Contracts-refinement proof system for component-based embedded systems

Alessandro Cimatti[a], Stefano Tonetta[a]

[a]*Fondazione Bruno Kessler, Trento, Italy*

Abstract

Contract-based design is an emerging paradigm for the design of complex systems, where each component is associated with a contract, i.e., a clear description of the expected interaction of the component with its environment. Contracts specify the expected behavior of a component by defining the *assumptions* that must be satisfied by the environment and the *guarantees* satisfied by the component in response. The ultimate goal of contract-based design is to allow for compositional reasoning, stepwise refinement, and a principled reuse of components that are already pre-designed, or designed independently.

In this paper, we present fully formal contract framework based on temporal logic[1]. The synchronous or asynchronous decomposition of a component into subcomponents is complemented with the corresponding refinement of its contracts. The framework exploits such decomposition to automatically generate a set of proof obligations. Once verified, the conditions allow concluding the correctness of the architecture. This means that that the components ensure the guarantee of the system and the system ensures the assumptions of the components. The framework can be instantiated with different temporal logics. The proof system reduces the correctness of contracts refinement to entailment of temporal logic formulas. The tool support relies on an expressive property specification language, conceived for the formalization of embedded system requirements, and on a verification engine based on automated SMT techniques.

Keywords: contract-based design, temporal logics, embedded systems, OCRA

1. Introduction

Embedded systems are continuously growing in complexity, and carry out critical functions. This calls for effective and rigorous methods to find defects early in the development process, to guarantee safety and correctness, and to reduce the costs of certification.

Component-based design is a very promising paradigm, amenable to compositional reasoning and to reuse of components. In order to tame the complexity of embedded systems, the component implementations can be abstracted with properties that specify

[1]A preliminary version of this framework has been presented in [1].

the behavioral aspects that are relevant for the system-level properties. In this settings, the cost of formal methods is alleviated by compositional verification and reuse of proof.

Contract-based design, first conceived for software specification [2] and now applied also to embedded systems [3, 4, 5, 6, 7, 8, 1, 9, 10], structures the component properties into contracts. A contract specifies the properties assumed to be satisfied by the component environment (assumptions), and the properties guaranteed by the component in response (guarantees).

Contract-based design comes with multiple advantages: it supports stepwise refinement, compositional reasoning, and a principled reuse of pre-designed or independently designed components. This approach is adopted in several recent projects for embedded systems, such as SafeCer (`www.safecer.eu`), which exploits contracts to enable a compositional certification and reuse of pre-qualified components.

In this paper, we give full account of a contract framework where contracts are tightly integrated within an architectural decomposition of the system, and are specified with temporal logics. The framework provides a formal notion of correctness of the contract refinement, which is reduced to checking the validity of set of (necessary and sufficient) proof obligations. The generated proof obligations are formulae in the same temporal logic used to express the contracts, and can be decided by means of suitable decision procedures.

The approach, which extends and completes the framework presented preliminarily in [1], encompasses both synchronous and asynchronous composition of components, and is based on a generic notion of traces, so that it can be instantiated with different temporal logics.

This framework has some distinguishing features. First, the refinement checks are tightly integrated with the architecture decomposition flow of safety-critical applications. This includes taking into account the connections of components that play a fundamental role in the contract refinement. This allows for an early validation of the choices underlying the architectural decomposition and requirements delegation. Second, our approach supports the automated production of refinement checks. Finally, compared to other approaches (that are typically either theoretic or limited to simple specification patterns), our specialization supports more expressive and general properties.

The approach has been implemented in the tool OCRA [11], which was used to analyze systems from various application domains. The tool instantiates the framework by allowing the specification with two (of many possible) logics: Linear Temporal Logic (LTL) [12], which models discrete traces, and HRELTL, a variant of LTL defined in [13], which models hybrid traces (combining discrete and continuous transitions). In the case study proposed in [7], used as a running example in this paper, OCRA was able to pinpoint some inaccuracies in the original formulation.

This paper is structured as follows. In Section 2, we discuss relevant related work. In Section 3, we present a motivating case study and the envisaged contract-based design that requires the methods proposed in the paper. In Section 4, we present our contract-based framework in terms of sets of generic traces. In Section 5, we define the proof obligations proving that they characterize the correctness of contract refinement. In Section 6, we instantiate the trace-based framework described in the previous

section using temporal logics and we describe the support in terms of tool and concrete language to the framework. In Section 7, we evaluate the approach on the case study. In Section 8, we draw some conclusions and outline directions for future work.

2. Related work

Contracts have been first defined in the context of object-oriented programming by Meyer [2]. For software programs, assumptions and guarantees are represented respectively by preconditions and postconditions of functions: preconditions define the assumptions that the function caller must satisfy at the entry point of the function; postconditions define the guarantee that the function provider must satisfy at the exit point. Other used assertions are class- and loop-invariants. In concurrent programs, the interaction between service clients and providers is more complex, and require the use of assertions over execution traces.

As in other works, such as [7, 14, 9], we separate the architectural design (where the primitive components are specified as black boxes) from the behavioral models of the component (that may be specified in different languages). In these works, however, contracts distinguish between assumption and guarantee only for enabling assume-guarantee reasoning. Thus the semantics of a contract is simply the implication "if the assumption holds, also the guarantee holds". In our work, instead, following the seminal work of Meyer [2] and the recent applications to embedded systems, contracts represent two distinct properties, one for the environment of the component and one for the component itself. If the environment does not satisfy the assumption, then the architecture is not correct.

This paper builds on the recent works presented in [8], from which we inherit some formal notions for contract-based reasoning. While [8] describes when and how contracts can be *composed* (in a bottom-up design process), we focus on the verification conditions necessary to prove that a contract is correctly refined by a specific *decomposition* (in a top-down design process). Moreover, we detail how the architectural connections play a fundamental role in the composition of components and therefore also in the contract refinement. Based on this insight, we provide new theoretical results tailored to the verification of contracts decomposition.

The idea of contract decomposition and refinement is also provided in [7]. Our approach has two important differences. First, [7] adopts a semantics of a contract in terms of implementations, without reference to the notion of environment, and the refinement is defined as trace-set inclusion. This approach is therefore missing the notion of assumption as constraint for the environment and allows to refine a contract by strengthening the assumptions. Another important difference with respect to [7] is that it uses contract patterns converted into automata, while our approach is based on temporal logics.

As in [3], we use a trace-based semantics for contracts. However, we use a concrete property specification language to express the assumption and guarantees of the contracts. Also, differently from [3], we do not assume that contracts are in normal form, but we reduce to normal form just for the refinement checks. This may become important when the negation of assertion is not possible (as for timed and hybrid automata) or

when performing syntactic checks of realizability which can be hindered by the normal form having to deal with the (possibly un-realizable) negation of the assumption.

The composition of components have been studied in many works on interface theories and automata (see, e.g., [15]). As in [3, 8, 1], we define the satisfaction and refinement of contracts in terms of trace-based language inclusions. Checking that an implementation is receptive to any input of the environment satisfying the assumption is not in the scope of the present framework. Rather, the framework is focused on checking that contracts are correctly refined from the language point of view.

We build our framework on existing advanced techniques for property specifications and analysis [13, 16]. However, the contract-based framework can be instantiated with any temporal logic with a linear model of time. In particular, MTL [17, 18] can be used to formalize real-time requirements and is supported by different tools also for by graphical languages [19]. Although decidable variants of MTL exist, only few tool supports the verification of satisfiability and they are mainly based on bounded satisfiability [20] (similarly to our support of HRELTL).

This paper extends the previous version published in [1] by revising the trace-based contract framework and giving full account of syntax and semantics of the underlying component model, detailing the role of port connections in the contracts refinement, and extending the framework to asynchronous composition of subcomponents. In fact, in [1], the framework is simplified by assuming that all symbols are global. In reality, the ports are local to every component and the connections among them play a fundamental role in the refinement.

3. Contract-based design

3.1. Motivating example

In this section, we informally describe our contract-based design through an example taken from [21], where it is used for giving guidelines on how to conduct safety assessment in avionics. The same example has been used in [7] to illustrate the formalization of contracts.

The benchmark describes a Wheel Braking System (WBS), which takes care of translating the brake signals of the braking pedals into physical brake of the wheel. The brake pedal position is electrically fed to the braking computer, which in turn produces corresponding control signals to the brakes. This computer is named the Braking System Control Unit (BSCU). The Preliminary System Safety Assessment leads to the design of a primary and secondary sub-system to perform the wheel braking function. We call each sub-system a subBSCU. Therefore, the system takes in input two redundant `Pedal_Pos` brake positions and outputs a pressure on the `Brake_Line`. See Figure 1 for a high-level picture.

The property under analysis of the WBS system is that the maximum delay between the brake signal and its execution is 10 time units. This is guaranteed under the assumptions that (i) the two inputs always carry the same value, and that (ii) there is never more than one fault in the system.

The system is composed of a Brake System Control Unit (`BSCU`) and an `Hydraulic` subsystem. In this benchmark, the `BSCU` component is refined into three components:

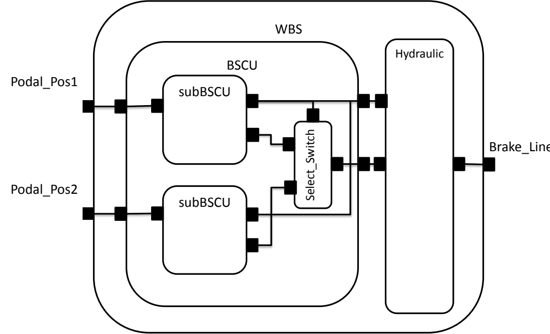

Figure 1: Component-based structure of the Wheel Braking System.

two redundant instances of subBSCU, and a Select_Switch that selects the back-up subBSCU when the signal of the primary subBSCU is no more valid due to a failure. The two subBSCU are further refined into two subcomponents, a Monitor and a Command, and the failure is due to a fault in either of these two subcomponents.

3.2. Design flow

The design flow, depicted in Figure 2, shows the WBS example of Figure 1 at the different levels of abstraction. It starts with the view of the system as a whole black box with ports to interact with its environment. Then, it is is decomposed into BSCU and Hydraulic components. The BSCU is in turn decomposed into two redundant subBSCU and a switch. The decomposition also defines how the ports of the component being decomposed are mapped down into the decomposition. For example, the "left" ports of the WBS are mapped onto the "left" ports of the BSCU.

Each component in the hierarchy is associated with a set of *contracts*, depicted in green, specifying the acceptable behaviors for the component and its environment. More precisely, a contract is a pair *assumption-guarantee*: the assumption is a property that the component expects to be satisfied by its environment; the guarantee is a property that the component ensures, provided that its environment does behave according to the assumptions.

Contracts can be refined, following the decomposition of components. For example, the contracts of the WBS are refined by some contracts of the BSCU and the Hydraulic subcomponents. The framework guarantees that, under specific conditions (corresponding to correct contract refinement), if the contracts of the subcomponents hold, then the contract of the parent component also holds.

A contract can be fulfilled by an implementation, e.g. a concrete module whose behaviors satisfy the guarantees, as long as the environment behaves according to the assumptions. In Figure 2, if the implementations (corner-less rectangles) are proved to satisfy the contracts of the leaf components (i.e. subBSCU, Select_Switch, and

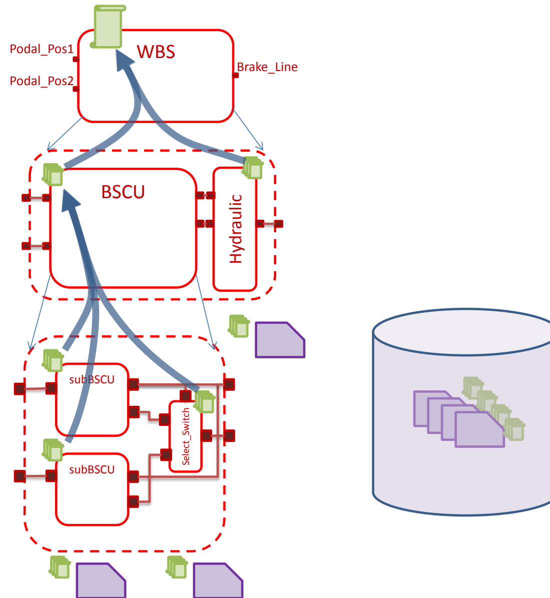

Figure 2: Component-based design flow. Contracts (papyrus shape) are attached to components. The arrows represent the refinement of a component contract by the contracts the subcomponents.

`Hydraulic`), then the resulting system satisfies the top-level contract for `WBS`. Within this framework, reuse can be supported by maintaining a library of implementations together with the corresponding contracts. Instead of checking whether the implementation satisfies the leaf component contract (e.g. as a model checking problem), it may be easier to check whether it is refined by the contracts stored in the library, that the implementation is known to satisfy.

There are several points supporting the idea of contract-based reasoning. The first one is that it provides a clean framework for compositional verification of global properties of an existing systems: the contracts are used as landmarks for the proof, so that in the end it is possible to obtain the guarantee for the global property out of the proof that each of the components satisfies its contracts, and that the individual contracts entail the global property. The second is that it supports stepwise refinement, so that when a component is decomposed, the corresponding specification is decomposed at the same time. This means for example that the allocation of functions to subcomponents is decided and proved correct at the moment of decomposition, i.e. way before the behavioral descriptions are provided. The third reason is the support of component reuse: the proof of refinement holds for any component implementation satisfying the contracts of the component leaves.

4. Trace-based framework

4.1. Traces

The basic structure underlying the semantics of contracts and components is a *trace*. A trace represents the observable part of a run (execution) of a component. It consists of the events and values of data ports that an observer can see watching at a component from outside.

Formally, a trace is defined over a set of variables. It represents the evolution of the value of these variables along time. Different notions of traces exist depending on the model of time. The contract-based framework presented in this paper is independent of such choice. The related tool support instantiates the framework using different notions of traces.

We present the framework based on a generic notion of traces. A generic trace is seen as a sequence of states. Each state contains an assignment to the variables, and can encompass information on time as in timed state sequences [18]. We just assume to have a notion of composition $\times$ and projection π such that, given two traces σ_1 and σ_2 respectively on V_1 and V_2, the projection $\pi_{V_1}(\sigma_1 \times \sigma_2)$ is equivalent to σ_1, and $\pi_{V_2}(\sigma_1 \times \sigma_2)$ is equivalent to σ_2.

In the following we overview some specific notions of traces.

Given a set of variables V over a domain D, a *discrete trace* over V is a simple sequence of assignments to the variables V in D (without explicit information on time). For example, given $V = \{x\}$ a discrete trace over V can be $x = 0, x = 1, x = 4, \ldots$. Composition and projection are defined state by state in a straightforward way.

Timed traces contain, besides the value of variables, also the value of time along the sequence. We define a timed trace as a sequence $\langle s_0, I_0\rangle, \langle s_1, I_1\rangle, \langle s_2, I_2\rangle, \ldots$ such that:

- the intervals are adjacent, i.e., for all $i \in \mathbb{N}$, the upper endpoint of I_i is equal to the lower endpoint of I_{i+1};
- $I_0 = [0, 0]$ and the intervals cover $\mathbb{R}^+$: $\bigcup_{i\in\mathbb{N}} I_i = \mathbb{R}^+$.

Note that time is weakly monotonic, i.e. it is possible for two subsequent states to have a common time point. This case allows to model instantaneous changes.

Hybrid traces extend timed traces considering also continuous and differentiable real functions as the domain of the variables. We partition V in three types of variables: continuous variables, discrete variables and events.

Continuous variables are interpreted with differentiable functions. Given a state of the trace and the related time interval I, the value of a continuous variable varies along the time points of the interval. Moreover, if an endpoint of I is open, the interpretation of the variable in that endpoint must be the same as the adjacent state (while it can be a discontinuous point for the derivative).

Discrete variables are interpreted with constant functions. They can change value only during the discrete instantaneous changes. We actually do not distinguish between the constant function and the value itself, so that if a discrete variable v is interpreted in a state with the constant function c, we say that v is assigned with c. Therefore,

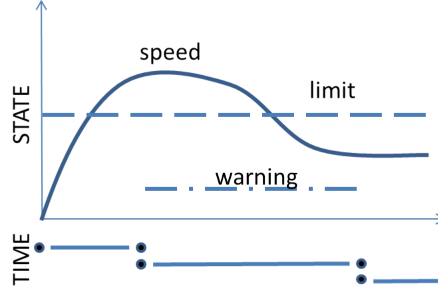

Figure 3: Example of hybrid trace. The variable “speed” evolves continuously along time; the variable “warning” changes instantaneously; “limit” remains constant.

discrete variables may be of type real, integer, or Boolean, depending on the domain of constant functions they are restricted to.

Finally, events are Boolean variables that may be true only before discrete instantaneous changes. For simplicity, we consider them state variables, even if they would be more properly characterized by the transition between the two states in the discrete change.

To understand better the concept of hybrid trace, let us consider the example depicted in Figure 3: the variable “speed” is a continuous variable and thus evolves with a continuous function, while the variable “limit” is a discrete variable that changes instantaneously, and thus evolves with a step function. In this example, “limit” is constant and never changes. Suppose the trace represents the movement of a train; in this scenario the train is initially in a still position and starts moving, it passes the speed limit and then brakes until it reaches a speed below the limit.

In the bottom part of the figure, the sequence of time intervals is shown. It starts with a singular interval $[0, 0]$, followed by an open interval in which “speed” changes, followed by two consecutive singular points where “warning” instantaneously changes.

In order to define the composition and projection of hybrid traces, let us introduce the concept of sampling refinement. Given two hybrid traces σ_1 and σ_2, we say that σ_1 is a sampling refinement of σ_2 (denoted with $\sigma_1 \preceq \sigma_2$) if the σ_1 can be obtained by σ_2 by splitting some open intervals into two or more intervals. Therefore, the two traces have the same discrete changes and the same interpretation of the variables.

Given two traces σ_1 and σ_2, respectively on V_1 and V_2, with $V_1 \cap V_2 = \emptyset$, we define the product $\sigma_1 \times \sigma_2$ as the trace σ over $V_1 \cap V_2$ such that for all i, $\sigma(i)(x) = \sigma_1'(i)(x)$ if $x \in V_1$ and $\sigma(i)(x) = \sigma_2'(i)(x)$ if $x \in V_2$, where σ_1' and σ_2' are the greatest sampling refinements of σ_1 and σ_2 respectively such that σ_1' and σ_2' have the same sequence of intervals.

Given a trace σ over the variables V and a subset V' of V, we define the projection of σ over V' (denoted as $\pi_{V'}(\sigma)$) as the trace σ' such that for all i for all $x \in V'$, $\sigma'(i)(x) = \sigma(i)(x)$. Note that given two traces σ_1 and σ_2 respectively on V_1 and V_2, the projection $\pi_{V_1}(\sigma_1 \times \sigma_2)$ is a sampling refinement of σ_1, and $\pi_{V_2}(\sigma_1 \times \sigma_2)$ is a sampling refinement of σ_2.

4.2. Components

In this section, we define the semantics of contracts and components in terms of generic traces. Therefore, given a set V of variables, we denote with $Tr(V)$ the set of all possible traces over V.

A *component* S is described with a set V_S of ports, which are the variables representing the relevant information of the component that are visible from outside it. Given a set $\mathbb{S}$ of components, we denote with $V_{\mathbb{S}}$ the union of the ports, i.e., $V_{\mathbb{S}} = \bigcup_{S \in \mathbb{S}} V_S$.

An *implementation* of a component S is defined as a subset of traces over V_S, i.e., a subset of $Tr(V_S)$. An *environment* of S is also defined as a subset of $Tr(V_S)$ (since V_S are the variables at the interface of S).

4.3. Contracts

We assume to be given an assertion language, which can be automata-based or logic-based, for which every assertion A has associated a set of variables V_A and a semantics $[\![A]\!]$ as a subset of $Tr(V_A)$. We also assume to have the operations of complementation (denoted with $\neg$), intersection (denoted with $\wedge$), union (denoted with $\vee$), and projection over a subset of variables V' (denoted with $\exists V(\cdot)$). Formally, $[\![\neg A]\!] = Tr(V) \setminus [\![A]\!]$, $[\![A \vee B]\!] = [\![A]\!] \cup [\![B]\!]$, $[\![A \wedge B]\!] = [\![A]\!] \cap [\![B]\!]$, $\exists V'(A) = \{\sigma \mid$ there exists $\sigma' \in [\![A]\!]$ and $\pi_{V'}(\sigma') = \sigma\}$. We say that $A \models B$ iff $[\![A]\!] \subseteq [\![B]\!]$.

Given a component S, a contract for S is a pair $\langle A, G \rangle$ of assertions over V_S representing respectively an *assumption* and a *guarantee* for the component. Let $C = \langle A, G \rangle$ be a contract of S. Let I and E be respectively an implementation and an environment of S. We say that I is a implementation satisfying C iff $I \cap [\![A]\!] \subseteq [\![G]\!]$. We say that E is an environment satisfying C iff $E \subseteq [\![A]\!]$. We denote with $\mathcal{I}(C)$ and with $\mathcal{E}(C)$, respectively, the implementations and the environments satisfying the contract C.

Two contracts C and C' are *equivalent* (denoted with $C \equiv C'$) iff they have the same implementations and environments, i.e., iff $\mathcal{I}(C) = \mathcal{I}(C')$ and $\mathcal{E}(C) = \mathcal{E}(C')$.

A contract $C = \langle A, G \rangle$ is in normal form iff $\overline{[\![A]\!]} \subseteq [\![G]\!]$. We denote with $nf(C)$ the assertion $\neg A \vee G$. The contract $\langle A, nf(C) \rangle$ is obviously in normal form, and is equivalent to (i.e., has the same implementations and environments of) C [3].

4.4. System architecture and components decomposition

A *system architecture* is defined by a tree of components. The root of the tree is called the system component. The leafs of the tree are called atomic (or basic or primitive) components. The tree structure is given by the decomposition of each non-atomic component.

The *architectural decomposition* is a pair $\rho = \langle Sub, \gamma \rangle$ where, for every component S:

- $Sub(S)$ is the set of subcomponents of S such that $S \notin Sub^*(S)$, where Sub^* is the transitive closure of the function Sub; $Sub(S) \neq \emptyset$ iff the S is a composite component;

- $\gamma(S)$ represents the connections among the subcomponents and the delegation of V_S to the ports of the subcomponents.

Note that we avoid to distinguish between component type and component instances to simplify the notation. Actually, the subcomponents of a component type may be instances of the same component type and the system is a component instance. We simply see two component instances as two distinct components with the same structure and renamed ports and subcomponents.

The decomposition may be synchronous or asynchronous. The architecture can also contain both types of decomposition. The composite implementation of a decomposed component S (denoted with CI_S^ρ) consists of the composition of the implementations of its subcomponents $Sub(S)$. Similarly, the composite environment of a subcomponent $U \in Sub(S)$ (denoted with CE_U^ρ) consists of the composition of the implementations of the other subcomponent $Sub(S) \setminus \{U\}$ and the environment of S. The formal definition of CI_S^ρ and CE_U^ρ depend on the type of composition, which can be synchronous or asynchronous. In the following sections, we detail such definitions.

4.4.1. Synchronous decomposition

The synchronous composition of component implementations is given by their intersection restricted by the connection. Namely, it is given by all the traces that are compatible with the subcomponents and their connection.

The connection γ is interpreted with a subset (denoted with $[\![\gamma]\!]$) of traces over $V_S \cup \bigcup_{S' \in Sub(S)} V_{S'}$, i.e. $[\![\gamma]\!] \subseteq Tr(V_S \cup \bigcup_{S' \in Sub(S)} V_{S'})$.

Formally, given a decomposition $\rho = \langle Sub, \gamma \rangle$ of S, a set of implementations $\mathcal{I} = \{I_{S'}\}_{S' \in Sub(S)}$, one for every $S' \in Sub(S)$, the composite implementation of S induced by the decomposition ρ is defined as:

$$CI_S^\rho(\mathcal{I}) := \{\sigma \in Tr(V_S) \mid \text{ there exists } \sigma_{S'} \in I_{S'} \text{ for every } S' \in Sub(S) \text{ such that } \sigma \times \bigotimes_{S' \in Sub(S)} \sigma_{S'} \in [\![\gamma(S)]\!]\}$$

Similarly, the environment of a subcomponent $U \in Sub(S)$ is composed by the environment of S and by the sibling subcomponents of S. Formally, given a decomposition $\rho = \langle Sub, \gamma \rangle$ of S, a set of implementations $\mathcal{I} = \{I_{S'}\}_{S' \in Sub(S) \setminus \{U\}}$, one for every $S' \in Sub(S)$ different from U, and an environment E of S, the composite environment of U induced by the decomposition ρ of S (denoted with $CE_U^\rho(E, \mathcal{I})$) is defined as:

$$CE_U^\rho(E, \mathcal{I}) := \{\sigma_U \in Tr(V_U) \mid \text{ there exists } \sigma_{S'} \in I_{S'} \text{ for every } S' \in Sub(S) \setminus \{U\} \text{ and } \sigma \in E_U \text{ such that } \sigma \times \bigotimes_{S' \in Sub(S)} \sigma_{S'} \in [\![\gamma(S)]\!]\}$$

4.4.2. Asynchronous decomposition

In an asynchronous composition, the traces of the implementations of the subcomponents are concurrent mixing interleaving and synchronizations. In particular, the composite implementation of a component at every discrete step runs a subset of the

subcomponents while the others must stutter. The connections can be used to define which subcomponents must run together on which events. Therefore, we are not forcing a strict interleaving semantics where only one subcomponent active at a time, but we allow more subcomponents to progress together. Note that the supported communication mechanism is given by synchronizations such as hand-shakes or multi-cast as in many formal architectural languages like AltaRica [22], AADL-SLIM [23], and BIP [24]. It would be possible to model buffered communication, but with an explicit buffered component.

Stuttering means that the component is not involved in the transition so that all output data ports do not change and no event of the component happens in that transition. When we consider the asynchronous decomposition of a component S and a trace resulting from such decomposition, if we project the trace over the $V_{S'}$ of a subcomponent $S' \in Sub(S)$, we obtain a sequence where some assignments to the output data ports are repeated (stuttered) without any event of S' happening on these transitions. Removing such stuttering we obtain a trace of S'.

For every component S, we introduce a fictitious event st_S. We denote with V_S^{st} the extended set of ports $V_S \cup \{st_S\}$. The stuttering version of a trace is obtained by inserting in the trace an arbitrary number of additional transitions where st occurs, no other event occurs, and the data ports do not change. If I is an implementation of the component S, we define I^{st} the stuttering version where every trace has been replaced by the corresponding set of stuttering traces.

The connection γ is interpreted with a subset (denoted with $[\![\gamma]\!]$) of traces over $V_S \cup \bigcup_{S' \in Sub(S)} V_{S'}^{st}$, i.e. $[\![\gamma]\!] \subseteq Tr(V_S \cup \bigcup_{S' \in Sub(S)} V_{S'}^{st})$.

Given a decomposition $\rho = \langle Sub, \gamma \rangle$ of S, a set of implementations $\mathcal{I} = \{I_{S'}\}_{S' \in Sub(S)}$, one for every $S' \in Sub(S)$, the asynchronous composite implementation of S induced by the decomposition ρ is defined as:

$$CI_S^\rho(\mathcal{I}) := \{\sigma \in Tr(V_S) \mid \text{ there exists } \sigma_{S'} \in I_{S'}^{st} \text{ for every } S' \in Sub(S) \text{ such that } \sigma \times \bigotimes_{S' \in Sub(S)} \sigma_{S'} \in [\![\gamma(S)]\!]\}$$

Similarly, given a subcomponent U of S, a set of implementations $\mathcal{I} = \{I_{S'}\}_{S' \in Sub(S) \setminus \{U\}}$, one for every $S' \in Sub(S)$ different from U, and an environment E of S, the asynchronous composite environment of U induced by the decomposition ρ of S is defined as:

$$\begin{aligned} CE_U^\rho(E, \mathcal{I}) := \{\sigma'_U \in Tr(V_U) \mid & \text{ there exist } \sigma_U \text{ stuttering of } \sigma'_U, \\ & \sigma_{S'} \in I_{S'}^{st} \text{ for every } S' \in Sub(S) \setminus \{U\} \\ & \text{and } \sigma \in E_U \text{ such that } \sigma \times \bigotimes_{S' \in Sub(S)} \sigma_{S'} \in [\![\gamma(S)]\!]\} \end{aligned}$$

Notice that the γ operator could model different types of synchronization used in various practical modeling languages, such as the strong and weak synchronization mechanisms used in the Altarica language. Consider for example a component S with an input event port p and output event port q and two subcomponents s_1 and s_2 with

the same interface. We refer to their ports as $s_1.p$, $s_1.q$, s_2,p, and $s_2.q$. Consider the connection $\gamma = G((p \leftrightarrow s_1.p) \wedge (q \leftrightarrow s_2.q) \wedge (s_2.p \rightarrow s_1.q))$. In this case, the ports p and q of S are delegated respectively to $s_1.p$ and $s_2.q$. The synchronization between s_1 and s_2 is weak: the input p of s_2 happens only if s_1 generates q, but not the contrary so that $s_1.q$ may be lost, it may triggered when s_2 is stuttering.

4.5. Contract refinement

Contracts are used to specify the assumptions and guarantees of components in a system architecture. We denote with $\xi(S)$ the contracts of the component S. We extend ξ to sets of components such that, if $\mathbb{S}$ is a set of components, $\xi(\mathbb{S}) = \bigcup_{S \in \mathbb{S}} \xi(S)$.

Since the decomposition of a component S into subcomponents induces a composite implementation of S and composite environment for the subcomponents, it is necessary to prove that the decomposition is correct with regards to the contracts. In particular, it is necessary to prove that the composite implementation of S satisfies the guarantee of S's contracts and that the composite environment of each subcomponent U satisfies the assumptions of U's contracts. We perform this verification compositionally only reasoning with the contracts of the subcomponent independently from the specific implementation of the subcomponents or specific environment of the composite component.

Given a component S and a decomposition $\rho = \langle Sub, \gamma \rangle$, a set of contracts $\mathcal{C} \subseteq \bigcup_{S' \in Sub(S)} \xi(S')$ is a refinement of C, written $\mathcal{C} \leq_\rho C$, iff the following conditions hold:

1. if, for all $S' \in Sub(S)$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$, then $CI^\rho_S(\mathcal{I}) \in \mathcal{I}(C)$ (i.e., the correct implementations of the sub-contracts form a correct implementation of C);
2. for every subcomponent U of S, for every contract $C_U \in \xi(U) \cap \mathcal{C}$, if $E \in \mathcal{E}(C)$ and, for all $S' \in Sub(S) \setminus \{U\}$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$, then $CE^\rho_U(E, \mathcal{I}) \in \mathcal{E}(C_U)$ (i.e., for each sub-contract C_U, the correct implementation of the other sub-contracts and a correct environment of C form a correct environment of U).

This is the extension of the definition of [8] of one contract dominating two contracts extended to more components[2]. In the case of one subcomponent C', the definition reduces to that of [8], where the contract C' refines the contract C iff $\mathcal{I}(C') \subseteq \mathcal{I}(C)$ and $\mathcal{E}(C) \subseteq \mathcal{E}(C')$.

Also, differently from [3] and [8], we do not define the composition of contracts since we assume that contracts are given for both the composite component and its subcomponents. This is a pragmatic choice to avoid computing existential quantifications.

5. Proof system for contracts refinement

5.1. Proof system and proof obligations

The purpose of the contracts is to support the compositional verification of a system, but also reuse of component and independent implementation. A verification

[2] We prefer not to use the term "dominance", that is used differently in [3].

method is compositional if it deduces the system properties from the properties of its components without using the internals of the components [25]. The proof of a system property therefore follows the classic structure of a *deduction proof* starting from a set of *axioms* (in this case, the properties of the components) and applying iteratively some *inference rules*. Each rule has as consequence a property of a composite component and as premises the properties of its subcomponents and the definition of the composite component decomposition.

The deduction rules that we use to prove the correctness of a system architecture are the following:

$$\frac{\forall S' \in Sub(S), I_{S'} \models_{imp} C_{S'}, \{C_1, \ldots, C_n\} \leq_\rho C}{CI^\rho_S(I_1, \ldots, I_n) \models_{imp} C} \tag{1}$$

$$\frac{E \models_{env} C, \forall S' \in Sub(S) \setminus U, I_{S'} \models_{imp} C_{S'}, \{C_1, \ldots, C_n\} \leq_\rho C}{CE^\rho_U(E, I_1, \ldots, I_{S' \neq U}, \ldots, I_n) \models_{env} C_U} \tag{2}$$

Rule 1 deduces the correctness of the composite implementation of a component S from the correctness of its subcomponents and the correctness of the contract refinement. Rule 2 deduces the correctness of the composite environment of a subcomponent U of S from the correctness of other subcomponents, the correctness of the environment of S, and the correctness of the contract refinement. These rules are applied along the architectural decomposition and composed to form a proof tree whose leaves correspond to the correctness of the basic components and the contract refinement of each decomposition. The correctness of the contract refinements is discharged generating a set of sufficient and necessary conditions and proving their validity. These *proof obligations* combine the assertions of the contracts involved in the refinement and depend on the connection used in the decomposition and on the type (synchronous vs. asynchronous) decomposition.

In the following sections, we define the proof obligations in the synchronous case (Section 5.2), in the asynchronous case (Section 5.3).

5.2. Checking correct refinement of synchronous decomposition

The following theorem defines the proof obligations of contracts refinement in the case of synchronous decomposition. In the following, since we are considering the decomposition of just one component, in order to simplify the notation, we simply write Sub for $Sub(S)$ and γ for $\gamma(S)$.

Theorem 1. *Consider a component S, a contract C of S, a decomposition $\rho = \langle Sub, \gamma \rangle$, and $\mathcal{C} \subseteq \xi(Sub)$. $\mathcal{C} \leq_\rho C$ iff the following conditions hold:*

$$\exists V_{Sub}(\bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub} (nf(C')) \wedge \gamma) \models (nf(C)) \tag{3}$$

for all $U \in Sub$, for all $\langle A_U, G_U \rangle \in \xi(U) \cap \mathcal{C}$,

$$\exists V_S, V_{Sub \setminus \{U\}}(A \wedge \bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub \setminus \{U\}} (nf(C')) \wedge \gamma) \models (A_U) \tag{4}$$

Basically, the entailment 3 says that the conjunction of the contracts of the subcomponents combined with the connection γ and projected on the variable of the component S, entails the contract of S; for each subcomponent U, the entailment 4 says that the conjunction of the assumption of S with the contracts of the other subcomponents combined with the connection γ and projected on the variable of the component U, entails the assumption of U.

In order to prove the theorem, we use the following lemma that comes directly from the definitions of composite implementation and composite environment.

Lemma 1. *Considering a component S, a decomposition $\rho = \langle Sub, \gamma \rangle$, and a subcomponent $U \in Sub(S)$, the following equivalences hold:*

$$CI^{\rho}_{U}(\mathcal{I}) \equiv \pi_{V_S}(\bigcap_{S' \in Sub \setminus \{U\}} I_{S'} \cap [\![\gamma]\!]) \quad (5)$$

$$CE^{\rho}_{U}(\mathcal{I}) \equiv \pi_{V_U}(E \cap \bigcap_{S' \in Sub \setminus \{U\}} I_{S'} \cap [\![\gamma]\!]) \quad (6)$$

Proof of Theorem 1. We first prove the proof obligation (3). We want to prove that (3) is valid iff $CI^{\rho}_{S}(\{I_{S'}\}_{S' \in Sub}) \models C$ for any set $\{I_{S'}\}_{S' \in Sub}$ of implementations, one for each subcomponents of S, such that for all $S' \in Sub$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$.

$\Leftarrow$ (only if case). We define $I_{S'} := \bigwedge_{C' \in \xi(S') \cap \mathcal{C}} [\![nf(C')]\!]$. Clearly, $I_{S'} \models C'$ for any $C' \in \xi(S') \cap \mathcal{C}$. Then, by hypothesis, $CI^{\rho}_{S}(\{I_{S'}\}_{S' \in Sub}) \models C$. Thus, by Lemma 1, $\pi_{V_S}(\bigcap_{S' \in Sub} I_{S'} \cap [\![\gamma]\!]) \models C$, from which (3) follows.

$\Rightarrow$ (if case). Let us consider a set $\{I_{S'}\}_{S' \in Sub}$ of implementations, one for each subcomponents of S, such that for all $S' \in Sub$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$. Then, $\bigcap_{S' \in Sub} I_{S'} \cap [\![\gamma]\!] \models \bigwedge_{C' \in \xi(S'), S' \in Sub} nf(C') \wedge \gamma$, thus by Lemma 1 $CI^{\rho}_{S}(\mathcal{I}) \models \exists V_{Sub}(\bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub}(nf(C')) \wedge \gamma)$. Then by hypothesis, $CI^{\rho}_{S}(\mathcal{I}) \models \neg A \vee B$.

Second, we prove the proof obligation (4). We want to prove that (4) is valid iff $CE^{\rho}_{U}(E, \{I_{S'}\}_{S' \in Sub}) \models A_U$ for any set $\{I_{S'}\}_{S' \in Sub}$ of implementations, one for each subcomponents of S, such that for all $S' \in Sub$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$, any environment E of S such that $E \models A$, and any contract $C_U \in \xi(U) \cap \mathcal{C}$ such that $C_U = \langle A_U, G_U \rangle$.

$\Leftarrow$ (only if case). We define $I_{S'} := \bigwedge_{C' \in \xi(S') \cap \mathcal{C}} [\![nf(C')]\!]$ and $E := [\![A]\!]$. Clearly, $I_{S'} \models C'$ for any $C' \in \xi(S') \cap \mathcal{C}$ and $E \models A$. Then, by hypothesis, $CE^{\rho}_{S}(E, \{I_{S'}\}_{S' \in Sub}) \models A_U$. Thus, by Lemma 1, $\pi_{V_U}(E \cap \bigcap_{S' \in Sub \setminus \{U\}} I_{S'} \cap [\![\gamma]\!]) \models A_U$, from which (3) follows.

$\Rightarrow$ (if case). Let us consider a set $\{I_{S'}\}_{S' \in Sub}$ of implementations and an environment E such that for all $S' \in Sub$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$ and $E \in \mathcal{E}(C_U)$. Then, $E \cap \bigcap_{S' \in Sub \setminus \{U\}} I_{S'} \cap [\![\gamma]\!] \models A \wedge \bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub \setminus \{U\}}(nf(C')) \wedge \gamma$, thus by Lemma 1 $CE^{\rho}_{U}(E, \mathcal{I}) \models \exists V_{Sub \setminus \{U\}}(A \wedge \bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub \setminus \{U\}}(nf(C')) \wedge \gamma)$. Then by hypothesis, $CE^{\rho}_{U}(E, \mathcal{I}) \models A_U$. □

5.3. Checking correct refinement of asynchronous decomposition

The proof obligations in the case of an asynchronous decomposition follows the same structure of the synchronous case, but the contracts of the subcomponents must

be extended with stuttering. For this reason, we assume to have a transformation st over assertions such that $[\![\phi^{st}]\!] = [\![\phi]\!]^{st}$[3].

Theorem 2. *Consider a component S, a contract C of S, an asynchronous decomposition $\rho = \langle Sub, \gamma\rangle$, and $\mathcal{C} \subseteq \xi(Sub)$. $\mathcal{C} \leq_\rho C$ iff the following conditions hold:*

$$\exists V_{Sub}(\bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub} (nf(C')^{st}) \wedge \gamma) \models (nf(C)) \tag{7}$$

for all $U \in Sub$, for all $\langle A_U, G_U\rangle \in \xi(U) \cap \mathcal{C}$,

$$\exists V_{Sub\setminus\{U\}}(A \wedge \bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub\setminus\{U\}} (nf(C')^{st}) \wedge \gamma) \models (A_U^{st}) \tag{8}$$

Lemma 2. *Considering a component S, an asynchronous decomposition $\rho = \langle Sub, \gamma\rangle$, and a subcomponent $U \in Sub(S)$, the following equivalences hold:*

$$CI_S^\rho(\mathcal{I}) \equiv \pi_{V_S}(\bigcap_{S' \in Sub} I_{S'}^{st} \cap [\![\gamma]\!]) \tag{9}$$

$$CE_U^\rho(\mathcal{I}) \equiv \pi_{V_U}(E \cap \bigcap_{S' \in Sub\setminus\{U\}} I_{S'}^{st} \cap [\![\gamma]\!]) \tag{10}$$

The proof of Theorem 2 follows the same line of the proof of Theorem 1 using Lemma 2 instead of Lemma 1.

6. Instantiation of the framework with temporal logics

6.1. Proof obligations in temporal logics

The framework described in the previous sections can be instantiated with any language to specify set of traces and in particular with any temporal logic with a linear model of time. When the component contracts are specified by means of temporal logic formulas, the validity of the proof obligations defined in Section 5 can be reduced to the satisfiability problem in the same temporal logic. Namely, the proof obligations (3), (4), (7), (8) are valid iff the following formulas are respectively unsatisfiable:

$$(\bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub} (nf(C')) \wedge \gamma) \wedge \neg(nf(C)) \tag{11}$$

$$(A \wedge \bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub\setminus\{U\}} (nf(C')) \wedge \gamma) \wedge (A_U) \tag{12}$$

$$(\bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub} (nf(C')^{st}) \wedge \gamma) \wedge \neg(nf(C)) \tag{13}$$

$$(A \wedge \bigwedge_{C' \in \xi(S') \cap \mathcal{C}, S' \in Sub\setminus\{U\}} (nf(C')^{st}) \wedge \gamma) \wedge \neg(A_U^{st}) \tag{14}$$

[3]Such transformation for the adopted temporal logics can be obtained by a transformation as done in [26].

We can consider different temporal logics depending on the type of traces used to describe the semantics of the components. In particular, Linear-time Temporal Logic (LTL) [12] is suitable for characterizing set of discrete traces, Metric Temporal Logic (MTL) [17] or other real-time extension of LTL for timed traces, and Hybrid LTL with Regular Expressions (HRELTL) [13] for hybrid traces.

If the formulas are restricted to the standard propositional LTL, the validity is decidable and can be performed by standard model checking techniques for propositional temporal logic. For the timed and hybrid cases, the problem is undecidable (although there exist decidable fragments) but most of tools rely on bounded satisfiability [20, 13]. In this case, the problem can be addressed with techniques based on Satisfiability Modulo Theories (SMT), which, although incomplete, are in practice quite effective to prove the satisfiability of the temporal formulas.

6.2. Tool support

The approach presented in previous sections has been implemented in the tool OCRA [11], and was used to analyze the formalization of the WBS example. Interestingly, OCRA allowed us to pinpoint some inaccuracies in the model provided in [7], where some details on the contracts refinement were missing.

OCRA takes in input a textual description of the components, specifying their interfaces and contracts, their decomposition and the refinement of contracts. More specifically, a component specification is composed of an *interface* part, and of an optional *refinement* part. The interface defines the set of *ports* and *parameters* (forming the set V of variables), and the *contracts* (over V). Ports are simple variables and represent what is visible as input/output of the component. Parameters are rigid variables (i.e., their value never changes) and represent some configuration option in the component or in the analysis (e.g., number of tolerated faults). The refinement defines the subcomponents decomposition, the delegation and inter-subcomponents connections, and the contracts decomposition.

The main functionality of OCRA is to generate the proof obligations required to check contract refinements, and to analyze them by exploiting the reasoning capabilities provided by NuSMV3 [27] for the satisfiability/validity checking of temporal logics. In particular, NuSMV3 lifts symbolic reasoning for LTL [28] to the continuous case of HRELTL, by leveraging the functions of MATHSAT [29], the underlying solver for SMT.

OCRA input language uses OTHELLO [16] to specify contracts. The relevant syntax of OTHELLO has been summarized in Table 1 together with the corresponding mathematical formulation in HRELTL. Apart from `der` and `time_until`, the formulas can be interpreted as LTL formulas. Note that in case of the `time_until` operator we use the formulation in State Clock Logic (SCL) [30], since it was not present in the original [13] paper. Thus, `time_until(e)<c` corresponds to the SCL formula $\rhd_{<c} e$, which is equivalent to the MTL formula $(\neg t) U_{<c} t$. We refer the reader to [12, 18, 30, 13] for a precise definition of the semantics. In the following, we give an informal intuition of the semantics of the expressions that are used later in the case study.

Basic formulas are defined with linear arithmetic predicates over the variables or their derivatives. For examples, `x-e<limit` and `der(x)<0` are well-defined formu-

Table 1: The OTHELLO constraints language grammar.

constraint	:=	*atom* |	ϕ	:=	a |
		not *constraint* |			$\neg\phi$ |
		constraint **and** *constraint* |			$\phi \wedge \phi$ |
		constraint **or** *constraint* |			$\phi \vee \phi$ |
		constraint **implies** *constraint* |			$\phi \rightarrow \phi$ |
		always *constraint* |			$G\phi$ |
		never *constraint* |			$G\neg\phi$ |
		in the future *constraint* |			$F\phi$ |
		constraint **until** *constraint*;			$\phi\, U\phi$;
atom	:=	**true** |	a	:=	$\top$ |
		false |			$\bot$ |
		term relation term |			$t \bowtie t$ |
		time_until(*term*) *relation term* |			$\triangleright_{\bowtie t} t$ |
		change(*port* **)** |			$v' \neq v$ |
		term ;			t;
term	:=	*port* |	t	:=	v |
		constant |			c |
		term function term |			$t \star t$ |
		der(*port* **)** |			$\dot{v}$ |
		next(*port* **)** ;			v';

las. Predicates can be combined with Boolean and temporal operators. For example, `x-e<limit and der(x)<0` and `always x-e<limit` are well-defined formulas.

In temporal logic, a formula without temporal operators is interpreted in the initial state. Thus, `x=0` characterizes all traces that start with a state evaluating `x` to 0, and then `x` can evolve arbitrarily. Instead, to express that a predicates holds along the whole evolution, one may use the `always` operator as in `always x=0`.

Another classical example of properties is the response to a certain event. The formula `always (p implies in the future q)` defines the set of traces where every occurrence of `p` is followed by an occurrence of `q`. Note that `q` may happen with a certain delay (although there is no bound on such delay). The formula `always (p implies q)` instead forces `q` to happen at the instant of `p`.

The above formulas do not constrain the time model of the traces. Therefore, they can be interpreted either as discrete traces or as hybrid traces. However, the logic is suitable to characterize specific sets of hybrid traces, constraining when there should be discrete events and how the continuous variables should evolve along continuous evolutions.

The `der(.)` operator is used to specify constraints on the derivative of the continuous evolution of continuous variables. For example, the following OTHELLO constraint:

```
always (train.location<=target implies der(train.location)>=0)
```

characterizes the set of hybrid traces where in all states, if the train has not yet reached the target location, its speed (expressed as the derivative of the location) is greater than or equal to zero.

The `next(.)` operator is used to specify functional properties requiring discrete changes to variables. For example, we can express the property that the warning variable will change value after the train's speed passes the limit with the following con-

```
COMPONENT system

INTERFACE

INPUT PORT Pedal_Pos1: boolean;
INPUT PORT Pedal_Pos2: boolean;
OUTPUT PORT Brake_Line: continuous;
PARAMETER No_Double_Fault: boolean;

CONTRACT brake_time
 assume:
  No_Double_Fault and always Pedal_Pos1=Pedal_Pos2;
 guarantee:
  always ( (change(Pedal_Pos1) or change(Pedal_Pos2))
   implies (time_until( change(Brake_Line) ) <=10) );

REFINEMENT

SUB bscu: BSCU;
SUB hydr: Hydraulic;

DEFINE bscu.Pedal_Pos1 := Pedal_Pos1;
DEFINE bscu.Pedal_Pos2 := Pedal_Pos2;
DEFINE Brake_Line := hydr.Brake_Line;
DEFINE No_Double_Fault := bscu.No_Double_Fault;
DEFINE hydr.CMD_AS := bscu.CMD_AS;
DEFINE hydr.Valid := bscu.Valid;

CONTRACT brake_time REFINEDBY
  bscu.cmd_time, bscu.safety, hydr.brake_time;
```

Figure 4: The WBS `system` component.

straint:

```
always (speed>limit implies in the future next(warning)!=warning)
```

The expression `change(x)` can be used instead of `next(x)!=x`.

In order to constrain the delay between two events, we use the `time_until(.)` operator, which denotes the time that will elapse until the next occurrence of an event. For example, the formula `always (p implies time_until(q)<max_delay)` defines the set of hybrid traces where `p` is always followed by `q` in less than `max_delay` time units.

7. The Wheel Breaking System Case Study

In the following, we illustrate the application of approach on the Wheel Braking System example described in Section 3. We use the concrete syntax of OCRA descriptions. In this example, the components interact synchronously and have hybrid traces as execution models. Therefore, we are considering a continuous model of time and during discrete instantaneous changes all components can do synchronously some actions (without interleaving or specific scheduling).

```
COMPONENT BSCU

INTERFACE

INPUT PORT Pedal_Pos1: boolean;
INPUT PORT Pedal_Pos2: boolean;
OUTPUT PORT CMD_AS: boolean;
OUTPUT PORT Valid: boolean;
PARAMETER No_Double_Fault: boolean;

CONTRACT cmd_time
 assume:
  No_Double_Fault and always Pedal_Pos1=Pedal_Pos2;
 guarantee:
  always ( (Valid and ((change(Pedal_Pos1) or change(Pedal_Pos2))))
   implies (time_until((change(CMD_AS) or fall(Valid))) <=5) );

CONTRACT safety
  assume:
    No_Double_Fault;
  guarantee:
    always Valid;

REFINEMENT

SUB bscu1: subBSCU;
SUB bscu2: subBSCU;
SUB switch: Select_Switch;

DEFINE bscu1.Pedal_Pos := Pedal_Pos1;
DEFINE bscu2.Pedal_Pos := Pedal_Pos2;
DEFINE Valid := bscu1.Valid or bscu2.Valid;
DEFINE switch.In1 := bscu1.CMD_AS;
DEFINE switch.In2 := bscu2.CMD_AS;
DEFINE switch.Select := bscu1.Valid;
DEFINE CMD_AS := switch.Out;
DEFINE No_Double_Fault :=
  always ( (not bscu1.fault_Monitor) and
                (not bscu1.fault_Command) and
                     (not bscu2.fault_Monitor) ) or
  always ( (not bscu1.fault_Monitor) and
                (not bscu1.fault_Command) and
                     (not bscu2.fault_Command) ) or
  always ( (not bscu1.fault_Monitor) and
                (not bscu2.fault_Command) and
                     (not bscu2.fault_Monitor) ) or
  always ( (not bscu1.fault_Command) and
                (not bscu2.fault_Command) and
                     (not bscu2.fault_Monitor) );

CONTRACT cmd_time REFINEDBY
  bscu1.cmd_time, bscu2.cmd_time,
  switch.sel0_time, switch.sel1_time,
  bscu1.safety, bscu2.safety;
CONTRACT safety REFINEDBY
  bscu1.safety, bscu2.safety;
```

Figure 5: The WBS BSCU component.

```
COMPONENT Hydraulic

INTERFACE

INPUT PORT CMD_AS: boolean;
INPUT PORT Valid: boolean;
OUTPUT PORT Brake_Line: continuous;

CONTRACT brake_time
 assume:
  true;
 guarantee:
  always (change(CMD_AS) implies
   (time_until(change(Brake_Line))<=5));
```

Figure 6: The WBS `Hydraulic` component.

Consider Figure 4. The top level `system` component has an interface with two input ports (`Pedal_Pos1` and `Pedal_Pos2`) and an output port (`Brake_Line`). The parameter (`No_Double_Faults`) is used for safety analysis. The contract `brake_time` specifies that, assuming that there is at most one fault and that the value of the input pedal positions is always the same, it is guaranteed that the delay between a change in the input and the execution of the brake shall not exceed 10 time units.

The refinement part shows that `system` is decomposed, as informally shown in Section 3, in the composition of the components `bscu` (of type `BSCU`) and `hydr` (of type `Hydraulic`). The ports of `system` are mapped on the ports of `bscu` and `hydr`. Then, the contract `break_time` is refined by the contract `bscu.cmd_time`, `bscu.safety`, and `hydr.brake_time`.

The interfaces of `BSCU` and `Hydraulic` are shown in Figures 5 and 6, respectively. In turn, `bscu` is refined by the composition of `switch` (of type `Select_Switch`, presented in Figure 7), and of `bscu1` and `bscu2` (of type `subBSCU`, presented in Figure 8).

The correctness of the refinement of the contract `brake_time` of `system` is based on the following arguments. First, the guarantees of `bscu.cmd_time` and `hydr.brake_time` imply the guarantee of the system's contract, provided that `Valid` is always true and this is guaranteed by `bscu.safety`. Second, the assumption of the system contract is the conjunction of the assumptions of `bscu.cmd_time` and `bscu.safety`, while `hydr.brake_time` have a true assumption. Similar but more complex reasoning can be applied to prove the correctness of the refinement of the `BSCU` contracts (shown in Figures 5, 8, and 7). In the case of `bscu.brake_time`, for example, the contract is refined by four contracts of the subcomponents and a mechanized verification is necessary to prove the refinement, because subtle errors may be present as described below. The OCRA tool supports the analysis of the correctness of the above contract refinements. The OCRA engine is able to generate the suitable proof obligations and prove that, for all refinement steps, there is no counterexample up to $k = 10$ in few seconds.

We now contrast the formalization provided above with the original version of [7], in which OCRA was able to pinpoint some inaccuracies (actually, this issue is due to the semantics given by [7] to assumptions so that they do not constrain the environment). In [7] the contract `sel1_time` is formulated as:

```
COMPONENT Select_Switch

INTERFACE

INPUT PORT In1: boolean;
INPUT PORT In2: boolean;
INPUT PORT Select: boolean;
OUTPUT PORT Out: boolean;

CONTRACT sel0_time
assume:
  true;
guarantee:
  (not Select or fall(Select)) releases
     ( change(In1) implies (time_until(change(Out)) <=1) );

CONTRACT sel1_time
assume:
  true;
guarantee:
 always ( (not Select or time_until(fall(Select))<=2)
  implies always (change(In2) implies (time_until(change(Out))<=3)));
```

Figure 7: The WBS `Select_Switch` component.

```
COMPONENT subBSCU

INTERFACE

PORT fault_Monitor: boolean;
PORT fault_Command: boolean;
INPUT PORT Pedal_Pos: boolean;
OUTPUT PORT CMD_AS: boolean;
OUTPUT PORT Valid: boolean;

CONTRACT cmd_time
assume:
  TRUE;
guarantee:
  always ( (Valid and (change(Pedal_Pos))) implies
           (time_until(change(CMD_AS) or fall(Valid)) <=2) );

CONTRACT safety
assume:
 true;
guarantee:
 (always (not fault_Command) and always (not fault_Monitor))
  implies always Valid;
```

Figure 8: The WBS `subBSCU` component.

```
assume:
  (always Select);
guarantee:
  (always (in1 implies time_until(out)<=0.25));
```

Let us denote this contract with C_1, while with C_2 the following contract:

```
assume:
 true;
guarantee:
 (always Select)
  implies (always (in1 implies time_until(out)<=0.25));
```

From the point of view of the implementations, the two versions are equivalent in the sense that $I \in \mathcal{I}(C_1)$ iff $I \in \mathcal{I}(C_2)$. However, they are not equivalent in terms of environments. In fact, while the decomposition of `cmd_time` defined in `BSCU` is correct, it becomes wrong if we substitute C_2 with C_1. The reason is that the assumption `always Select` is too strong and is not covered by the other contracts in the decomposition. Our prototype implementation of the proof system finds (in negligible time) a counterexample to the decomposition, i.e. a trace where `Select` is false (and nothing changes along the evolution).

Another error that we fixed in a first version of the formalization of the WBS architecture decomposition is the connection definition of `Valid` in `BSCU` in terms of `bscu1.Valid` and `bscu2.Valid`. Following the structure of WBS graphically depicted in Figure 2 of [7], we defined `Valid` as the conjunction `bscu1.Valid and bscu2.Valid`. However, this lead to a too strong guarantee in the contract `safety` of `BSCU`. The tool reports in fact a counterexample with a false `bscu2.Valid` and a single fault (`bscu2.fault_Command`) in `bscu2`.

8. Conclusions and Future Work

In this paper we presented a framework for contract-based reasoning with component-based embedded systems. We addressed the problem of verifying the architectural decomposition of an embedded system based on the specification of contracts on the components. The contracts specify the assumptions and guarantees of every components, making clear what the components expect from their environment and what they can guarantee in response. If a component is decomposed into other components, the architecture is correct only if the contracts of the subcomponents imply the contract of the composite component. On the other hand, if a component is part of decomposition (a child of a composite component), the architecture is correct only if the assumptions of the component are satisfied by the contracts of the other connected components and by the environment of the parent component. These conditions can be converted into proof obligations that are valid if and only if the decomposition is correct.

The main contributions of the paper are the following. First, we extended the trace-based framework of [8] with a proper notion of synchronous and asynchronous decomposition, restricting the interaction of subcomponents to connections, and hiding

the variables of subcomponents to the interface of the parent component. Second, we defined a set of proof obligations over the assumptions and guarantees of the contracts involved in such decomposition. Third, we showed that these proof obligations can be reduced to a set of satisfiability problems in the logic used to express the assertions. Fourth, we analyzed the WBS case study presented by [7], using OTHELLO as specification language for hybrid properties [16], and the SMT-based satisfiability procedure presented in [13]. The results showed how we can automatically discover issues in the architecture description of the WBS.

We identify several directions for future research. We are working on improving the reasoning engines connected to OCRA, to effectively prove the validity of the proof obligations in presence of expressive logics such as HRELTL (although the problem is in general undecidable) with algorithms as those presented in [31, 32]. We are also integrating the framework with fault-tree analysis, to exploit the contract-based design flow to analyze the dependencies among component failures. Finally, we intend to integrate the framework with the verification of hybrid programs implementing the contracts.

Acknowledgment

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreements n° 269265 and 295373 and from National funding.

References

[1] A. Cimatti, S. Tonetta, A Property-Based Proof System for Contract-Based Design, in: SEAA, 2012.

[2] B. Meyer, Applying "Design by Contract", Computer 25 (10) (1992) 40–51. doi:10.1109/2.161279.

[3] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, C. Sofronis, Multiple Viewpoint Contract-Based Specification and Design, in: FMCO, 2007, pp. 200–225.

[4] A. Benveniste, B. Caillaud, R. Passerone, A Generic Model of Contracts for Embedded Systems, Tech. rep., INRIA (2007).

[5] S. Quinton, S. Graf, Contract-Based Verification of Hierarchical Systems of Components, in: SEFM, 2008, pp. 377–381.

[6] S. Graf, R. Passerone, S. Quinton, Contract-Based Reasoning for Component Systems with Complex Interactions , in: TIMOBD'11, 2011.

[7] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, I. Stierand, Using contract-based component specifications for virtual integration testing and architecture design, in: DATE, 2011, pp. 1023–1028.

[8] S. Bauer, A. David, R. Hennicker, K. Larsen, A. Legay, U. Nyman, A. Wasowski, Moving from Specifications to Contracts in Component-Based Design, in: FASE, 2012, pp. 43–58.

[9] D. Cofer, A. Gacek, S. Miller, M. Whalen, B. LaValley, L. Sha, Compositional Verification of Architectural Models, in: NFM, 2012, pp. 126–140.

[10] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, K. Larsen, Contracts for System Design, Tech. Rep. RR-8147, INRIA (Nov. 2012).

[11] A. Cimatti, M. Dorigatti, S. Tonetta, OCRA: A Tool for Checking the Refinement of Temporal Contracts, in: ASE, IEEE, 2013, pp. 702–705.

[12] A. Pnueli, The temporal logic of programs, in: FOCS, 1977, pp. 46–57.

[13] A. Cimatti, M. Roveri, S. Tonetta, Requirements Validation for Hybrid Systems, in: CAV, 2009, pp. 188–203.

[14] A. Sangiovanni-Vincentelli, W. Damm, R. Passerone, Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems, Eur. J. Control 18 (3) (2012) 217–238.

[15] L. de Alfaro, T. A. Henzinger, Interface Theories for Component-Based Design, in: EMSOFT, 2001, pp. 148–165.

[16] A. Cimatti, M. Roveri, A. Susi, S. Tonetta, Validation of requirements for hybrid systems: A formal approach, ACM TOSEM 21 (4) (2012) 22.

[17] R. Koymans, Specifying Real-Time Properties with Metric Temporal Logic, Real-Time Systems 2 (4).

[18] R. Alur, T. Henzinger, Real-time Logics: Complexity and Expressiveness, in: LICS, 1990, pp. 390–401.

[19] P. Zhang, B. Li, L. Grunske, Timed property sequence chart, J. Syst. Softw. 83 (3) (2010) 371–390.

[20] M. Pradella, A. Morzenti, P. S. Pietro, Bounded satisfiability checking of metric temporal logic specifications, ACM TOSEM 22 (3) (2013) 20.

[21] SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (Dec. 1996).

[22] A. Arnold, A. Griffault, G. Point, A. Rauzy, The AltaRica formalism for describing concurrent systems, Fundamenta Informaticae 40 (2000) 109–124.

[23] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, M. Roveri, Safety, dependability, and performance analysis of extended AADL models, The Computer Journal 54 (5) (2011) 754–775.

[24] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, J. Sifakis, Rigorous Component-Based System Design Using the BIP Framework, IEEE Software 28 (3) (2011) 41–48.

[25] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers, Concurrency verification: introduction to compositional and noncompositional methods, Cambridge University Press, 2001.

[26] N. Benes, L. Brim, I. Cerná, J. Sochor, P. Vareková, B. Zimmerova, Partial Order Reduction for State/Event LTL, in: IFM, 2009, pp. 307–321.

[27] The NuSMV3 system, fondazione Bruno Kessler, Trento, Italy, `https://es.fbk.eu/tools/nusmv3/`.

[28] A. Cimatti, M. Roveri, V. Schuppan, S. Tonetta, Boolean Abstraction for Temporal Logic Satisfiability, in: CAV, 2007, pp. 532–546.

[29] A. Cimatti, A. Griggio, B. J. Schaafsma, R. Sebastiani, The MathSAT5 SMT Solver, in: TACAS, 2013, pp. 93–107.

[30] J.-F. Raskin, P.-Y. Schobbens, State Clock Logic: A Decidable Real-Time Logic, in: HART, 1997, pp. 33–47.

[31] A. Cimatti, A. Griggio, S. Mover, S. Tonetta, IC3 Modulo Theories via Implicit Predicate Abstraction, in: TACAS, 2014, pp. 46–61.

[32] A. Cimatti, A. Griggio, S. Mover, S. Tonetta, Verifying LTL Properties of Hybrid Systems with K-Liveness, in: CAV, 2014, to appear.