res}}$ to its abstract form $\mathscr{C}_{\text{top}}^{\text{res}}$, which in turn is amenable of Lemma 10. Besides assigning concurrent resources to tasks, mapping $\mathbf{A}_{\text{top}}^{\text{res}}$ to $A_{\text{top}}^{\text{res}}$ is immediate since resource allocation causes no conflict. Mapping $\mathbf{G}_{\text{top}}^{\text{res}}$ to $G_{\text{top}}^{\text{res}}$, however, requires solving a global schedulability analysis. We thus perform this, which results in strengthened statements in out pattern language, to reflect the semantics of the scheduling policy. Table 8 shows the resulting guarantee.

Then, as a final step, we proceed to the decomposition by using Lemma 10, resulting in contracts $\mathbf{C}_{BL}^{\text{res}}$ and $\mathbf{C}_{TL}^{\text{res}}$ as shown in Table 9. The languages of their assumptions and guarantees are obtained by projecting, respectively, to the sorts $K_{G,BL}^{\text{res}}$, $K_{A,BL}^{\text{res}}$ and $K_{G,TL}^{\text{res}}$, $K_{A,TL}^{\text{res}}$, the language resulting from the scheduling analysis that has been carried out to compute the semantics of $\mathbf{C}_{\text{top}}^{\text{res}}$. The asymmetry between local assumptions A_{BL}^{res} and A_{TL}^{res} is justified by the chosen priorities: Each task τ belonging to the guarantee of contract $\mathbf{C}_{BL}^{\text{res}}$ has a higher priority than any task τ' of the guarantee of contract $\mathbf{C}_{TL}^{\text{res}}$, which is allocated to the same resource as τ.

$K^{\text{res}}_{\text{G,top}}$:	T	$\tau^{CBE}_{BL}, \tau^{CAN}_{BL}, \tau^{RIE}_{BL}, \tau^{CLK1}_{10ms}$ $\tau^{CBE}_{TL}, \tau^{CAN}_{TL}, \tau^{RIE}_{TL}, \tau^{CLK2}_{10ms}$
	P_T	$P^i(\tau^{CBE}_{BL}) = p^t(\tau^{CBE}_{BL})$ = ext_pedal, $P^o(\tau^{CBE}_{BL})$ = {emcy,lamp_out}, $P^i(\tau^{CAN}_{BL}) = p^t(\tau^{CAN}_{BL})$ = lamp_out, $P^o(\tau^{CAN}_{BL})$ = {lamp_in}, $P^i(\tau^{RIE}_{BL})$ = {trig_BLA,lamp_in}, $p^t(\tau^{RIE}_{BL})$ = trig_BLA, $P^o(\tau^{RIE}_{BL})$ = {ext_brake_lamp}, $P^o(\tau^{CLK1}_{10ms})$ = {trig_BLA} $P^i(\tau^{CBE}_{TL}) = p^t(\tau^{CBE}_{TL})$ = trig_TL, $P^o(\tau^{CBE}_{TL})$ = {rear_dir_out}, $P^i(\tau^{CAN}_{TL}) = p^t(\tau^{CAN}_{TL})$ = rear_dir_out, $P^o(\tau^{CAN}_{TL})$ = {rear_dir_in}, $P^i(\tau^{RIE}_{TL})$ = {trig_RDA,rear_dir_in}, $p^t(\tau^{RIE}_{BL})$ = trig_RDA, $P^o(\tau^{RIE}_{TL})$ = {ext_rear_di_lamp}, $P^o(\tau^{CLK2}_{10ms})$ = {trig_RDA}
	R	$R = \{CBE, CAN, RIE\}$ and $\rho(\tau^{CBE}_{BL}) = CBE, \rho(\tau^{CAN}_{BL}) = CAN, \rho(\tau^{RIE}_{BL}) = RIE$

$K^{\text{res}}_{\text{A,top}}$:	T	τ^{A1}_{VFB}, τ^{A2}_{VFB}
	P_T	$P^o(\tau^{A1}_{VFB})$ = {ext_pedal}, $P^o(\tau^{A2}_{VFB})$ = {trig_TL}

$\mathbf{C}^{\text{res}}_{\text{top}}$:	L_A	ext_pedal **occurs each** 20ms **and** trig_TL **occurs each** 20ms
	L_G	**exeT**(τ^{CBE}_{BL}) **within** [4ms,5ms] , **exeT**(τ^{CAN}_{BL}) = 250μs , **exeT**(τ^{RIE}_{BL}) = 2ms **exeT**(τ^{CBE}_{TL}) = 14ms , **exeT**(τ^{CAN}_{TL}) = 250μs , **exeT**(τ^{RIE}_{TL}) = 4ms **prio**(τ^{CBE}_{BL}) > **prio**(τ^{CBE}_{TL}) , **prio**(τ^{CAN}_{BL}) > **prio**(τ^{CAN}_{TL}) , **prio**(τ^{RIE}_{BL}) > **prio**(τ^{RIE}_{TL}) **delay between** ext_pedal **and** ext_brake_lamp **within** [0ms,25ms] **and** **delay between** ext_pedal **and** ext_rear_di_lamp **within** [0ms,60ms]

Table 7: Concrete contract $\mathbf{C}^{\text{res}}_{\text{top}}$ obtained by splitting tasks and adding resources to contract $\mathbf{C}_{\text{top}}$.

$\mathscr{C}^{\text{res}}_{\text{top}}$:	L_G	**exeT**(τ^{CBE}_{BL}) **within** [4ms,5ms] , **exeT**(τ^{CAN}_{BL}) = 250μs , **exeT**(τ^{RIE}_{BL}) = 2ms **exeT**(τ^{CBE}_{TL}) = 14ms , **exeT**(τ^{CAN}_{TL}) = 250μs , **exeT**(τ^{RIE}_{TL}) = 4ms **prio**(τ^{CBE}_{BL}) > **prio**(τ^{CBE}_{TL}) , **prio**(τ^{CAN}_{BL}) > **prio**(τ^{CAN}_{TL}) , **prio**(τ^{RIE}_{BL}) > **prio**(τ^{RIE}_{TL}) **delay between** ext_pedal **and** ext_brake_lamp **within** [4.25ms,15.25ms] **and** **delay between** ext_pedal **and** ext_rear_di_lamp **within** [12.25ms,46.25ms] **and** **delay between** ext_pedal **and** emcy **within** [4ms,5ms] **and** **delay between** emcy **and** ext_rear_di_lamp **within** [8.25ms,41.25ms]

Table 8: Guarantee of contract $\mathscr{C}^{\text{res}}_{\text{top}}$ obtained by scheduling analysis of $\mathbf{C}^{\text{res}}_{\text{top}}$.

$K^{\text{res}}_{G,BL}$:

T	τ^{CBE}_{BL}, τ^{CAN}_{BL}, τ^{RIE}_{BL}, τ^{CLK1}_{10ms}
P_T	$P^i(\tau^{CBE}_{BL}) = p^t(\tau^{CBE}_{BL})$ = ext_pedal, $P^o(\tau^{CBE}_{BL})$ = {emcy,lamp_out}, $P^i(\tau^{CAN}_{BL}) = p^t(\tau^{CAN}_{BL})$ = lamp_out, $P^o(\tau^{CAN}_{BL})$ = {lamp_in}, $P^i(\tau^{RIE}_{BL})$ = {trig_BLA,lamp_in}, $p^t(\tau^{RIE}_{BL})$ = trig_BLA, $P^o(\tau^{RIE}_{BL})$ = {ext_brake_lamp}, $P^o(\tau^{CLK1}_{10ms})$ = {trig_BLA}
R	$R = \{CBE, CAN, RIE\}$ and $\rho(\tau^{CBE}_{BL}) = CBE$, $\rho(\tau^{CAN}_{BL}) = CAN$, $\rho(\tau^{RIE}_{BL}) = RIE$

$K^{\text{res}}_{A,BL}$:

T	τ^{A1}_{VFB}
P_T	$P^o(\tau^{A1}_{VFB})$ = {ext_pedal}

$\mathbf{C}^{\text{res}}_{BL}$:

L_A	ext_pedal **occurs each** 20ms
L_G	**exeT**(τ^{CBE}_{BL}) **within** [4ms,5ms] , **exeT**$(\tau^{CAN}_{BL}) = 250\mu s$, **exeT**(τ^{RIE}_{BL}) = 2ms **prio**(τ^{CBE}_{BL}) > **prio**(τ^{CBE}_{TL}) , **prio**(τ^{CAN}_{BL}) > **prio**(τ^{CAN}_{TL}) , **prio**(τ^{RIE}_{BL}) > **prio**(τ^{RIE}_{TL}) **delay between** ext_pedal **and** ext_brake_lamp **within** [4.25ms,15.25ms] **and** **delay between** ext_pedal **and** emcy **within** [4ms,5ms]

$K^{\text{res}}_{G,TL}$:

T	τ^{CBE}_{TL}, τ^{CAN}_{TL}, τ^{RIE}_{TL}, τ^{CLK2}_{10ms}
P_T	$P^i(\tau^{CBE}_{TL}) = p^t(\tau^{CBE}_{TL})$ = trig_TL, $P^o(\tau^{CBE}_{TL})$ = {rear_dir_out}, $P^i(\tau^{CAN}_{TL}) = p^t(\tau^{CAN}_{TL})$ = rear_dir_out, $P^o(\tau^{CAN}_{TL})$ = {rear_dir_in}, $P^i(\tau^{RIE}_{TL})$ = {trig_RDA,rear_dir_in}, $p^t(\tau^{RIE}_{BL})$ = trig_RDA, $P^o(\tau^{RIE}_{TL})$ = {ext_rear_di_lamp}, $P^o(\tau^{CLK2}_{10ms})$ = {trig_RDA}
R	$R = \{CBE, CAN, RIE\}$ and $\rho(\tau^{CBE}_{TL}) = CBE$, $\rho(\tau^{CAN}_{TL}) = CAN$, $\rho(\tau^{RIE}_{TL}) = RIE$

$K^{\text{res}}_{A,TL}$:

T	τ^{A2}_{VFB}, τ^{CBE}_{BL}, τ^{CAN}_{BL}, τ^{RIE}_{BL}, τ^{CLK1}_{10ms}
P_T	import P_T of $K^{\text{res}}_{G,BL}$, $P^o(\tau^{A2}_{VFB})$ = {trig_TL}
R	import R of $K^{\text{res}}_{G,BL}$

$\mathbf{C}^{\text{res}}_{TL}$:

L_A	trig_TL **occurs each** 20ms **and** emcy **occurs each** 20ms **with jitter** 1ms **and** import L_G of $\mathbf{C}^{\text{res}}_{BL}$
L_G	**exeT**(τ^{CBE}_{TL}) = 14ms , **exeT**$(\tau^{CAN}_{TL}) = 250\mu s$, **exeT**(τ^{RIE}_{TL}) = 4ms **prio**(τ^{CBE}_{BL}) > **prio**(τ^{CBE}_{TL}) , **prio**(τ^{CAN}_{BL}) > **prio**(τ^{CAN}_{TL}) , **prio**(τ^{RIE}_{BL}) > **prio**(τ^{RIE}_{TL}) **delay between** emcy **and** ext_rear_di_lamp **within** [8.25ms,41.25ms]

Table 9: Contracts $\mathbf{C}^{\text{res}}_{BL}$ and $\mathbf{C}^{\text{res}}_{TL}$ obtained by projecting the result of the scheduling analysis that has been carried out to compute the semantics of $\mathbf{C}^{\text{res}}_{\text{top}}$.

As we obtained the guarantees of $\mathbf{C}^{\text{res}}_{BL}$ and $\mathbf{C}^{\text{res}}_{TL}$ by projections of the result of the scheduling analysis, (33) holds. So it remains to prove (34). Denote by $L^1_{A,TL}$ the first statement of L_A in $\mathbf{C}^{\text{res}}_{TL}$ and by $L^2_{A,TL}$ the second statement. Then

$$A^{\text{res}}_{\text{top}} = A^{\text{res}}_{BL} \cap L^1_{A,TL}$$

holds. Focus now on the conditions of (34). For our contracts they translate to

$$A^{\text{res}}_{BL} \cup (L^1_{A,TL} \cap L^2_{A,TL} \cap G^{\text{res}}_{BL}) \quad \supseteq \quad A^{\text{res}}_{\text{top}} \cap \neg(G^{\text{res}}_{BL} \cap G^{\text{res}}_{TL}) \tag{39}$$

$$\neg G^{\text{res}}_{BL} \cup (L^1_{A,TL} \cap L^2_{A,TL} \cap G^{\text{res}}_{BL}) \quad \supseteq \quad A^{\text{res}}_{\text{top}} \cap \neg(G^{\text{res}}_{BL} \cap G^{\text{res}}_{TL}) \tag{40}$$

$$\neg G^{\text{res}}_{TL} \cup A^{\text{res}}_{BL} \quad \supseteq \quad A^{\text{res}}_{\text{top}} \cap \neg(G^{\text{res}}_{BL} \cap G^{\text{res}}_{TL}) \tag{41}$$

$$A^{\text{res}}_{BL} \cap (L^1_{A,TL} \cap L^2_{A,TL} \cap G^{\text{res}}_{BL}) \cup \neg(G^{\text{res}}_{BL} \cap G^{\text{res}}_{TL}) \quad \supseteq \quad A^{\text{res}}_{\text{top}} \tag{42}$$

Since $A^{\text{res}}_{\text{top}} \subseteq A^{\text{res}}_{BL}$ holds, conditions (39) and (41) are satisfied. For the remaining conditions observe that $A^{\text{res}}_{BL} \cap G^{\text{res}}_{BL} \subseteq L^2_{A,TL}$ holds. This can be proved by reproducing the same reasoning as shown in Table 6. With this, (40) and (42) are implied by

$$\neg G^{\text{res}}_{BL} \cup A^{\text{res}}_{\text{top}} \quad \supseteq \quad A^{\text{res}}_{\text{top}} \cap \neg(G^{\text{res}}_{BL} \cap G^{\text{res}}_{TL})$$

$$A^{\text{res}}_{\text{top}} \cap G^{\text{res}}_{BL} \cup \neg G^{\text{res}}_{BL} \cup \neg G^{\text{res}}_{TL} \quad \supseteq \quad A^{\text{res}}_{\text{top}}$$

and both conditions are satisfied. Whence all conditions of (34) are satisfied and consequently $\mathscr{C}^{\text{res}}_{BL} \otimes \mathscr{C}^{\text{res}}_{TL} \preceq \mathscr{C}^{\text{res}}_{\text{top}}$ holds. So this concludes step 4 of Process 2. That means contracts $\mathbf{C}^{\text{res}}_{BL}$ and $\mathbf{C}^{\text{res}}_{TL}$ can be refined and implemented independently from each other, possibly by different suppliers. System integration remains safe.

4.6 Summary and discussion

We have illustrated the use of contracts in the context of AUTOSAR. Our application case was an example of semi-automatic—or semi-manual—use of contracts:

- Manual reasoning was applied for checking undecidable or computationally complex properties of small, local, sub-systems. Manual reasoning is not a formal analysis but it can be reasonably done and cross-checked as part of V&V activities;
- Combining small local proofs into a global system-level proof is error prone if done manually. The algebra of contracts offers formal support for this combination step.

Task scheduling is a resource allocation problem, which, by essence, can only be solved globally. Thus it was certainly not obvious to find a path toward independent development. Supporting the AUTOSAR methodology by a strict contract based approach was not feasible. It is, however, the merit of our approach:

- To properly bound the development steps that do not comply with the rules of contract based design, and
- To explain how risks at system integration can still be mitigated, with a clear and limited additional discipline regarding resource segregation.

A smooth transition of AUTOSAR toward using contracts seems feasible. Indeed, the current AUTOSAR release is expressive enough to represent real-time contracts. Using the concept of timing chains, both end-to-end deadlines as well as assumed response times for component execution and communication can be expressed. We think that the contract framework developed and used throughout this case-study is particularly valuable for compositional reasoning about scheduling of applications distributed over several resources. This kind of reasoning is currently not supported by AUTOSAR and it could be worthwhile to provide support for it in the autosar timing extensions.

On a more general level about contracts and AUTOSAR, it has to be noted that support for formal specification of different viewpoints is not provided—currently there is no established notion of viewpoints and, hence, no anchor to easily migrate from the current setting (timing and safety extensions) to one supporting multiple viewpoints. For example, viewpoints such as power or security are interesting candidates.

4.7 Bibliographical note

This short bibliographical note is divided into two parts: 1) formal studies related to AUTOSAR standard, and 2) compositional task scheduling.

Formal studies related to AUTOSAR: AUTOSAR[22] is a worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry, see [5] for an official introduction to 10 years of AUTOSAR developments. The following sentence taken from this reference properly defines what the scope of AUTOSAR is: "AUTOSAR is not going to standardise the functional internal behaviour of an application, for example algorithms, but the content exchanged between applications." Regarding our concerns, AUTOSAR provides, as part of its timing extension, standards to express the syntax part of a real-time scheduling problem (referring to Definition 1, means to specify sorts) and selected features of the language part (basic timing properties such as periods, latencies etc.). It says nothing regarding how to use these interface data to guarantee safe integration from the timing point of view.

While a comprehensive formal discussion about the execution semantics of AUTOSAR models does not exist, there is some literature addressing parts of it. For example, in [29] the authors proposed an abstract formal model to represent AUTOSAR OS programs with timing protection. Their goal is to compute on the one hand schedulability conditions for a set of periodic tasks and on the other hand allowed maximal preemption times by interrupts while keeping the task set schedulable. Whether this development is compositional is not discussed. In [24] the authors developed a formal model for an AUTOSAR multicore RTOS based on the language Promela. The model includes transition systems for tasks, interrupt service routines and also considers critical sections of task, as well as includes the priority ceiling protocol. Besides enabling model checking and detecting for example deadlocks and livelocks, the primary goal is to generate test cases based on the model. While the model has a certain level of detail with regard to the execution semantics of the operating system, its lacks an explicit notion of time.

Regarding the AUTOSAR timing extensions, in [3] a case study is developed and it is discussed how scheduling analysis techniques apply to an AUTOSAR model enriched by concepts from these timing extensions. The work [2] is interesting since the comparision of MARTE and the AUTOSAR timing extensions found in this paper,

[22] http://www.autosar.org/

also includes a discussion how specifications are composed. In the AUTOSAR timing extensions a notion of composition is only discussed for end-to-end delays assigned to software components. These delays may be decomposed along with the component architecture. At a later step scheduling analysis determines whether required delays for end-to-end communication are indeed satisfied. Different semantics for end-to-end communication delays are supported. [25] provides a formal foundation for them. MARTE, on the other hand, considers an assumption/guarantee style for the specification of non-functional properties, making it also a suitable language to specify contracts.

It should be noted that these works based on the AUTOSAR timing extensions or on the Timing Augmented Description Language (e.g. [42]) stay on the level of timing properties well established in classical scheduling theory like periods, latencies etc. So composition is discussed based on these properies. However, they do not offer a notion of independent implementability that would also encompass the deployment of multiple components to the same ECUs of a target platform. To support this, task scheduling itself has to be addressed in a compositional manner.

Compositional task scheduling: The following text is quoted verbatim from the pioneering work by Insup Lee et al. [48]: "Real-time systems could benefit from component-based design, only if components can be assembled without violating compositionality on timing properties. When the timing properties of components can be analyzed compositionally, component-based real-time systems allow components to be developed and validated independently and to be assembled together without global validation." The reader is referred to this paper for earlier related work from the real-time scheduling community. This paper develops a model of *scheduling interface* collecting the workloads, resources, and scheduling policy, addressing the above quoted objectives. Concepts and techniques used originate from the real-time scheduling community. Specific classes of hard real-time system scheduling problems are considered, namely periodic models and bounded-delay models. This group of authors has further developed the same track with the same techniques, enlarging the classes of real-time scheduling problems considered. This significant body of work is nicely summarized in the tutorial paper [1] and implemented through the CARTS tool for compositional analysis of real-time systems [43]. One interesting application case concerns the scheduling of ARINC partitions [22]. The approach generally followed in these works is the following. The schedulability problem is structured as a hierarchy of subproblems. The solution of each subproblem is summarized using some form of interface (depending on the particuliar approach) and, at the next upper level, the amount of additional resource and deadline conditions is computed as the solution of some optimization problem.

Lothar Thiele and co-workers have developed for real-time scheduling an elegant algebraic framework called the *Real-Time Calculus* (RTC) [52, 53, 36]. This algebraic framework builds on top of the foundational work on max-plus algebra, initially developed in the formerly available book Synchronization and Linearity, (1992), by F. Baccelli, G. Cohen, G. J. Olsder and J. P. Quadrat. This framework was developed to endow the class of event graphs (the subclass of conflict free Petri nets) with an algebra of *linear* input-output transfer functions. Components of the RT Calculus are thus linear transfer functions in this max-plus algebra and interface behaviors are expressed as *arrival curves*, which specify lower and upper bounds for event arrivals. Our Figure 10 could indeed be interpreted in this way, where wires carry flows of events driven by this algebra—in fact, our mapping from concrete to abstract scheduling components (Definition 5) is a trick for representing Thiele's max-plus algebra by a dataflow composi-

tion. Reference [52] considers real-time interfaces where assumptions and guarantees are expressed by means of arrival curves on inputs and outputs, respectively. Refinement of such interfaces is characterized and a parallel composition is defined. So-called *adaptive* interfaces are proposed in which arrival curves are propagated throughout the network of components, compositionally. This topic is further developed in [51]. A blending of this model with timed automata is studied in [33] together with a mapping of RTC-based real-time interfaces to timed automata. The very interesting work [35] applies and develops similar techniques for distributed heterogeneous time-triggered automotive systems.

Our work in this section has its roots in [50, 45, 46, 49]. Our aim is different from the previous set of references and complements it nicely. Whereas previous references considered restricted classes of real-time scheduling problems and addressed them with complete algorithmic solutions, our model makes no restriction on the class of scheduling problems, except that the allocation of tasks to resources is static and so are precedence conditions. In turn, we provide a full fledged contract algebra decomposing system wide problems into an architecture of (smaller) local scheduling problems, seen as "proof obligations", but, we do not discuss how these proof obligations can be automatically or algorithmically checked. These proof obligations can be either delegated to existing real-time scheduling algorithms (see the previous references) or addressed manually, as we did here in Section 4.5, or by using a model checking engine for a restricted class of scheduling contracts, as we did in Section 4.5. That is, we favor generality over full automatization—still, the lifting of local proofs to system-wide solutions is supported by our contract algebra. In doing so we specifically target OEM-supplier relations in a supplier chain, by providing support for decomposing a system-level scheduling contract into sub-contracts for suppliers while guaranteeing safe system integration, and support for fusing different viewpoints on the system using contract conjunction—either relavant to schedulability analysis or to different aspects of the system. With comparison to the above four references [50, 45, 46, 49], our development of Section 4.2 fully matches the meta-theory of contracts of companion paper [11]. This way, we inherit refinement and parallel composition (like some previous real-time interface models offer), and also a *conjunction,* which allows to specify real-time scheduling problems in a "requirement engineering" style.

We would like to conclude this discussion by the following question the reader may want to ask: did we really reuse the material of our companion "theory" paper [11]? Can we really claim that the theory developed there has a wide applicability? The point is that the development made in Section 4.2 cannot be claimed trivial: is it always the case? Here are our answers: First, the reader should not underestimate the cost of upgrading a framework of components (offering a $\times$ but no refinement nor conjunction) to a corresponding framework of contracts—the whole paper [11] illustrates how much it can cost. Second, and most importantly, the solid foundations of [11] provide very precise guidelines regarding what a framework of components should offer. These guidelines were indeed extremely useful in developing our model of scheduling components.

5 Conclusion

This paper complements companion paper [11] by developing two application cases.

The first application case, the parking garage example, examplified the benefits of using contracts in requirement engineering. We illustrated the use of viewpoints to

support modular development of top-level requirements. We showed how responsibilities can be accurately specified, by distinguishing guarantees offered from assumptions on the context of use throughout the entire contract based development process. Our encoding of requirements as Modal Interfaces formalizes the requirements and allows for their execution and exploration. This was the basis for giving formal support to important certification related properties of requirements such as consistency, compatibility, and completeness. While moving from a contract specified at a certain level, to an architecture of sub-contracts at the next level, is generally meant to be performed by hand and then formally verified using refinement checks, we illustrated in our example the possibility to synthesize this refinement step automatically, by only providing a structural specification of the refined architecture, à la SysML. This was made possible thanks to the availability of the MICA tool [17].

Open issues and future work remain regarding the use of multiple viewpoints. Our viewpoints in the parking garage example were homogeneous in nature. Developing and then fusing heterogeneous viewpoints such as function, safety/reliability, schedulability analysis, quantitative resources, and more, remains largely open. Heterogeneity has not been much considered. It is usually handled at the level of component frameworks. As an example, this leads to embedding function and safety/reliability frameworks into a unifying, more general, framework from which both views can be recovered by specialization. Doing so results in issues of computational complexity or even undecidability, due to the move to a more general modeling framework. We conjecture that our meta-theory, by being "framework agnostic", helps to manipulate contracts that are specified as pairs of heterogeneous sets of environments and implementations. This remains to be explored.

Our second application case addressed a key part of the AUTOSAR development process. AUTOSAR advocates a design methodology by which the functions, structured into tasks, are first designed independently of the computing and communication infrastructure, assuming a virtual AUTOSAR run time environment. We studied the key step by which time budgets are then allocated to tasks and computing resources are assigned. Lack of formal support in AUTOSAR methodology makes this step difficult today, with little guaranteed at system integration phase. We showed the benefit of using contracts for this step. To this end, we developed an adaptation of the Assume/Guarantee contracts of companion paper [11] that we call *scheduling contracts.*

Our study illustrated the semi-formal/semi-manual use of contracts, which we believe, is key to enable a smooth transition to contract based design. A contract engine (such as the MICA tool presented in Section 3) can be used in combination with both manual reasoning and dedicated formal verification engines as the basis for future development that will make contracts main stream.

References

[1] Madhukar Anand, Sebastian Fischmeister, and Insup Lee. A comparison of compositional schedulability analysis techniques for hierarchical real-time systems. *ACM Trans. Embedded Comput. Syst.*, 13(1):2, 2013.

[2] Saoussen Anssi, Sébastien Gérard, Stefan Kuntz, and François Terrier. AUTOSAR vs. MARTE for enabling timing analysis of automotive applications. In *SDL 2011: Integrating System and Software Modeling - 15th International SDL Forum Toulouse, France, July 5-7, 2011. Revised Papers*, pages 262–275, 2011.

[3] Saoussen Anssi, Sara Tucci Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. Enabling scheduling analysis for AUTOSAR systems. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2011, Newport Beach, California, USA, 28-31 March 2011*, pages 152–159, 2011.

[4] Chetan Arora, Mehrdad Sabetzadeh, Lionel C. Briand, and Frank Zimmer. Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns. In *4th IEEE International Workshop on Requirements Patterns, RePa 2014, Karlskrona, Sweden, August 26, 2014*, pages 1–8, 2014.

[5] AUTOSAR consortium. 10 years AUTOSAR. Technical report, AUTOSAR, 2013. available from http://www.autosar.org/fileadmin/files/events/10yearsautosar/ATZextra_AUTOSAR_-_THE_WORLDWIDE_AUTOMOTIVE_STANDARD_FOR_EE_SYSTEMS.pdf.

[6] Felice Balarin, Jerry R. Burch, Luciano Lavagno, Yosinori Watanabe, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT01)*, pages 129–133, Monterey, CA, November 7–9, 2001. IEEE Computer Society, Los Alamitos, CA, USA.

[7] Felice Balarin, Roberto Passerone, Alessandro Pinto, and Alberto L. Sangiovanni-Vincentelli. A formal approach to system level design: Metamodels and unified design environments. In *Proceedings of the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE05)*, pages 155–163, Verona, Italy, July 11–14, 2005. IEEE Computer Society, Los Alamitos, CA, USA.

[8] Sebastian S. Bauer, Alexandre David, Rolf Hennicker, Kim Guldstrand Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Moving from specifications to contracts in component-based design. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2012.

[9] Albert Benveniste, Anne Bouillard, and Paul Caspi. A unifying view of loosely time-triggered architectures. In Luca P. Carloni and Stavros Tripakis, editors, *Proceedings of the 10th International conference on Embedded software, EMSOFT 2010, Scottsdale, Arizona, USA, October 24-29, 2010*, pages 189–198. ACM, 2010.

[10] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple viewpoint contract-based specification and design. In *Proceedings of the Software Technology Concertation on Formal Methods for Components and Objects, FMCO'07*, volume 5382 of *Lecture Notes in Computer Science*, pages 200–225. Springer, October 2008.

[11] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Larsen. Contracts for System Design: Theory. Research Report RR-8759, Inria, 2015.

[12] Luca Benvenuti, Alberto Ferrari, Leonardo Mangeruca, Emanuele Mazzi, Roberto Passerone, and Christos Sofronis. A contract-based formalism for the specification of heterogeneous systems. In *Proceedings of the Forum on Specification, Verification and Design Languages (FDL08)*, pages 142–147, Stuttgart, Germany, September 23–25, 2008.

[13] Gerard Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.

[14] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[15] Jerry R. Burch, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Overcoming heterophobia: Modeling concurrency in heterogeneous systems. In *Proceedings of the 2nd International Conference on Application of Concurrency to System Design (ACSD01)*, pages 13–32, Newcastle upon Tyne, UK, June 25–29, 2001. IEEE Computer Society, Los Alamitos, CA, USA.

[16] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2005.

[17] Benoît Caillaud. Mica: A Modal Interface Compositional Analysis Library, October 2011. http://www.irisa.fr/s4/tools/mica.

[18] Werner Damm, Angelika Votintseva, Alexander Metzner, Bernhard Josko, Thomas Peikenkamp, and Eckard Bde. Boosting reuse of embedded automotive applications through rich components. In *Proceedings of FIT 2005 - Foundations of Interface Technologies*, 2005.

[19] Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. METROII: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12(1s):49:1–49:31, March 2013.

[20] Douglas Densmore, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. A platform-based taxonomy for ESL design. *IEEE Design and Test of Computers*, 23(5):359–374, May 2006.

[21] AUTOSAR development cooperation. Specification of Timing Extensions. http://www.autosar.org, Release 4.2.1, 2014.

[22] Arvind Easwaran, Insup Lee, Oleg Sokolsky, and Steve Vestal. A compositional scheduling framework for digital avionics systems. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*, pages 371–380, 2009.

[23] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, 2003.

[24] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. Formal model-based test for autosar multicore rtos. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 251–259, April 2012.

[25] Nico Feiertag, Kai Richter, Johan Nordlander, and Han Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *IEEE Real-Time System Symposium (RTSS), Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, November 2008.

[26] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley, 2003.

[27] Tayfun Gezgin, Raphael Weber, and Maurice Girod. A Refinement Checking Technique for Contract-Based Architecture Designs. In *Fourth International Workshop on Model Based Architecting and Construction of Embedded Systems, ACES-MB'11*, volume 7167 of *Lecture Notes in Computer Science*. Springer, October 2011.

[28] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. Sangiovanni-Vincentelli, and M. Di Natale. Software Components for Reliable Automotive Systems. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 549–554, march 2008.

[29] Yanhong Huang, João F. Ferreira, Guanhua He, Shengchao Qin, and Jifeng He. Deadline analysis of AUTOSAR OS periodic tasks in the presence of interrupts. In *Formal Methods and Software Engineering - 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1, 2013, Proceedings*, pages 165–181, 2013.

[30] INCOSE. Incose systems engineering handbook, 2010. http://www.incose.org/ProductsPubs/products/sehandbook.aspx.

[31] S. Karris. *Introduction to Simulink with Engineering Applications*. Orchard Publications, 2006.

[32] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[33] Kai Lampka, Simon Perathoner, and Lothar Thiele. Component-based system design: analytic real-time interfaces for state-based component implementations. *STTT*, 15(3):155–170, 2013.

[34] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[35] Martin Lukasiewycz, Reinhard Schneider, Dip Goswami, and Samarjit Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference, ASP-DAC 2012, Sydney, Australia, January 30 - February 2, 2012*, pages 665–670, 2012.

[36] Shanmuga Priya Marimuthu and Samarjit Chakraborty. A framework for compositional and hierarchical real-time scheduling. In *12th IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2006), 16-18 August 2006, Sydney, Australia*, pages 91–96, 2006.

[37] Object Management Group (OMG). Unified Modeling Language (UML) specification. [online], http://www.omg.org/spec/UML/.

[38] Object Management Group (OMG), . A UML profile for MARTE, beta 1. OMG Adopted Specification ptc/07-08-04, OMG, August 2007.

[39] Object Management Group (OMG), . System modeling language specification v1.1. Technical report, OMG, 2008.

[40] J. Hudak P. Feiler, D. Gluch. The Architecture Analysis and Design Language (AADL): An Introduction. *Software Engineering Institute (SEI) Technical Note, CMU/SEI-2006-TN-011*, February 2006.

[41] Roberto Passerone, Imene Ben Hafaiedh, Susanne Graf, Albert Benveniste, Daniela Cancila, Arnaud Cuccuru, Sébastien Gérard, Francois Terrier, Werner Damm, Alberto Ferrari, Leonardo Mangeruca, Bernhard Josko, Thomas Peikenkamp, and Alberto Sangiovanni-Vincentelli. Metamodels in Europe: Languages, tools, and applications. *IEEE Design and Test of Computers*, 26(3):38–53, May/June 2009.

[42] Marie-Agnès Peraldi-Frati, Arda Goknil, Morayo Adedjouma, and Pierre Yves Gueguen. Modeling a BSG-E automotive system with the timing augmented description language. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part II*, pages 111–125, 2012.

[43] Linh T. X. Phan, Jaewoo Lee, Arvind Easwaran, Vinay Ramaswamy, Sanjian Chen, Insup Lee, and Oleg Sokolsky. CARTS: a tool for compositional analysis of real-time systems. *SIGBED Review*, 8(1):62–63, 2011.

[44] Alessandro Pinto, Alvise Bonivento, Alberto L. Sangiovanni-Vincentelli, Roberto Passerone, and Marco Sgroi. System level design paradigms: Platform-based design and communication synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):537–563, July 2006.

[45] Philipp Reinkemeier and Ingo Stierand. Compositional timing analysis of real-time systems based on resource segregation abstraction. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro Cesar Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 International Embedded Systems Symposium, IESS 2013, Paderborn, Germany, June 17-19, 2013. Proceedings*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 181–192. Springer, 2013.

[46] Philipp Reinkemeier and Ingo Stierand. Real-Time Contracts - A Contract Theory Considering Resource Supplies and Demands. Reports of SFB/TR 14 AVACS 100, SFB/TR 14 AVACS, July 2014. http://www.avacs.org.

[47] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[48] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004), 5-8 December 2004, Lisbon, Portugal*, pages 57–67, 2004.

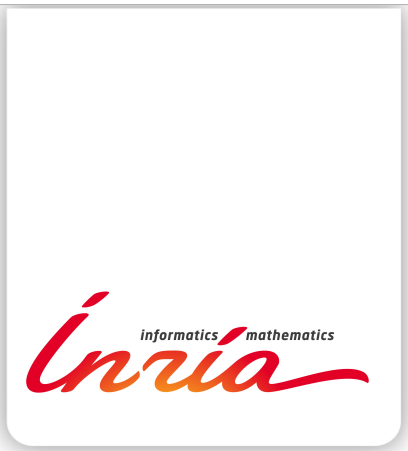
[49] Ingo Stierand, Philipp Reinkemeier, and Purandar Bhaduri. Virtual integration of real-time systems based on resource segregation abstraction. In *Formal Modeling and Analysis of Timed Systems - 12th International Conference, FORMATS 2014, Florence, Italy, September 8-10, 2014. Proceedings*, pages 206–221, 2014.

[50] Ingo Stierand, Philipp Reinkemeier, Tayfun Gezgin, and Purandar Bhaduri. Real-time scheduling interfaces and contracts for the design of distributed embedded systems. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, pages 130–139, 2013.

[51] Nikolay Stoimenov, Samarjit Chakraborty, and Lothar Thiele. Interface-based design of real-time systems. In *Advances in Real-Time Systems (to Georg Färber on the occasion of his appointment as Professor Emeritus at TU München after leading the Lehrstuhl für Realzeit-Computersysteme for 34 illustrious years).*, pages 83–101, 2012.

[52] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. Real-time interfaces for composing real-time systems. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, pages 34–43, 2006.

[53] Ernesto Wandeler and Lothar Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006), 4-7 April 2006, San Jose, California, USA*, pages 243–252, 2006.



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399