A Property-Based Proof System for Contract-Based Design

Alessandro Cimatti
Fondazione Bruno Kessler
Email: cimatti@fbk.eu

Stefano Tonetta
Fondazione Bruno Kessler
Email: tonettas@fbk.eu

***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 behaviour. Contracts specify the input-output behaviour of a component by defining what the component *guarantees*, provided that the its environment obeys some given *assumptions*. 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 a novel, fully formal contract framework. The decomposition of the system architecture is complemented with the corresponding decomposition of component contracts. The framework exploits such decomposition to automatically generate a set of proof obligations, which, once verified, allow concluding the correctness of the top-level system properties. The framework relies on an expressive property specification language, conceived for the formalization of embedded system requirements. The proof system reduces the correctness of contracts refinement to entailment of temporal logic formulas, and is supported by a verification engine based on automated SMT techniques.

I. Introduction

The development of safety-critical complex embedded systems is witnessing a paradigm shift in many industrial sectors: Original Equipment Manufacturer (OEM) reorganized the supply chain focusing on those parts of the design at the core of their competitive advantage, and share the design and development of other parts with the competitors. This poses strong challenges in terms of global verification and certification, and has lead to a growing interest in interface theories and frameworks of embedded systems [SV07], [BCF$^+$07]. Among them, contract-based design [BCF$^+$07], [BCP07], [QG08], [GPQ11], [DHJ$^+$11], [BDH$^+$12] 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 behavior. Contracts specify the input-output behavior of a component by defining what the component *guarantees*, provided that the its environment obeys some given *assumptions*. Unlike pre- and post-conditions of sequential programs, in the context of embedded systems, assumptions and guarantees are properties of the whole history/dynamics of a component.

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. 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 qualified components.

In this paper, we present a novel, fully formal contract framework, able to complement the decomposition of the system architecture with a corresponding decomposition of component contracts. We make two key contributions.

First, we define a *proof system*, that exploits contract decomposition to automatically generate a set of proof obligations: the correctness of the system properties decomposition directly follows from the discharge (i.e., verification) of such proof obligations. We associate to each component a set of contracts, each of which can be refined by a set of contracts. The framework is presented at a general and abstract level, in terms of traces, and encompassess many different concrete ways to specify contracts (e.g., automata, logics, patterns). We remark that the definition of the refinement poses several important technical challenges. The contract separation of assumptions and guarantees is moreover important to define properly the refinement of components. In fact, even if the specification language allows to capture the assume-guarantee with an implication so that $I \models \langle A, G\rangle$ iff $I \models A \rightarrow G$, this is not sufficient when considering refinement. As noted in [AHKV98], the implementation can satisfy a smaller (refined) specification while restricting the assumption of A. Since the assumptions guaranteed by other components cannot be controlled by the component under refinement, such restriction of assumptions must be avoided. This is the reason of the necessity of a covariant/contravariant refinement of contracts [BDH$^+$12], where guarantees are strengthened while assumptions are weakened.

The second contribution is to instantiate the framework so that contracts are expressed in terms of an expressive *property specification language* called Othello [CRST]. The underlying temporal logic, HRELTL [CRT09], has been conceived for the formalization of requirements for embedded systems and can express temporal constraints on both discrete events and real-time continuous quantities. The proof system is also specialized, so that the proof obligations that yield the correctness of contracts refinement is mapped onto entailment of temporal logic formulae.

The approach has been implemented on top of the NuSMV3 framework. The framework is based on a reasoning engine for satisfiability/validity checking on HRELTL, which lifts

symbolic reasoning for discrete temporal logics (SAT-based LTL) to the continuous case by leveraging on SMT techniques [BCF+08].

Compared to related work, our work has several key elements of novelty. First, in previous approaches the refinement checks are not integrated into a system architecture description. Our work paves the way to a tighter integration of contract based design in the flow of safety-critical applications, because the automated production of refinement checks. Second, the current approaches are typically either theoretic or limited to specific specification patterns, rather than a general temporal logic. Our specialization supports more expressive and general properties. Finally, ours is the first mechanized system for property-based contract refinement.

This paper is structured as follows. In Section II, we present a motivating case study. In Section III, we present our contract-based framework in terms of generic sets of generic traces. In Section IV, we discuss the property-based specialization of the framework, considering the particular type of hybrid traces and characterizing sets of traces with logical properties. In Section V, we discuss relevant related work. In Section VI, we draw some conclusions and outline directions for future work.

II. Motivating example

In this section, we describe informally the benchmark that is used in examples throughout the paper. It is taken from [DHJ+11], where the architecture of the system and the formalized contracts of the components are presented.

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 system takes in input two brake `Pedal_Pos` signals 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 no more than 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 componens: two redundant `sub-BSCU`, and a `Select_Switch` that selects the back-up `sub-BSCU` when the signal of the primary `sub-BSCU` is no more valid due to a failure. The two `sub-BSCU` 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.

III. Contract-based Compositional Verification of System Architecture

A. Trace-based Contracts

As in [BCF+07], we base our models on a set V of variables, which represent the relevant information of the system (e.g., the ports between components) and sets of

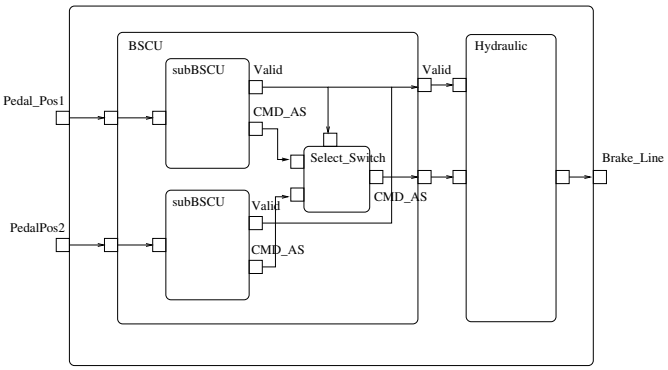

Fig. 1. High-level view of the Wheel Braking System.

traces/behaviors over V, which are (discrete or continuous) evolutions of the values assigned to the variables in V.

A *system architecture* is defined by means of *components*. In order to simplify the description of the proof system, we omit the details on input/output ports and their connection, and we consider a global set V of variables. In practice, components have their own set of variables and mappings are used to delegate ports to sub-components and to connect ports of sub-components.

A component may include both *implementations* and *contracts*. An implementation is an instantiation of a component and is here modeled as a set of traces. A contract is a pair $\langle A, G\rangle$ of assertions, which represents respectively an *assumption* and a *guarantee* for the component. An assertion is a property that may or may not be satisfied by a trace and is also modeled here as a set of traces.

As in [BCF+07], [BDH+12], contract semantics are defined in terms of implementations and environments, which consist of sets of traces. Let a contract $C = \langle A, G\rangle$ over V be given. Let I and E be two sets of traces over V. We say that I is an *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 $\mathcal{E}(C) = \mathcal{E}(C')$.

As defined in [BCF+07], a contract $C = \langle A, G\rangle$ is in normal form iff $G = \overline{A} \cup G$ (where $\overline{A}$ denotes the complement of A). We denote with $G^{nf(A)}$ the assertion $\overline{A} \cup G$.

The contract $\langle A, G^{nf(A)}\rangle$ is in normal form, and is equivalent (i.e., has the same implementations and environments) to $\langle A, G\rangle$.

Theorem 1 ([BCF+07]): $\langle A, G\rangle \equiv \langle A, G^{nf(A)}\rangle$.

We say that a contract C' refines a contract C ($C' \leq C$) iff $\mathcal{I}(C') \subseteq \mathcal{I}(C)$ and $\mathcal{E}(C) \subseteq \mathcal{E}(C')$. The following theorem says that a sufficient and necessary condition for the refinement relation is that the assumption is weakened and the guarantee is strengthened (relatively to the assumption).

Theorem 2 ([BDH+12]): $\langle A', G'\rangle \leq \langle A, G\rangle$ iff $A \subseteq A'$ and $G' \cap A \subseteq G$.

B. Compositional verification 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 method is compositional if it deduces the system properties from the properties of its components without using the internals of the components [dRdBH$^+$01]. 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 semantics of the proof is that, for all implementations of the basic components that satisfy the axiom properties, the system built with such component will satisfy the deduced property. In order to be correct, the proof generates a *proof obligation* for all inference rule used in the deduction. A proof obligation is a condition, typically written in a logical formula, which is sufficient to prove the soundness of the inference rule. If the proof obligation is valid only if the inference rule is correct, we say that the proof obligation is complete.

C. System architecture and components decomposition

A *system architecture* is defined by means of *components* interface and structural decomposition of components in subcomponents. Formally, we define the *interface* of a component S as a set $\xi(S)$ of contracts over V, and we define the *decomposition* of a component S as a set $\rho(S)$ of subcomponents. A *system architecture* consists of a top-level component, also called *system component*, and the recursive definition of the interface and decomposition of the system components and sub-components.

Implementations and environments of S are sets of traces over V. If S is decomposed in sub-components, an implementation of S is typically obtained by composing the implementations of the sub-components. In our settings, since we are considering a synchronous model and a global set of variables, the composition is obtained simply with the intersection of the traces. Formally, if $\rho(S) \neq \emptyset$, an implementation of S is given by $\bigcap_{S' \in \rho(S)} I_{S'}$, where $I_{S'}$ is an implementation of S'. We say that such implementation of S is induced by the implementations $\{I_{S'}\}_{S' \in \rho(S)}$.

Similarly, an environment of $S'' \in \rho(S)$ is given by $E \cap \bigcap_{S' \in \rho(S) \setminus \{S''\}} I_{S'}$, where E is an environment of S and $I_{S'}$ is an implementation of S'. We say that such environment of S'' is induced by the environment E and the implementations $\{I_{S'}\}_{S' \in \rho(S) \setminus \{S''\}}$.

We say that I is a *correct implementation* of S iff $I \in \mathcal{I}(C)$ for every contract $C \in \mathcal{C}$. We say that E is a *correct environment* of S iff $I \in \mathcal{E}(C)$ for every contract $C \in \mathcal{C}$.

D. Proof obligations for contracts decomposition

Let us consider a component S which is refined in the sub-components $\rho(S) = \{S_1, \ldots, S_n\}$. We say that a set of contracts $\mathcal{C} \subseteq \bigcup_{S' \in \rho(S)} \xi(S')$ ($\mathcal{C} \leq C$) is a correct decomposition of C iff the following conditions hold:

1) for any implementation I of S induced by the implementations $\{I_{S'}\}_{S' \in \rho(S)}$ of the sub-components of S, if, for all $S' \in \rho(S)$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$, then $I \in \mathcal{I}(C)$ (i.e., the correct implementations of the sub-contracts form a correct implementation of C);
2) for every subcomponent S'' of S, for every contract $C'' \in \xi(S'') \cap \mathcal{C}$, for any environment E'' of S'' induced by the environment E of S and the implementations $\{I_{S'}\}_{S' \in \rho(S) \setminus \{S''\}}$ of the sub-components of S, if $E \in \mathcal{E}(C)$ and, for all $S' \in \rho(S) \setminus \{S''\}$, for all $C' \in \xi(S') \cap \mathcal{C}$, $I_{S'} \in \mathcal{I}(C')$, then $E'' \in \mathcal{E}(C'')$ (i.e., for each sub-contract C'', the correct implementation of the other sub-contracts and a correct environment of C form a correct environment of C'').

This is the extension of the definition of [BDH$^+$12] of one contract dominating two contracts extended to more components. We prefer not to use the term "dominance" that is used differently in [BCF$^+$07]. Also, differently from [BDH$^+$12], we take the viewpoint of a top-down design.

Note that the refinement coincides with the notion of decomposition when the set is a singleton.

The following theorem gives a sufficient and necessary condition for the decomposition relation as done by Theorem 2 for the refinement relation.

Theorem 3: Consider $\mathcal{C} = \{\langle A_1, G_1\rangle, \ldots, \langle A_n, G_n\rangle\}$. $\mathcal{C} \leq C$ iff ${G_1}^{nf(A_1)} \wedge \ldots \wedge {G_n}^{nf(A_n)} \subseteq G^{nf(A)}$ and for all i, $1 \leq i \leq n$, $A \cap \bigcap_{1 \leq j \leq n, j \neq i} {G_j}^{nf(A_j)} \subseteq A_i$.

E. Deduction proofs of system contracts

The architecture abstract description defined in Section III-C is enriched with contracts for each component composing the system. The correctness of the contracts of the system component is proved by composing the correctness arguments of the sub-components.

To this purpose, we define for each component S a refinement of each of its contracts C, defined as the set $\chi(C)$ of contracts of the sub-components of S (i.e., $\chi(C) \subseteq \bigcup_{S' \in \rho(S)} \xi(S')$). For each contract of the system component, the relation χ defines a proof tree as the one depicted in Figure 9 for the WBS contract `brake_time`. Note that the decomposition of a contract may contain more than one contract of the same sub-component.

The correctness of the system component contracts is therefore proved by checking that for every contract C in the proof tree, the defined decomposition $\chi(C)$ is a correct decomposition of C, i.e., $\chi(C) \leq C$.

IV. Property-based proof system

A. Hybrid traces

The system specified by the contracts we are considering is an embedded system interacting with the physical environment. The underlying model is given by hybrid systems [ACHH92], which combines discrete aspects representing the control part (including for example the system modes) and

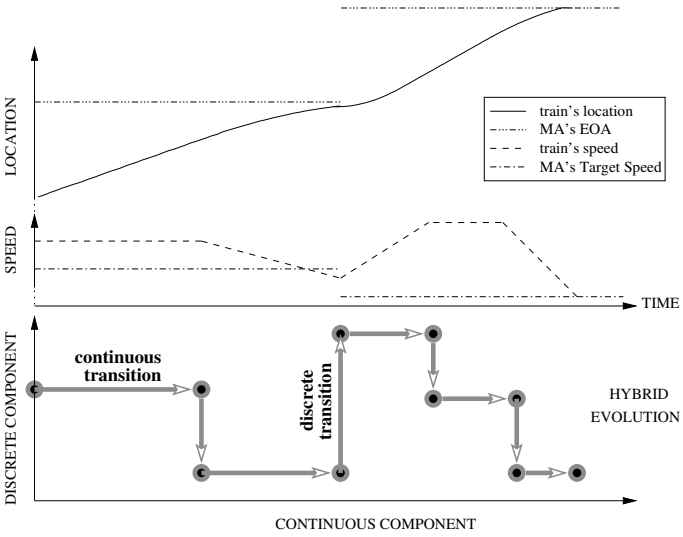

Fig. 2. Example of hybrid evolution. The location and the speed of the train evolve continuously along time. The EOA and the Target Speed of the MA instead change instantaneously.

continuous aspects representing the physical part (including the time and the monitored and controlled variables). An evolution of the system is represented by a sequence of discrete and continuous transitions. Discrete transitions are characterized by *instantaneous* changes of the systems involving control switches and events such as system-mode transitions, and also changes to the continuous variables such as the reset of a timer.

Continuous transitions are characterized by the *elapsing of time*, which is associated with the evolution of the continuous aspects according to its dynamics constraints, while the discrete aspects remains unchanged.

An example of such evolution is depicted in Figure 2: the location and the speed of the train are continuous variables and thus evolve with a continuous function, while the End Of Authority (EOA) and the Target Speed (TS) of the Movement Authority (MA) are discrete variables that change instantaneously, and thus evolve with a step function. In the figure, the train starts from an initial location running with constant speed and alternating continuous and discrete transitions (see lower part of the figure) it moves to a different location; it passes to deceleration mode; then, it decelerates while approaching the first EOA until the speed is lower than the corresponding target speed; then, the train receives a new MA with a farther EOA, and passes to acceleration mode; it moves on, passes to constant speed, moves on, passes to deceleration mode, and finally reaches the second EOA.

B. A temporal logic for embedded systems

We characterize sets of hybrid traces by means of a symbolic logic, called OTHELLO, which allows for complex combinations of linear temporal operators, Boolean connectives, regular expressions, over terms referring to the variables of components.

In the following, we give some examples of OTHELLO constraints, while we refer the reader to [CRST] for a complete definition of the language. A formal definition of the set of hybrid traces represented by these constraints can be found instead in [CRT09].

The following OTHELLO constraint:

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

states that if the train has not yet reached the EOA, its speed is greater than or equal to zero.

Similarly, the following OTHELLO constraint:

```
always (train.location=MA.EOA implies
                              der(train.location)<=MA.TS)
```

states that the speeds of the train must be less than the target speed when passing over the MA.EOA.

The logic is suitable to express functional, timing, and safety-related properties as some of the patterns described in [CES10]. The **der** operator is used to specify constraints on the derivative of the continuous evolution of continuous variables. The **next** operator is used to specify functional properties requiring discrete changes to variables. For example, we can express the property that the train will receive a new MA after passing the EOA with the following constraint:

```
always (train.location=MA.EOA implies in the future change(MA))
```

where **change**(x) is an abbreviation for **next**(x)!=x which means that there is a discrete event in which MA is changed.

Timers are continuous variables with derivative constantly equal to one and are used to express timing constraints. For example, we use the following declaration:

```
DELAY d: delay between (trigger , reaction);
```

as an abbreviation for declaring a continuous variable d with constraint:

```
(always (trigger implies (reaction releases
                          (der(d)=1 and not change(d)))))
and (always (reaction implies next (trigger releases d=0)))
and (trigger releases d=0)
```

which means that d tracks the maximum delay between the condition trigger and reaction. More specifically, initially and after a reaction, it remains zero until the first trigger; after the first trigger (since the beginning or since the last reaction) it behaves as a timer until a new reaction. Figure 3 gives a pictorial view of the semantics.

Finally, as in [DHJ+11], faults are modeled as standard discrete events and are used to specify safety-related properties such as fault-tolerance.

C. Othello system and components specification

In this section, we define the concrete language that we propose to describe the architecture decomposition of a system components and the decomposition of system contracts into the contracts of sub-components.

An Othello System Specification (OSS) is written following the grammar below (in Extended Backus-Naur Form):

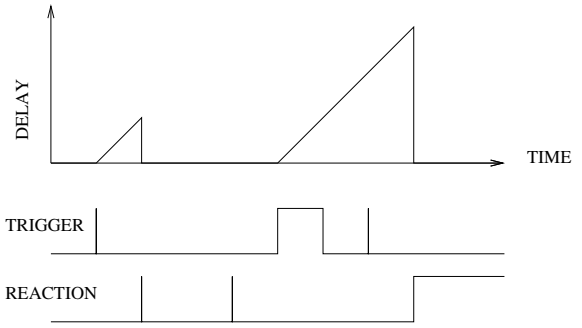

Fig. 3. An example of the semantics of a delay variable. The real value of the delay is function of the Boolean waveforms of the trigger and reaction expressions shown at the bottom.

```
OCS         = system_comp  component* ;
system_comp = "COMPONENT" "system" interface refinement? ;
component   = "COMPONENT" name interface refinement? ;
interface   = "INTERFACE" var* contract* ;
refinement  = "REFINEMENT" sub* connection* refinedby* ;
var         = (port | parameter | delayv) ;
port        = ("INPUT" | "OUTPUT") "PORT" name ":" type ";" ;
parameter   = "PARAMETER" name ":" type ";" ;
delayv      = "DELAY" name ":" "delay between"
                    "(" constraint "," contraint ")" ";" ;
type        = "boolean" | "integer" | "real" | "continuous" ;
contract    = "CONTRACT" name "assume" ":" constraint ";"
                         "guarantee" ":" constraint ";" ;
sub         = "SUB" name ":" name ";" ;
connection  = "DEFINE" name ":=" constraint ";" ;
refinedby   = "CONTRACT" name "REFINEDBY" contr_id+ ";" ;
contr_id    = name "." name ;
```

where `name` is replaced with a string and `constraint` is replaced with an Othello constraint. Note that if we do not need continuous variables and timing constraints, we can consider discrete-time LTL formulas, for which satisfiability is decidable.

An Othello System Specification (OSS) is tuple $\langle \mathcal{S}, S \rangle$ where $\mathcal{S}$ is a set of Othello Component Specification (OCS) and $S \in \mathcal{S}$ is the system top-level component. An OCS consists of an interface and possibly a refinement. 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 difference with the abstract architecture of Section III is that a property specification language is used to specify the assumption and guarantees of contracts. The trace sets used in Section III are given by the semantics of the formulas. The variables used in the contracts are local to the owning component but the connection definitions are used to map the variables used in a refinement step to a global set of variables.

D. SMT-based verification of property-based proof obligations

When the component contracts are specified by means of logical formulas, the proof obligations defined in Section III can be specified with formulas too. In particular, if a contract $C = \langle \alpha, \beta \rangle$ is decomposed in the contracts $\langle \alpha_1, \beta_1 \rangle, \ldots, \langle \alpha_n, \beta_n \rangle$, the correctness of the decomposition can be verified by checking the validity of the following formulas:

$$((\neg\alpha_1 \vee \beta_1) \wedge \ldots \wedge (\neg\alpha_n \vee \beta_n)) \rightarrow (\neg\alpha \vee \beta)$$

$$(\alpha \wedge \bigwedge_{2 \leq j \leq n, j} (\neg\alpha_j \vee \beta_j)) \rightarrow \alpha_1$$

$$\ldots$$

$$(\alpha \wedge \bigwedge_{1 \leq j \leq n, j \neq i} (\neg\alpha_j \vee \beta_j)) \rightarrow \alpha_i$$

$$\ldots$$

$$(\alpha \wedge \bigwedge_{1 \leq j \leq n-1} (\neg\alpha_j \vee \beta_j)) \rightarrow \alpha_n$$

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. In the most general case, the problem is undecidable but can be addressed with SMT-based model checking techniques, which, although incomplete, are in practice quite effective. In particular, we use the approach described in [CRT09], which can decide if there exists a lasso-shape trace with up to k states violating the proof obligations.

We implemented a prototype that takes in input a OSS and generates the proof obligations on top of NuSMV3, which interfaces with NuSMV2 [CCG+02] model checking and with the MathSAT [BCF+08] SMT solver, and supports the analysis of OTHELLO constraints.

E. Case study

We consider the Wheel Braking System (WBS) described in [DHJ+11] and we show how the proposed approach is able to increase the rigor of the reasoning at support of the contract refinement.

The system component has two input ports (`Pedal_Pos1` and `Pedal_Pos2`) and an output port (`Brake_Line`). We added a parameter (`No_Double_Faults`) used for safety analysis. The contract `brake_time` of the system component specifies that, assuming that there are no more than 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 OCS of this system component is shown in Figure 4. The system is decomposed in the `BSCU` component `bscu` and in the `Hydraulic` component `hydr` as described in Section II.

The decomposition of the system component requires the definition of the `BSCU` and `Hydraulic` components interface, shown respectively in Figures 5 and 6. These contain the definition of the contracts `bscu.cmd_time`, `bscu.safety`, and `hydr.brake_time`, which refine

```
COMPONENT system

INTERFACE

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

DELAY Delay_between_pedal_and_brake: delay between
((change(Pedal_Pos1) or change(Pedal_Pos2)) ,
 change(Brake_Line));

CONTRACT brake_time
assume:
  No_Double_Fault and always Pedal_Pos1=Pedal_Pos2;
guarantee:
  always ( Delay_between_pedal_and_brake<=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;
```

Fig. 4. OCS of the WBS system component.

the system component contract. The refinement is correct because, first, the guarantees of `bscu.cmd_time` and `hydr.brake_time` implies 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, 7, and 8). The complete deduction tree is shown in Figure 9. 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. Our prototype implementation proved that, for all refinement steps, there is no counterexample up to $k = 10$ in few seconds.

a) Example of too strong assumption: In the original version of [DHJ+11], the contract `sel1_time` is formulated as:

```
assume:
  (always Select);
guarantee:
  (always (Delay_between_in1_and_out<=0.25));
```

Let us denote this contract with C_1, while `sel1_time` with C_2. 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`

```
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;

DELAY Delay_between_pedal_and_cmd: delay between
((change(Pedal_Pos1) or change(Pedal_Pos2)) ,
 (change(CMD_AS) or fall(Valid)));

CONTRACT cmd_time
assume:
  always Pedal_Pos1=Pedal_Pos2;
guarantee:
  always ( Delay_between_pedal_and_cmd<=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;
CONTRACT safety REFINEDBY bscu1.safety, bscu2.safety;
```

Fig. 5. OCS of the WBS BSCU component.

```
COMPONENT Hydraulic

INTERFACE

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

DELAY Delay_between_cmd_and_brake: delay between
(change(CMD_AS) , change(Brake_Line));

CONTRACT brake_time
assume:
  TRUE;
guarantee:
  always ( Delay_between_cmd_and_brake<=5 );
```

Fig. 6. OCS of the WBS Hydraulic component.

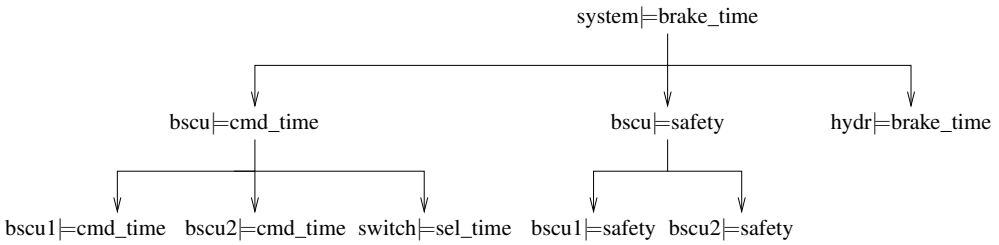

Fig. 9. Deduction tree of system property brake_time.

```
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;

DELAY Delay_between_pedal_and_cmd: delay between
((change(Pedal_Pos)) , change(CMD_AS));

CONTRACT cmd_time
assume:
  TRUE;
guarantee:
  (always Valid) implies
    (always ( Delay_between_pedal_and_cmd<=4 ));

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

Fig. 7. OCS of the WBS subBSCU component.

```
COMPONENT Select_Switch

INTERFACE

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

DELAY Delay_between_in1_and_out:
  delay between (change(In1) , change(Out));
DELAY Delay_between_in2_and_out:
  delay between (change(In2) , change(Out));

CONTRACT sel0_time
assume:
  TRUE;
guarantee:
  always ( fall(Select) implies
    ( always Delay_between_in2_and_out<=1 ) );

CONTRACT sel1_time
assume:
  TRUE;
guarantee:
  (always Select) implies
    ( Delay_between_in1_and_out<=0.25 );
```

Fig. 8. OCS of the WBS Select_Switch component.

defined in BSCU OCS 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 which is trace where `Select` is false (and nothing changes along the evolution).

b) Example of too strong guarantee: 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 a figure of [DHJ+11], we defined `Valid` as the conjunction `bscu1.Valid and bscu2.Valid`. However, this leads 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`.

V. Related work

There are many works such as [BCF+07], [BCP07], [QG08], [GPQ11], [DHJ+11], [BDH+12], which present different approaches to contract-based design. Most of these works (see, e.g., [BCF+07], [BCP07], [QG08], [GPQ11]) are very theoretical and abstract without concrete languages and examples of complete decomposition from system properties. This paper advances the state-of-the-art by concretely showing how a system architecture can be verified by means of contracts. We provide a concrete language and tool to support such verification, and we associate the underlying reasoning to a rigorous formal contract framework.

The paper builds in particular on two recent works, [BDH+12] and [DHJ+11], the first as the basis for the formal contract-based reasoning, the second for a concrete example of contracts decomposition. In particular, we build on [BDH+12], which described how a specification language (such as Othello) can be used in a contract framework. Besides instantiating this method for the Othello specification language, we also provide new theoretical results tailored to the verification of contracts decomposition. In fact, while [BDH+12] 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 refined by a specific decomposition (in a top-down design process).

As in [BCF+07], we use a trace-based semantics for contracts. However, we use a concrete property-based language to express the assumption and guarantees of the contracts. Also, differently from [BCF+07], 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.

As in [DHJ+11], we combine contracts with functional, timing, and safety aspects. However, in [DHJ+11], the semantics of a contract is given just in terms of implementations and the refinement is defined as trace-set inclusion, therefore missing the covariant/contravariant relation, which is necessary for a proper refinement of behavioral typing. In fact, trace-set inclusion allows to refine a contract by strengthening the assumptions. Another important difference is that the language is based on CSL and thus on hybrid automata while our approach is based on hybrid temporal logics for which we have an established tool support.

Finally, we build our framework on existing advanced techniques for property specifications and analysis [CRT09], [CRST]. In this works, HRELTL and Othello are used to formalize requirements without distinction between assumptions and guarantees. The properties are checked for entailment, which, as discussed, is a pre-order relation not suitable for contracts. Intuitively, an implementation satisfies a contract iff its behaviors, restricted to the assumption, satisfy the guarantee. As we have shown in the case study, the distinction between assumption and guarantee has a strong impact in terms of semantics of contracts and their refinement. In this sense, contracts are more expressive and more adequate than simple unstructured property specification languages.

VI. Conclusions and Future Work

We addressed the problem of verifying the decomposition of hybrid properties in a component-based specification. The properties are specified in a contract-based framework, with an explicit distinction between assumption and guarantees. Their decomposition is correct provided that guarantees are strengthened and assumptions are relaxed while proceeding along the architectural decomposition. We provided a proof system that generates a set of proof obligations that valid if and only if the decomposition is correct.

The main contributions of the paper are the following. First, we presented the proof obligations in the generic trace-based contract framework of [BCF+07] and [BDH+12] and proved their correctness. Second, we instantiated the generic framework by adopting the Othello language as specification language for hybrid properties [CRST] and the SMT-based satisfiability procedure presented in [CRT09]. We extended the tool support of NuSMV3 to take in input a component-based specification with assumptions and guarantees written in Othello and to automatically generate the proof obligations. We validated the tool with the case study presented by [DHJ+11].

Directions for future research include to extend the framework to asynchronous systems as HyDI [CMT11] and to improve the support for validity checking based on abstraction techniques as in [Ton09] and syntactic simplifications as in [CRT07].

Acknowledgment

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

References

[ACHH92] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[AHKV98] R. Alur, T. A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating Refinement Relations. In *CONCUR*, pages 163–178, 1998.

[BCF+07] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In *FMCO*, pages 200–225, 2007.

[BCF+08] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*, pages 299–303, 2008.

[BCP07] A. Benveniste, B. Caillaud, and R. Passerone. A Generic Model of Contracts for Embedded Systems. Technical report, INRIA, 2007.

[BDH+12] S.S. Bauer, A. David, R. Hennicker, K.G. Larsen, A. Legay, U. Nyman, and A. Wasowski. Moving from Specifications to Contracts in Component-Based Design. In *FASE*, pages 43–58, 2012.

[CCG+02] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, pages 359–364, 2002.

[CES10] CESAR Deliverable D SP2 R2.2 M2: Definition and exemplification of RSL and RMM, 2010.

[CMT11] A. Cimatti, S. Mover, and S. Tonetta. HyDI: A Language for Symbolic Hybrid Systems with Discrete Interaction. In *EUROMICRO-SEAA*, pages 275–278, 2011.

[CRST] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Validation of Requirements for Hybrid Systems: a Formal Approach. *ACM TOSEM*, 21(4). To appear.

[CRT07] A. Cimatti, M. Roveri, and S. Tonetta. Syntactic Optimizations for PSL Verification. In *TACAS*, pages 505–518, 2007.

[CRT09] A. Cimatti, M. Roveri, and S. Tonetta. Requirements Validation for Hybrid Systems. In *CAV*, pages 188–203, 2009.

[DHJ+11] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *DATE*, pages 1023–1028, 2011.

[dRdBH+01] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods*. Cambridge University Press, 2001.

[GPQ11] S. Graf, R. Passerone, and S. Quinton. Contract-Based Reasoning for Component Systems with Complex Interactions . In *TIMOBD'11*, 2011.

[QG08] S. Quinton and S. Graf. Contract-Based Verification of Hierarchical Systems of Components. In *SEFM*, pages 377–381, 2008.

[SV07] A. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, 2007.

[Ton09] S. Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM*, pages 89–105, 2009.