Towards a Generic Modeling Language for Contract-Based Design

Johannes Iber, Andrea Höller, Tobias Rauter, and Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Inffeldgasse 16, Graz, Austria
{johannes.iber, andrea.hoeller, tobias.rauter, christian.kreiner}@tugraz.at

***Abstract*—Component-based and model-driven engineering are key paradigms for handling the ever-increasing complexity of technical systems. Surprisingly few component models consider extra-functional properties as first class entities.**

Contract-based design is a promising paradigm, which has the potential to fill this shortage of methods for dealing with extra-functional properties. By defining the concept of using assumptions in order to determine the environment, and by using the concept of guarantees to state what a component provides to the environment, it enables the analyzability of components and compositions in advance and during system execution.

With this work, we aim to create the base for a pragmatic model-driven method that provides reusable modeling concepts for contracts targeting arbitrary extra-functional properties. Furthermore, we expand the current state-of-the-art of contract-based design by introducing the concept of a finite state machine, where single states consist of several valid contracts. It is also assumed that these modeling language features will ease the use of contract-based design. Additionally, we demonstrate the applicability of the presented modeling concepts on an exemplary use case from the automotive domain.

***Index Terms*—Metamodeling, contract-based design, extra-functional properties, component models**

I. Introduction

Numerous industrial sectors are currently confronted with massive difficulties originating from managing the increasing complexity of systems. The automotive industry, for instance, has an annual increase rate of software-implemented functions of about 30% [1]. This rate is even higher for avionics systems [2]. Additionally, this development of systems is not restricted to software, as we are facing a so-called Internet of Things, where the number of physical devices is expected to expansively explode [3]. New challenges regarding complexity of systems emerge caused by this dramatic increase of diverse hardware/software, possible interactions and distributed intelligences [4].

Component-based engineering is today a widely recognized and well-established paradigm for tackling complexity of systems [5]. Together with model-driven engineering, it forms a potentially powerful union to construct, analyze, and deploy systems.

But still, modern component models are flawed. As shown by Crnković et al. [5], astonishingly few (software) component models are addressing extra-functional properties (e.g. timing, safety, memory consumption, etc.) as first class entities. However, these properties are essential for composing a component-based system predictable and safe. Management of extra-functional properties is thus still one of the core challenges faced by component-based design [6].

Contract-based design is a promising paradigm for filling or narrowing this gap, [7]. It captures the behavior of a specific functional or extra-functional property in relationship with the environment of a component. Despite the existence of a mathematical groundwork [7] [8] and exemplary applications, a standard and generic metamodel for contract-based design does not yet exist.

With this work, we provide pragmatic modeling concepts that pave the way for integrating contract-based design into component models of systems. We present a metamodel fragment for contracts which target arbitrary single extra-functional properties. Furthermore, we introduce the concept of a finite state machine, where single states constitute valid contracts. This concept extends the current state-of-the-art regarding contract-based design. We show the applicability of these modeling concepts by using an example from the automotive domain. The target component of the use case is a simplified electronic steering column lock, which we examine with respect to the extra-functional properties safety and timing.

The remainder of this paper is structured as follows: the next Section provides a brief overview of the background to this work. In Section III the proposed modeling concepts are introduced. Subsequently, a use case demonstrating the applicability of these concepts is described in Section IV. Finally, concluding remarks and future research opportunities are given in Section V.

II. Background and Related Work

Here, we give an overview of system abstractions and properties. After this, we briefly explain contract-based design. Finally, we summarize the related work concerning contract-based design, which is also the motivation setting for this work.

A. System Abstractions and Properties

According to Jantsch [9], there are four main different abstraction models or views concerning embedded system engineering. First is the *computational model*, which describes the observable behavior of a system or of its single parts

(hardware, software components), i.e. the relationship between inputs and outputs [10]. Second, a *data model* exists that provides notations for information (e.g. integer, boolean). Third, a *time model* is needed to constitute the causality of events. Fourth, a *communication model* is established to specify how components interact. This model forms the top-level system behavior.

In the context of the properties of systems the literature distinguishes between functional and extra-functional (also known as non-functional) properties. Functional properties describe the function of a system or component, i.e. behavior, input or output data types. Extra-functional properties provide additional information and give a better insight into the behavior and capability of a system or component [6]. A wide range of such properties exists, e.g. safety, security, portability, performance. Since these issue from humans, there is no method to determine a priori which extra-functional properties exist in a system [6] [11].

B. Contract-based Design

Contract-based design usually sees a component as an abstraction, a hierarchical entity that represents a single unit of design [8] [12]. Therefore in the context of contract-based design a component can represent, for instance a module, a composition, a complex system or even a physical device.

The essence of this paradigm is to decompose a component into different independent views referred to as contracts, which capture the behavior of a target functional or extra-functional property under certain conditions [12] [13]. This approach significantly reduces the complexity of design and verification, because the single properties become manageable.

Informally, a contract is a set of assumptions and guarantees.

An assumption asserts what a contract expects from the component environment (this can include interactions with other components). Additionally, an assumption provides a certain context for the guarantees. The condition contained in an assumption can reference for instance input data, events or system properties. In general, the available variables are set or inferred by the analysis environment.

A guarantee describes what a component provides to the environment if the corresponding assumptions become valid. In the simplest case a guarantee states that a component just works under the constrained context. More complex contracts define limits for instance for output data, environment characteristics or extra-functional properties such as timing.

Historically, contract-based design is influenced by Meyer's design-by-contract principle [14] for object-oriented software [7]. The main difference is that contract-based design goes much further and provides means to integrate components in the design hierarchy [10]. This is achieved through capturing the context by assumptions (which may include platforms, other components, etc.), under which a component behaves as specified by the guarantees. Furthermore, a system can be viewed by selecting only appropriate contracts of interest.

Fig.1 illustrates that contract-based design not only allows the analyzing of components on a horizontal design level (e.g. interaction between software modules, hardware devices, etc.). It also enables analyzing to take place on a vertical level between different kinds of abstraction [7].

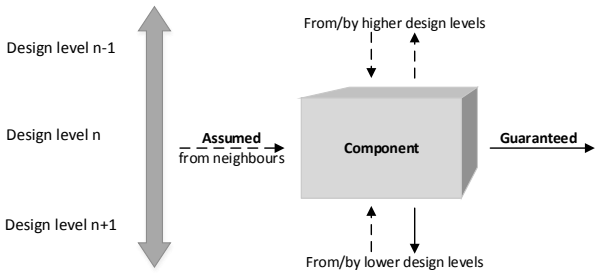

Fig. 1. Contract assumptions and guarantees for a component (Adapted from [15])

A solid mathematical groundwork already exists for this as provided by several authors, including Benveniste et al. [7], and Sangiovanni-Vincentelli et al. [8].

Promising applications of contract-based design have been shown for several domains. For instance, this paradigm has been demonstrated for smart integrated energy management systems [16], aircraft electric power systems [12], mixed-signal integrated circuits [17], and automotive [18] [7]. Despite these examples, contract-based design is still at its infancy [19].

Little work has been done towards establishing a generic standard metamodel for contract-based design. Warg et al. [20] presents a prototype modeling tool for contracts, but their work solely focuses on safety integrity levels.

C. Summary of Contract-Based Design

There exist a few approaches for realizing contract-based design, for instance the contract-based model developed in the framework of the SPEEDS project [13]. The problem is that state-of-the-art approaches either tackle single extra-functional properties, or take a relatively theoretical approach without concrete modeling examples or tool implementations. A survey concerning the certification of safety-relevant systems, carried out by the SafeCer consortium [21], shows that only a few companies are actually using contracts for components. And where this is the case they are relying on Meyer's design-by-contract principle on a programming language level.

III. Proposed Modeling Language Concepts

In this section, we explain concepts which are necessary for a pragmatic modeling language that targets contract-based design.

A. Target System Abstractions and Properties

With the following concepts, we aim at enriching the computational, time, and communication models of a system. Furthermore, the data model plays an important role, as it provides data types and notations, which could be used by contracts.

In the context of properties, our intention is to capture extra-functional properties and not necessarily functional behavior. We take the view that functional behavior is better described by other well-established methods than by the use of many different contracts.

The issue of what extra-functional properties we are aiming at, is dependents on the specific use case or context under which the following language features are used. These concepts may be applied for a wide range of different extra-functional properties (e.g. security, safety, timing, expected hardware/platform, memory consumption, many-core environment, etc.). But certainly not for all of them, since no silver bullet exists for dealing with every extra-functional property [11].

B. Pragmatic Modeling Langugage Features

In the following, we present a modeling concept for contracts. Additionally, we introduce the concept of a finite state machine for contracts.

1) Contract: Fig.2 illustrates our proposed metamodel for contracts. We separate a contract into two parts. A *Contract Declaration* represents a type for *Contract Definitions*. It states the available parameters, assumptions and guarantees. Furthermore, it represents the target extra-functional property. A *Contract Definition* captures the unique behavior concerning the target extra-functional property of a component in relationship to its environment.

Parameters can represent properties of the execution environment, data ports or events. They can be used by *Constraint Definitions* in order to set the specific assumption or guarantee. *Parameter Declarations* are used to specify that a variable of a specific data type may exist, but the concrete value has to be defined by the realizing *Contract Definition*. This can be useful for data arrays where the data points contained are individual for each component.

In the context of assumptions and guarantees, it is possible for a *Constraint Declaration* to set expected data types. The associated *Constraint Definition* must provide an expression where the resulting data type equals one of the expected types.

As we can see in Fig.2, we use the placeholders *Variable* for parameters, *DataType* for data types, and *Expression* for constraint expressions. These elements should be provided by a suitable constraint language or referable by the language that is used for the *Constraint Definition* expressions.

2) Finite State Machine for Contracts: Single contracts are sometimes not adequate for representing extra-functional properties. As we explain with our presentation in the following Section IV, cases exist where the behavior of a component - including extra-functional properties - changes over time or as a result of specific events. We thus expand the theory of contract-based design and capture such differences concerning contracts by applying the concept of a finite state machine. The idea is to have a finite state machine, where the single states may contain several currently valid contracts. The state machine itself operates on parameters provided by the environment or the internal states of a component.

Fig.3 illustrates our proposed metamodel for such a state machine. We again use the concept of declaration and definition in order to separate the specification and actual instance of a so-called contract state machine.

A *Contract State Machine Declaration* constitutes allowed *Contract Declarations*, concrete parameters and declarations of parameters which need to be defined by corresponding *Contract State Machine Definitions*.

Parameters are supposed to be used by *Contract State Machine Events* within constraint expressions, which trigger transitions to other *Contract State Machine States*. Such a state contains zero to infinite *Contract Definitions*.

Again, the metamodel elements *Variable*, *DataType* and *Expression*, refer to an arbitrary constraint language.

The actual semantics of a contract state machine depends on the target extra-functional properties and is determined by convention. It may be that entering a state implies that only those *Contract Definitions* it contains are valid. An alternative convention would be, that all visited *Contract Definitions* are valid except that a current *Contract Definition* overrides a former visited one by using the same *Contract Declaration*.

IV. Use Case

In this Section we show the application of our modeling concepts as presented on an exemplary use case from the automotive domain. First we give an overview of the target component and system. After that, we apply contracts together with a contract state machine. Finally, we discuss the use case presented.

A. Example - Electronic Steering Column Lock

Fig.4 illustrates a simplified electronic steering column lock (ESCL). Such locks are mandatory for cars in many countries. The Electronic Control Unit (ECU) decides whether to lock the steering column based on the input signals *Key State* and *Velocity*. These signals may be transmitted by a CAN bus or separate connections. If the ECU decides to lock the steering column, an actuator is activated which inserts the bolt into the steering column. Otherwise, the ECU decides to hold or eject the bolt.

There are several extra-functional properties which are worth considering in a system of this kind. In the following, we apply the modeling concepts presented for the extra-functional properties safety and timing. In the safety context we capture the data on whether the component ESCL is performing *normally*, is in a *failure* state, or *recovering* from a failure state. A failure state can be induced for instance by faulty transmitted data or other misbehaving components. Further to this we capture the data on how long it takes to execute the lock or unlock mechanism in two separate contract definitions.

B. Declarations

According to our metamodel concepts, the first step is to specify general declarations for components. Such declarations are known to contract checkers, interpreters or model transformers in advance. Fig.5 illustrates declarations for a

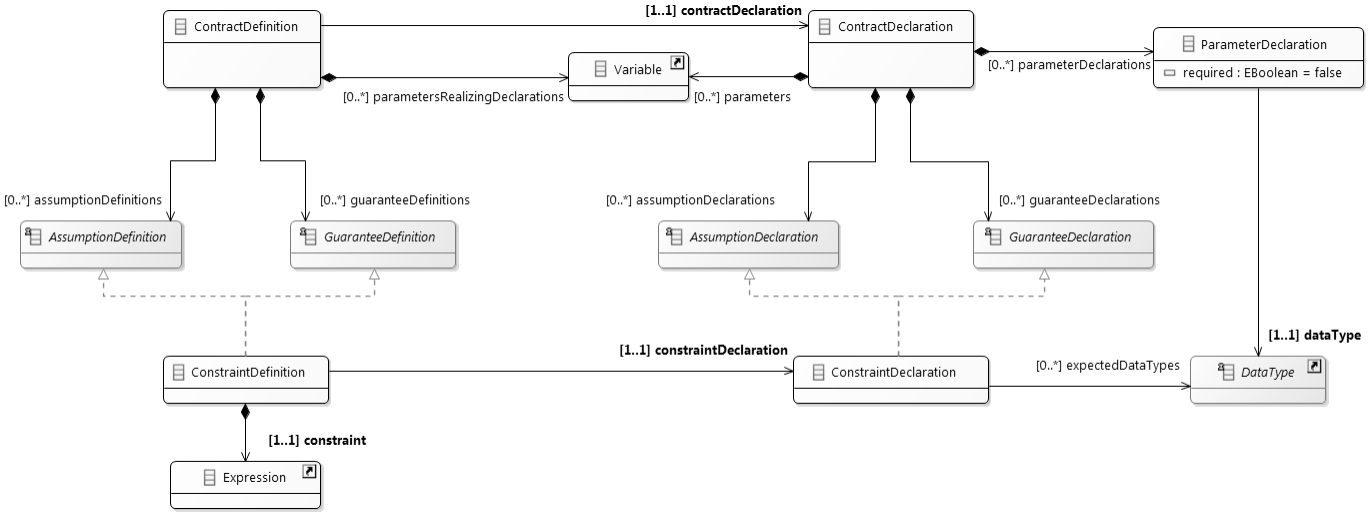

Fig. 2. Proposed Metamodel for Contract Declarations and Definitions

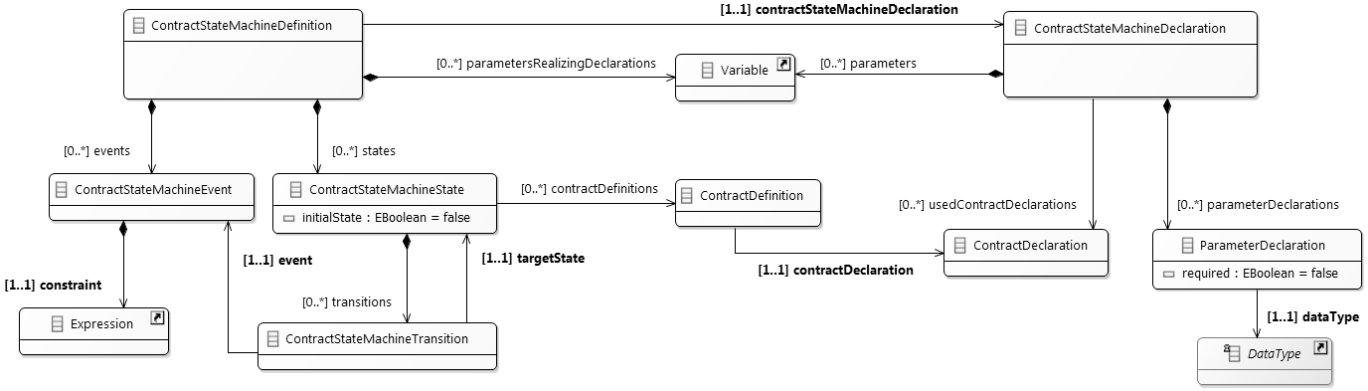

Fig. 3. Proposed Metamodel for Contract State Machine Declarations and Definitions

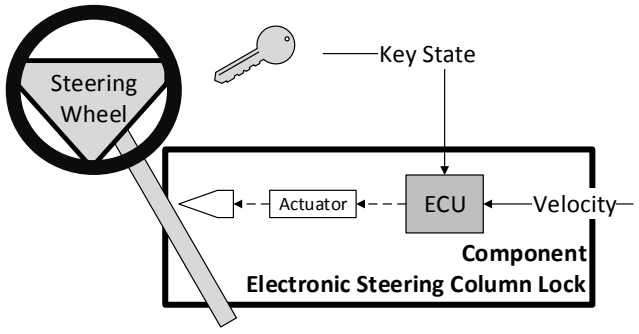

Fig. 4. Example Component - Electronic Steering Column Lock

Contract Declaration "Component Safety Status"

Assumption Enabled : Boolean

Guarantee State : SafetyStateEnum

Contract Declaration "Component Timing"

Parameter key_state : Boolean = **false**
Parameter velocity : Float = 0.0
...

Assumption Environment : Boolean

Guarantee Execution Time : Time

Contract State Machine Declaration "Component Contract Behavior"

Uses contract declaration "Component Safety Status“
Uses contract declaration "Component Timing“

Parameter declaration key_state : Boolean
Parameter declaration velocity : Float
Parameter component_restart : Boolean = **false**
...

Fig. 5. Use Case Declarations for the target extra-functional properties

component's safety status and timing. Additionally, we specify a contract state machine declaration that is used to capture the behavior of a component in order to set valid contracts.

The contract declaration *Component Safety Status* assumes whether the component of interest is enabled and guarantees a certain safety state to the environment. The available types for this guarantee are restricted by the data type *SafetyStateEnum*, which contains the literals *NORMAL*, *FAILURE*, and *RECOVER* (not shown in Figure 5).

The contract declaration *Component Timing* is used to

guarantee a specific execution time for certain assumed environments. The parameters *key_state* and *velocity* are provided by the analysis environment. The boolean parameter *key_state* indicates whether the ignition system is activated (boolean value true), while the parameter *velocity* states the current speed of the car. A comprehensive contract declaration would provide several other parameters, which may be obtained for instance by a CAN bus or observed from the condition of a system. The issue of which of these parameters are actually used by the assumption *Environment* depends on the component. When this assumption results in a boolean true, the guarantee *Execution Time* becomes valid.

Furthermore, contract definitions of these declarations can be used by the single states of the contract state machine *Component Contract Behavior*. Here again the parameters contained are obtained by the analysis environment or transmitted by the available connections. For instance, the parameter *component_restart* must be set by the analysis environment or by the described component. These parameters are used by a contract state machine definition in order to specify the events for state transitions.

C. Definitions

We now present how the declarations from above are used.

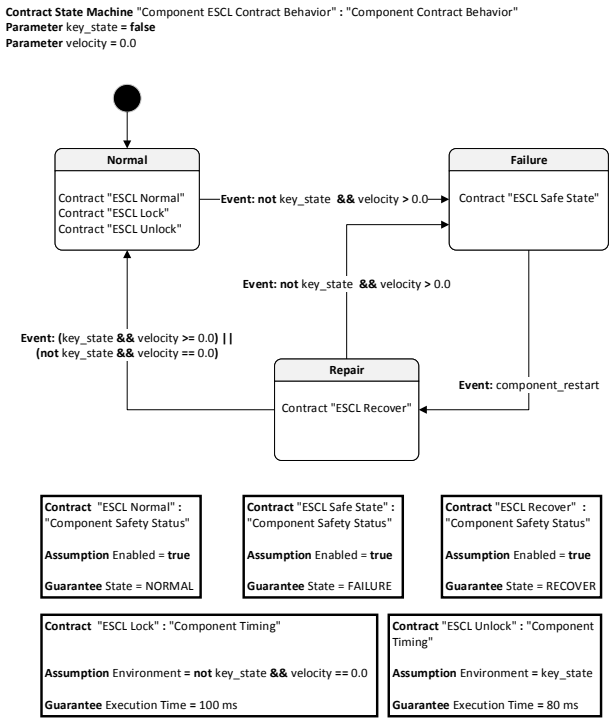

Fig. 6. Contract State Machine and Contract Definitions of the ESCL Example

Fig.6 illustrates a contract state machine definition which sets the valid contract definitions according to the current state. The parameters are realizations of the parameter declarations declared by the contract state machine declaration *Component Contract Behavior* and are initialized to default values.

The initial state of this example is state *Normal*. Within this state, we can guarantee the execution time in respect to the locking and releasing mechanism. Furthermore, the contract *ESCL Normal* determines the safety state *NORMAL* to the environment. Whenever an abnormal event occurs such as there is no key but the car is moving, the contract state machine changes to the state *Failure*. In this state we cannot constitute the execution time of the ECSL and the contract *ESCL Safe State* becomes valid. After the component ESCL restarts, the state machine changes to the state *Repair*, which is reflected by the contract *ESCL Recover*. When the recover procedure was successful, the state machine changes to the state *Normal*, where the contained contracts become valid again, otherwise the state machine switches back to state *Failure*.

D. Discussion of the Use Case

We have shown how our contract modeling features can be used as presented on a simplified use case. It is imaginable that this example can be further advanced to capture the target and other extra-functional properties in more detail.

Note that we do not capture the actual functional behavior of the component ESCL. We rather use the functional behavior of the environment in order to determine how the target extra-functional properties timing and safety status of the component are changing and what guarantees are valid in that state. The semantics of the contract state machine we present is such that a new state invalidates the former visited contracts. The assumptions and guarantees of the *Contract Definitions* must be either automatically gathered by a measurement software or issued by humans.

Such a contract state machine can be used for two purposes.

One purpose is that a system becomes analyzable in advance, also with respect to composability. A model checker could simulate such a system and calculate the different expected safety states. Another model checker would be able to estimate the overall timing of a system.

The second purpose would be that a detection mechanism observes and constitutes the single states during runtime of a system and takes appropriate action based on predetermined contracts.

V. Conclusion and Future Work

In this paper, we presented concepts for modeling contracts and showed in a use case how these concepts can be applied.

The vision is to have a generic modeling language for specifying contract types and contract instances. By using the term generic we mean contracts that are suitable for at least a substantial number of extra-functional properties.

We introduced the concept of splitting a contract into a declaration and a definition. For analysis purposes a specific contract declaration would be known by a model checker or code generator beforehand. It declares the available parameters, assumptions and guarantees, while a contract definition uses such a declaration to define the actual behavior of a target extra-functional property.

Furthermore, we introduced the concept of a contract state machine which is basically a finite state machine where the single states represent different contract definitions. This concept is necessary, because a component may behave in different ways depending on the input data, environment properties or specific events. For instance, the timing of a component may be different depending on its previous processed data. It may also be different if the environment has changed. Such changes may require different valid contracts.

Concerning our future work, we are currently working on a configurable constraint modeling language, inspired by OCL [22], which we want to use for setting assumptions and guarantees. The idea is to have a constraint language where language elements, such as an if expression or a boolean operation, can be disabled and is afterwards not usable by an assumption or guarantee. This is useful, in our opinion, to simplify the construction of contract checkers or interpreters, because not all concepts of an expression language need to be considered and handled properly. It would also provide a user with direct feedback concerning what language elements are allowed for use.

Additionally, the presented modeling features for contracts do not consider composition, refinement, and conjunction of contracts as described theoretical by Benveniste et al. [7]. We are still working on finding pragmatic and usable metamodel solutions for these concepts.

After building this in a form suited to our use case metamodel for contract-based design, we are planning to develop a thin generic UML profile [23] for contracts and contract state machines.

This profile will be aligned with the existing OMG specifications MARTE [24] and SysML [25]. As mentioned by Selić and Gérard [26], a natural complementarity exists between these two profiles. We are of the view that a UML profile for contract-based design would benefit from concepts such as the physical types of MARTE or the constraint blocks of SysML. Not using such existing and standardized modeling concepts would be like reinventing the wheel.

The advantages of such a UML profile for contracts could be manifold. The most important one is, that it would allow the rise of specialized analyzing tools of different vendors which target single extra-functional properties. The input of such tools would depend, in such an ideal ecosystem, on the same UML profile for contract-based design.

REFERENCES

[1] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," *Computer*, vol. 42, no. 4, Apr. 2009.

[2] P. Feiler, J. Hansson, D. de Niz, and L. Wrage, "System Architecture Virtual Integration: An Industrial Case Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, CMU/SEI-2009-TR-017, 2009.

[3] M. Miller, *The Internet of Things: How Smart TVs, Smart Cars, Smart Homes, and Smart Cities Are Changing the World*. Pearson Education, 2015.

[4] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, Sep. 2012.

[5] I. Crnkovic, S. Sentilles, V. Aneta, and M. R. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, Sep. 2011.

[6] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković, "Integration of Extra-Functional Properties in Component Models," in *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2009.

[7] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. L. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for Systems Design," INRIA, Rennes, France, Tech. Rep., 2012.

[8] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, Jan. 2012.

[9] A. Jantsch, *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation.* San Francisco, Amsterdam: Morgan Kaufmann, 2004.

[10] N. Kajtazovic, "A Component-based Approach for Managing Changes in the Engineering of Safety-critical Embedded Systems," Ph.D. dissertation, Graz University of Technology, 2014.

[11] I. Crnkovic, M. Larsson, and O. Preiss, "Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes," in *Architecting Dependable Systems III*. Springer Berlin Heidelberg, 2005.

[12] P. Nuzzo, Huan Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A Contract-Based Methodology for Aircraft Electric Power System Design," *IEEE Access*, vol. 2, 2014.

[13] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," 2008.

[14] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, Oct. 1992.

[15] A. Rajan and T. Wahl, Eds., *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Vienna: Springer Vienna, 2013.

[16] M. Maasoumy, P. Nuzzo, and A. Sangiovanni-Vincentelli, "Smart Buildings in the Smart Grid: Contract-Based Design of an Integrated Energy Management System," 2015.

[17] P. Nuzzo, A. Sangiovanni-Vincentelli, Xuening Sun, and A. Puggelli, "Methodology for the Design of Analog Integrated Interfaces Using Contracts," *IEEE Sensors Journal*, vol. 12, no. 12, Dec. 2012.

[18] N. Kajtazovic, C. Preschern, A. Höller, and C. Kreiner, "Constraint-Based Verification of Compositions in Safety-Critical Component-Based Systems," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. Studies in Computational Intelligence. Springer International Publishing, 2015.

[19] P. Nuzzo and A. Sangiovanni-Vincentelli, "Lets Get Physical: Computer Science Meets Systems," in *From Programs to Systems. The Systems perspective in Computing*. Springer Berlin Heidelberg, 2014.

[20] F. Warg, B. Vedder, M. Skoglund, and A. Soderberg, "Safety ADD: A Tool for Safety-Contract Based Design," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov. 2014.

[21] O. Bridal, R. Mader, A. Geven, E. Schoitsch, H. Martin, M. Larramendi, A. Aristimuno, A. Fritsch, E. Vaumorin, M. Bordin, A. Solinas, A. Martelli, I. Korago, A. Levcenkovs, F. Joakim, R. Land, A. Söderberg, P. Conmy, and M. Illarramendi, "State-of-practice and state-of-the-art agreed over workgroup," Tech. Rep., 2011. [Online]. Available: http://www.safecer.eu/images/pdf/pSafeCer_D1.0.1StateOfThePracticeAndTheArt.pdf

[22] Object Management Group (OMG), "Object Constraint Language Version 2.4," 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4/

[23] ——, "Unified Modeling Language (UML)," 2015. [Online]. Available: http://www.omg.org/spec/UML/Current

[24] ——, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1," 2011. [Online]. Available: http://www.omg.org/spec/MARTE/

[25] ——, "OMG Systems Modeling Language (OMG SysML) Version 1.3," 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[26] B. Selić and S. Gérard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*, 2014.