BCL: a compositional contract language for embedded systems

Orlando Ferrante*, Roberto Passerone*†, Alberto Ferrari*, Leonardo Mangeruca*, Christos Sofronis*
*Advanced Laboratory on Embedded Systems S.r.l., Via Barberini 50, Rome, Italy – e-mail: first.last@utsce.utc.com
†DISI, University of Trento, Italy – e-mail: first.last@unitn.it

***Abstract*—The design of large scale complex systems demands the ability to correctly specify and verify as early as possible in the design cycle the interaction of the different components that ensure that the global level requirements are satisfied. We address this issue using an approach based on the notion of contract. In particular, we propose a graphical and text-based language for requirement definition that allows designers to incrementally and hierarchically construct contract specifications for system components by composing a set of simple and intuitive patterns. The patterns have a formal semantics, and are implemented as monitor components in the Simulink framework for run-time verification. The contracts are simulated together with the components to verify both satisfaction and compatibility. A cruise control case study demonstrates the effectiveness of the approach.**

I. Introduction

The verification of complex and large scale systems, such as an airplane or an automobile, is becoming a primary concern in the industry. Of particular importance when integrating this kind of systems is the ability to correctly specify and verify as early as possible in the design cycle the interaction of the different components that ensure that the global level requirements are satisfied. The current practice is based on flowing of requirements written in natural languages, to/from different companies, and verification is typically done locally with only partial information of the other parts of the system, subject to considerable interpretation. The integration of a system wide model is therefore very partial and covers only few aspects with limited capability to perform a cross-company verification.

In this paper, we address the issue of requirement definition and their verification in the context of a distributed system using a simulation infrastructure. Because of the distributed nature of the design and of the organization, which is fragmented in several companies that cooperate to reach the final result, it is essential to provide a precise specification of the properties that each component should satisfy, and the assurance that those properties together are sufficient to guarantee the requirements. We thus base our requirement capture methodology on the notion of *contract*, which favors the definition of a correct interaction mechanism by making both the assumptions and the guarantees of a component explicit. Contracts are abstractions of system components which express their desired properties, as well as their correct context of use, therefore clearly separating the responsibilities of the different suppliers and the system integrator. Moreover, contracts can be made available early in the design cycle, enabling concurrent independent implementability [1] and system verification. To simplify their correct definition, we introduce a text-based and a graphical-based pattern language that makes it easy for the designer to visualize and compose different requirements. Executable versions of the contracts, which are automatically generated from the specification, are embedded in the simulation framework in the form of monitor components that continuously interact with the system to verify that the history of their inputs and outputs is consistent with the specification. At the end of the simulation, designers are able to check for any violation of the system properties, and take appropriate corrective actions.

II. Related Work

The definition of a user-friendly and semantically precise language is considered a key aspect for the successful application of contract-based methodologies in the industrial context. Lee and Sokolsky propose a graphical language for the description of temporal formulae [2]. Even though this language supports a great level of expressiveness, it misses all the user-level facilities that augment its usability. In the SPEEDS European Project, a pattern-based language called Contract Specification Language (CSL) is formalized [3]. This language provides the user a number of parameterizable concepts (patterns), which are translated into the underlying SPEEDS HRC model [4]. However, the notion of time (or even logical step) is not explicitly part of the language, instead CSL inherits the implicit time defined by the basic SPEEDS HRC semantics. The Requirement Specification Language [3] is a CSL-based pattern language that provides additional support for the composition of patterns. However, RSL does not provide the capability of specifying liveness-based requirements. Othello is another contract specification language, based on a synchronous hybrid temporal logic [5], focusing on building a formal proof system for contracts. While supported by a set of analysis tools, the language is not oriented towards run-time verification, which is an essential feature for approaching large systems for which formal verification is impractical. Moreover, to make the generation of proof obligations feasible, the contract hierarchy is limited to following the component hierarchy. In this paper, we overcome these limitations and propose a textual and graphical language for the specification of contracts called BCL. Contracts can be developed independently of the component structure. However, our implementation in Matlab/Simulink allows the designer to mix the contract and the component definitions in the same framework, naturally extending the traditional component only approach.

Formal verification methods make use of analytical analysis of the model to establish whether a property is true under all possible conditions and input patterns. Passerone et al.

describe the process from a theoretical point of view, and highlight the similarities between verification and synthesis [6]. Quinton et al. have developed a hierarchical approach that deals with the problem of checking contract dominance [7]. Later, Raclet et al. have developed a theory of contracts based on modal specifications implemented in the InterSMV toolset, dedicated to checking dominance and refinement between contracts [8]. More recently, contract-based specification methods were extended to timed models by David et al. [9]. The use of formal methods in our context is however limited, because they are generally computationally complex, and are unable to handle the large state space found in typical systems. Secondly, and more importantly, formal methods are not yet able to integrate components for which a formal model does not exist (because, for instance, they are only available as shared binary libraries). For this reason, our approach is oriented towards semi-formal simulation-based methods. Monitor techniques, or *observers*, have been used extensively in system design through simulation. In our previous work [10] we discuss these techniques [11], [12], [13], [14], [15] and address the problem of asynchronous interaction. In this paper, instead, we focus on the language issues. In particular, we discuss the specification of formal requirements using a *pattern* based language. The structured and formal nature of the patterns allows for efficiently synthesizing monitors, reducing the complexity of the monitoring problem.

III. Motivational Example

In this section we describe a design scenario that will be used throughout the rest of the paper as a case study to show both contract authoring and contract verification. The application is an embedded system that regulates the speed of a vehicle based on a set of commands provided by the user. We consider a modified version of the model proposed by Aldrich [16]. The top level view of the system, modeled in Matlab/Simulink, is shown in Figure 1. The system

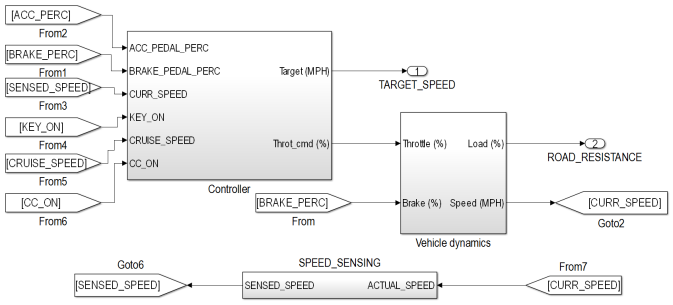

Fig. 1. Cruise control system top level view in Matlab/Simulink

is composed of a discrete time controller, a model of the vehicle dynamics and a speed sensor. The top level has as input the acceleration and brake pedal pressures (signals ACC_PEDAL_PERC and BRAKE_PEDAL_PERC) in percent, the desired cruise speed (signal CRUISE_SPEED) and two additional Boolean inputs representing the status of the ignition key (KEY_ON) and the activation status of the cruise control (CC_ON). The system outputs the speed of the car (SPEED, measured in miles per hour). The Controller acts on the speed of the car by modulating the aperture of the engine throttle (signal Throttle_cmd, with values between 0 and 4000). When the cruise control is active, the aperture value is automatically computed to meet the cruise speed requirements; otherwise it is directly regulated by the pressure of the acceleration pedal. The Vehicle Dynamics mimics the behavior of the vehicle using discretized integral equations that compute the speed as a function of the controls and of the profile of the resistance of the road, expressed as a percentage value (input LOAD_PERC). Finally, the Speed Sensor models the speed measuring process taking into account the presence of possible errors due to uncertainty. Several requirements can be expressed for this system. Table I lists some of the natural language properties that we would want to verify, separated into assumptions and guarantees, and assigned to the individual components of the system or to the system as whole. For instance, requirements r_1 and r_5 represent bound-

Id	Requirement	A/G	Comp.
r_1	Once the driver turns on the car, she will eventually turn it off	A	System
r_2	A throttle aperture command corresponds to a movement of the car	A	Control
r_3	A brake command disengages the cruise control within 3 seconds	G	Control
r_4	When the cruise control is on, the desired speed is reached with a maximum error of 5 mph within 10 seconds	G	System
r_5	Road slope corresponds to a resistance value between 0 and 30%	A	System
r_6	Positive brake pressure corresponds to reduction in speed	G	Dynamics
r_7	Sensed speed is equal to real speed with a maximum error of 5 mph	G	Sensor
r_8	Vehicle speed is always between 0 and 250 mph	A	Sensor
r_9	Brake pedal pressure grows at a maximum rate of 50% per second	A	Dynamics

TABLE I. Natural language requirements as assumptions and guarantees for the cruise control

ary conditions for the correct use of the cruise control system, while requirements r_3 and r_4 express performance properties. Certain consistency properties can also be expressed, as for requirement r_6.

IV. Block-Based Contract Language

The specification of the system starts from a set of informal requirements which are formalized as *contracts*, expressed using either a graphical block-based or text-based pattern language. In brief, a contract $C = (A, G)$ is a specification of a component that defines the properties G that a component must be able to *guarantee*, and the context A or the *assumptions* under which the guarantee must be established [17], [3]. This assumption on the environment is what distinguishes a contract model from a traditional component model. When the assumptions are not satisfied, a component is free to behave as desired, potentially violating its own guarantees. The availability of the pair of specifications $C = (A, G)$ opens up the possibility for checking conditions which are not limited to property checking. In particular, we are interested in two essential aspects:

- Satisfaction: the verification that a component satisfies its contract, i.e., that when placed in a context that

meets the assumptions A, the component is able to guarantee the properties in G.

- Compatibility: the verification that components work well in combination, i.e., that the guarantees of one are able to meet the assumptions of the other, and vice-versa.

Our approach to contract formalization uses a language based on blocks which allows the user to specify contracts using a high level abstraction mechanism. We call this language *Block-based Contract Language*, or BCL. The language consists of a set of blocks that can be seen either as standard parameterizable elements with specific semantics or as operators. The former are called *patterns*, and represent high-level parameterizable requirements. In the remainder of this section we present the structure of the language and express its semantics in LTL. A pattern is a structured and formalized sentence containing free uninterpreted expressions, to be filled out by the designer. A pattern can be *atomic* (part of a predefined library), or *composite*, obtained by composition of other patterns using a set of operators. The actual assertions that constitute assumptions and guarantees of contracts are simply instantiations of patterns which bind a concrete expression to each free expression of the pattern. Each pattern is composed of a *condition*, which restricts the values of the signals, and of a *temporal specification*, which defines when the condition must be true. On top of this, we use quantification and pattern composition to hierarchically construct more complex assertions out of the basic ones, in a layered fashion. We review these aspects in the rest of this section. The condition of a pattern can be either the *occurrence of an event* or a *boolean expression* built on signal values. An event represents a condition that happens at a given instant and does not carry any value: it may be present or absent. Boolean expressions are conditions that can be true or false at every time instant. The patterns **[E] happens** and **[C] holds** constrain the event **E** to occur and the boolean expression **C** to be true, respectively. Two more patterns are used as implications: **Everytime [E] then [C]** and **Everytime [E] then [E2]** assert that whenever event **E** occurs, the expression **C** is true or event **E2** occurs, respectively. The expressions **E** and **C** are called *event* and *condition* expressions. A condition expression is built on top of the model input and output signals using assertions, relations and logical connectives. These elements can be expressed either textually, as operators, or as *blocks* in a graphical format. Each block takes as input the operands of the operator, and provides as output the corresponding result, which can be used as input of another block. This way, one can graphically construct complex expressions. Every signal x can be evaluated at time instant k with the expression $x[k]$, which form the basic terms of the expressions. A complete expression is then built using the operators according to the following grammar:

$$\begin{aligned} t &:= x[k] \\ r &:= t_1 = t_2 \mid t_1 < t_2 \mid \text{TRUE}(x[k]) \mid \text{FALSE}(x[k]) \\ C &:= r \mid \sim r \mid r_1 \wedge r_2 \mid r_1 \vee r_2, \end{aligned}$$

where the semantics of each operator is derived from its conventional mathematical meaning. To simplify the specification, the usual derived relational operators ($\neq$, $>$ and $\geq$) and connectives ($\vee$, $\rightarrow$, $\iff$) are also provided. An *event expression* is constructed from signals by identifying the occurrences of specific conditions. In particular, one can turn any boolean expression **C** into an event, using the operator **ev(C)**. A *timeout* event identifies a logical event that is present when a counting process reaches a given threshold. The timeout block provides a native counting functionality with an input signal that represents the start counting signal and an output signal that represent the timeout event. The textual representation of a timeout event that has a starting event **E** and a timeout of **N** steps is **timeout(E; N)**. Signal condition blocks allow for capturing the change of value of a signal. A *rising edge* condition takes as inputs a signal u and a reference value v. The event is present if the conditions $u[k-1] < v$ and $u[k] \geq v$ are verified. The textual representation of this event is **re(u, v)**. A *falling edge* condition is defined dually, and represented as **fe(u, v)**. Rising or falling edge signal condition blocks can be combined using the logical operators, with the exception of negation, to construct more complex situations. In particular, we natively provide the *rising or falling edge* condition as **rfe(u, v)**. Temporal constraints are used to identify the safety or liveness nature of the specification. Each constraint takes as input a portion of assertion, which we identify with the symbol ϕ, and defines an interval over which ϕ must hold. The simplest patterns refer to the validity of the ϕ assertion in the time interval that goes from the current time to indefinitely in the future. In particular,

- **[ϕ] always** represents a safety constraint on ϕ that imposes to the formula be true at every time instant. Formally, the specification specifies that for each time instant k, $\phi[k]$ is true.
- **[ϕ] infinitely often** constrains the input ϕ to be true infinitely often. This means that for every time instant k, there exists a time instant $j \geq k$ such that $\phi[j]$ is true.
- **[ϕ] eventually** constrains the formula ϕ to be true at some time instant in the future. The specification is satisfied if there exists a time instant k such that $\phi[k]$ is true.

Patterns are also provided in which it is possible to specify a defined time interval **I**. An interval specification identifies a set of time instants delimited by a start event and an end event. Intervals may be open or closed at both ends obtaining four possible combinations. Formally, an interval is the set of time instants between the start event and the end event each defined using one of the event blocks described in the previous paragraphs. As an examples of interval specifications consider the interval specification expression **[re(x,1), fe(x,1)]** that formalizes an interval that starts when signal x takes value 1 with a positive derivative and ends when it assumes value 1 with a negative derivative. As an example of pattern that uses the interval specifications consider **[ϕ] during [I]** constrains ϕ to be true during the time instants identified by the interval. The formula evaluates to true if, at every k-th time instant, $k \notin I$ or $k \in I$ and $\phi[k]$ is true. The patterns and the operators seen so far are already sufficient to model a large variety of requirements. In particular, we are interested in formalizing the requirements listed in Table I. The act of formalization accomplishes various goals. One is to convey the intent of the natural language requirements unambiguously.

Another is to express the requirements in a form that is analyzable and/or executable by tools, to make an efficient use of the specification. And finally, formalization helps the designer refine the requirements by making them more precise. Patterns are particularly well suited for this, since their predefined structure requires that certain templates be followed that include the necessary information. Table II shows the formalization of the requirements for our cruise control system. Here, the **der** operator is used to compute the first derivative of

Id	Formalization	A/G	Comp.
r_1	[KEY_ON] **implies** [KEY_OFF] **eventually**	A	System
r_2	[THROTTLE $> 0 \rightarrow$ SPEED > 0]	A	Control
r_3	**Everytime** [BRAKE_PERC > 0] **then** [THROTTLE $= 0$] **within** [3 s]	G	Control
r_4	**Everytime** [CC_ON] **then** [**abs**(SPEED - CRUISE_SPEED) < 5] **within** [10 s]	G	System
r_5	[$0 \leq$ ROAD_RESISTANCE ≤ 30] **always**	A	System
r_6	**Everytime** [BRAKE_PERC > 0] **then** [**der**[SPEED] < 0]	G	Dynamics
r_7	[**abs**(SPEED $-$ SENSED_SPEED) ≤ 5] **always**	G	Sensor
r_8	[$0 \leq$ SPEED ≤ 250] **always**	A	Sensor
r_9	[**der**(BRAKE_PERC) < 50] **always**	A	Dynamics

TABLE II. FORMALIZED REQUIREMENTS FOR THE CRUISE CONTROL

the argument over time. These examples show the expressive power of the pattern language, which could be sufficient for our case study. More complex cases require additional facilities to iterate through signals and to compose requirements, as described next.

Quantification patterns provide a syntactic mechanism for the logical unfolding of requirements. The universal quantifier $\forall$, or *for every signal* block, applies the input signals to a formula R and computes their logical conjunction. The semantics of $\forall s \in S.R(s)$ corresponds to the equivalent formula $R(s_1) \wedge R(s_2) \wedge \cdots \wedge R(s_n)$, where $S = \{s_1, \ldots, s_n\}$. As an example, the requirement that "no two signals in a set S are both positive" can be expressed as

$$\forall s', s'' \in S. \sim [(s' > 0) \wedge (s'' > 0)]$$

A corresponding formula without quantifiers grows quickly with the number of signals. The dual existential operator $\exists$ is also available. The semantics of the expression $\exists s \in S.R(s)$ is the disjunction $R(s_1) \wedge \cdots \wedge R(s_n)$.

A key element of the proposed language is the capability to compose patterns to derive more complex ones. A number of *pattern composition* blocks are used to derive a composite pattern C out of two patterns A and B. The following composition patterns are available in the language:

- **AND** ($\&$): $A \,\&\, B$ is an assertion that identifies a set of behaviors satisfying both assertion A and B;
- **OR** ($|$): $A \mid B$ is an assertion that identifies the set of behaviors satisfying assertion A or assertion B;
- **IMPLIES** ($\Rightarrow$): $A \Rightarrow B$ is an assertion that identifies the set of behaviors that do not satisfies A or that satisfies both A and B;
- **NOT** ($\sim$): $\sim A$ is an assertion that identifies the set of behaviors that do not satisfy A (complement set).

An assertion is identified using an assertion definition block that has as (unique) parameter the name for the assertion. A valid assertion is obtained by the composition of a condition assertion, its conditions and a temporal specification. The assertion block can be placed at the end of the temporal specification block or it can be attached to a composition block to identify the compound assertion. Examples of composition will be shown in Section IV-A to construct the contracts for our case study. In the previous sections we defined the available patterns and the means of composition. The formal semantics is defined by mapping the patterns into the LTL specification language. This mapping can also serve as an implementation when we want to use an analysis engine that supports this type of specification of requirements. In order to translate the requirement into temporal logic, we must primarily find a way to express the pattern temporal specification. An interval specification of type **I = [E1, E2]** can be implemented using the *until* operator. For instance, the pattern **Everytime [E] then [C] holds during [I]** is translated as follows:

$$G(E \Rightarrow \neg E1\, U\, (E1 \wedge C \wedge (C\, U\, (E2 \wedge C)))),$$

where the first until is used to wait for the occurrence of **E1**, while the second wait for the occurrence of **E2**. The case of an open or semi-closed interval can be handled analogously. The patterns that do not make use of an explicit interval specification map easily onto LTL using the corresponding operators. The complete semantics specification is lengthy, and the details will be omitted in this paper. Contracts are pairs of assertions (namely the assumption and the guarantee). The contract specification layer includes a specific block for the definition of contracts which identifies the assumption and guarantee using separate parameters. Composition of contracts is similar to pattern composition, however the semantics of the composition operators follows that of contract models, rather than that of traditional logic models. For instance, for conjunction, contracts take the *disjunction* of the assumptions and the *conjunction* of the guarantees, to ensure that the refinement relation (contract dominance) is preserved [18].

A. Simulink Language Implementation

The textual assertion specification language described in the previous section is useful in formalizing the requirements using signals, patterns and their operators. An even more convenient representation is a graphical format, that can express the specification in a way similar to the implementation model. Our Block-based Contract Language is of this kind. For our implementation, we have chosen the Matlab/Simulink software, which is the standard-de-facto for modeling discrete and continuous time dynamic systems and control algorithms. To support contract-based design in BCL, we have developed a Simulink toolbox implementation in which the different specification layers of the language have been implemented as executable library elements supporting constructs for the specification of events, atomic assertions and composition. Patterns are also represented as blocks, encapsulating a complex specification. This approach has two advantages. First, the user is able to provide the specification in a graphical environment which is easier to learn and visualize, and with which many designers are already familiar. This greatly decreases the steep learning curve associated with learning a new formalism. In the second place, we can take advantage of the many facilities already provided by the hosting language, such as the built-in operators and functions, which enrich the specification language with no

additional cost. In addition, we can make use of the code generation functionality offered by the Simulink framework to help automatically generate the executable version of the assertions. The individual assertions are combined to form contracts, by assembling together the assumptions and guarantees of each component. A composition operator combines the assertions for the assumption. The guarantee is instead composed of a single pattern, a time-related assertion in which the time bound is computed from the requirement (time interval of 3 seconds) and the sampling time of the discrete systems (50 milliseconds). Figure 2 shows the contract for the controller, formed by combining requirements r_2 and r_3. This is a simpler

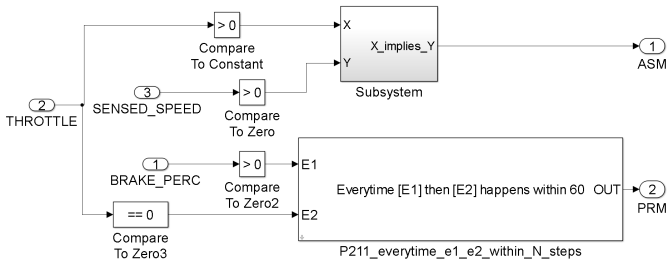

Fig. 2. The controller contract

specification that uses the patterns directly to formalize the assertions. The contract for the Vehicle Dynamics can be obtained similarly, and is not shown here.

V. Contract verification

Our monitor-based contract verification method uses simulation to keep track of the validity of the assumption and guarantee assertions in the contract. A monitor is a component that verifies at run-time a specific relation between its inputs. The assumptions and promises are described using BCL and linked to their corresponding component. Each contract-enriched component exposes two additional boolean output signals: signal a, which is updated with the violation status of the assumptions, and signal g which holds the violation status of the guarantees. The verification technique based on monitors has to be carefully applied to the verification of liveness constraints. The MATLAB Simulink based implementation of the BCL language follows the approach described by Bauer et al. [19]: given a partial execution of a model (i.e., a finite trace), each monitor output can be:

- TRUE, if the property checked by the monitor *holds irrespective* of the future evolution of the system;
- FALSE, if the property *does not hold irrespective* of the future evolution of the system;
- UNDEFINED, if the property has not yet been verified or disproved up to the current execution time, but it could be verified or disproved in the future.

Using the two additional signals, it is possible to evaluate at run-time the satisfaction and compatibility relations between the component implementation and the contract. If we denote by a_i and g_i the value of the monitor status signal at time instant i, the contract is satisfied by the component if and only if, for every i, $a_i \rightarrow g_i$, and the system is compatible if for all monitors $a_i \leftrightarrow$ TRUE. We must, however, extend the implication operators to account for the extra undefined value. This can easily be accommodated as shown in Table III.

A/G	undefined	violated	satisfied
undefined	*undefined*	*undefined*	*satisfied*
violated	*satisfied*	satisfied	satisfied
satisfied	*undefined*	violated	satisfied

TABLE III. Extension of the implication operator

We applied the contract-based verification methodology to the analysis of the cruise control system. The system components and their interconnections were modeled in Simulink, as described in Section III, and the corresponding contracts were attached to the components, as discussed in Section IV-A. Then, the run-time verification capabilities of the tool have been exploited to perform a satisfaction and a compatibility relation check. Figure 3 shows a plot of the results of the analysis. Our simulation scenario starts with the car and the

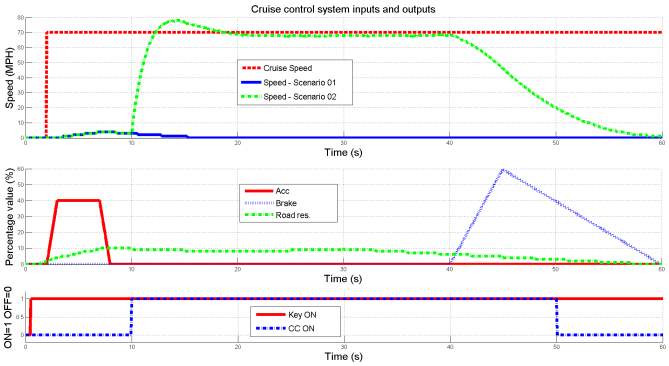

Fig. 3. System behavior during simulation

cruise control initially switched off. Then, the car is turned on and the driver presses the acceleration pedal. After a few seconds, the desired cruise speed is set, after which the accelerator pedal is released, and the cruise control is switched on (at time 10). At this point the system reaches the desired speed. At time 40 seconds, the brake pedal is pressed and eventually the driver re-gains control of the car disengaging the controller. During the entire simulation, the system was provided with a specific road resistance profile, simulating a landscape where an initial increase of the road slope is followed by a slow decrease. We exploited the simulation capabilities of the Simulink software to execute the run-time verification of the contract compatibility and satisfaction relation.

A. Contract compatibility verification

The compatibility relation can be checked using run-time verification only for closed systems (i.e., systems not containing free inputs). In this case, the contracts are not compatible if at least one of the modeled assumptions is violated. Incompatible contracts cannot be safely composed and intuitively expose logical incoherence between the requirements. The simulation of the cruise control system exposed the incompatibility between the cruise control and the vehicle dynamics contracts, which leads to the falsification of the assumptions of the former. The problem lies in the formalization

of the requirement r_2, which does not take into account the response time of the engine. The proposed formalization, in fact, asserts that there should be an *instantaneous* response of the vehicle as a consequence of the accelerator pedal pressure whereas the vehicle dynamics contract only constraints the behavior of the car in presence of a brake pedal pressure allowing the implementation to react to an accelerator pressure within an arbitrary amount of time (Requirement r_6). The implementation of the dynamics models properly the physics of the car reacting to an accelerator pressure in a non-zero time satisfying, on one hand, the vehicle dynamics contract and violating, on the other, the controller assumption. This violation shows an internal incompatibility between these two contracts w.r.t. the implementation model and the input trace used to close the system. To solve the incompatibility, the violated assumptions has been refined using the less restrictive formulation

$$r_2 : \textbf{Everytime } [\text{THROTTLE} > 0] \textbf{ then } [\text{SPEED} > 0] \textbf{ within } [1 \text{ s}]$$

to take into account the response time of the physical parts. The new formulation of the assumption does not expose incompatibility between the contracts.

B. Contract satisfaction verification

The contract satisfaction relation can likewise be verified or falsified for a closed system at run-time by evaluating the status of the executable monitors. For the system under analysis, the run-time verification exposes a contract violation due to an implementation error of the throttle aperture control, which does not allow the controller to sense the CC_ON signal. After the cruise control is turned on (at step 10 seconds), the car speed does not increase reaching the desired speed within 10 seconds as stated by requirement r_4, leading to the violation of the contract. A correct implementation of the controller has been implemented in a second scenario, which solves the satisfaction violation.

VI. Conclusions

We have presented the Block-based Contract Language, a pattern-oriented graphical and textual language for the specification of contracts for system components. The graphical notation was implemented in Simulink, making the contract methodology a natural extension of the system design. In our current work, we are studying how to employ different interaction paradigms for modeling, to allow both synchronous and asynchronous components to be part of the system, and introducing hardware in the loop in the verification infrastructure, to allow contracts to be verified against a physical prototype implementation of the components.

Acknowledgments

The authors would like to acknowledge the support of the SPRINT EU project[1] (grant agreement no: 257909) and the EU ARTEMIS Joint Undertaking under grant agreement no. 269335 (project MBAT[2]) and the Italian Ministry of Education, University and Research (MIUR).

[1] http://www.sprint-iot.eu/

[2] https://www.mbat-artemis.eu/home/

References

[1] T. A. Henzinger and D. Ničković, "Independent implementability of viewpoints," in *Proc. of the* 17^{th} *Monterey conference on Large-Scale Complex IT Systems: development, operation and management*, Oxford, UK, 2012.

[2] I. Lee and O. Sokolsky, "A graphical property specification language," in *Proc. of the* 2^{nd} *IEEE Workshop on High-Assurance Systems Engineering*, Washington, DC, USA, August 11-12, 1997.

[3] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *Proc. of the Conference on Design, Automation and Test in Europe*, Grenoble, France, March 14-18 2011.

[4] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis, "A contract-based formalism for the specification of heterogeneous systems," in *Proc. of the Forum on Specification, Verification and Design Languages*, Stuttgart, Germany, Sep. 2008.

[5] A. Cimatti and S. Tonetta, "A property-based proof system for contract-based design," in *Proc. of the* 38^{th} *Euromicro Conference on Software Engineering and Advanced Applications*, ser. SEAA12, Cesme, Turkey, September 5-8 2012.

[6] R. Passerone, L. d. Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: Two faces of the same coin," in *Proc. of the* 20^{th} *IEEE/ACM International Conference on Computer-Aided Design (ICCAD02)*, San Jose, California, November 10–14, 2002, pp. 132–139.

[7] S. Quinton and S. Graf, "Contract-based verification of hierarchical systems of components," in *Proc. of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, ser. SEFM '08, Washington, DC, USA, 2008.

[8] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone, "A modal interface theory for component-based design," *Fundamenta Informaticae*, vol. 108, no. 1–2, pp. 119–149, 2011.

[9] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed I/O automata: a complete specification theory for real-time systems," in *Proc. of the 13th ACM international conference on Hybrid systems: computation and control*, ser. HSCC '10, Stockholm, Sweden, 2010.

[10] O. Ferrante, R. Passerone, A. Ferrari, L. Mangeruca, C. Sofronis, and M. D'Angelo, "Monitor-based run-time contract verification of distributed systems," in *Proc. of the* 9^{th} *IEEE International Symposium on Industrial Embedded Systems*, Pisa, Italy, June 18–20, 2014.

[11] F. Balarin and R. Passerone, "Specification, synthesis and simulation of transactor processes," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 10, Oct. 2007.

[12] A. Davare, D. Densmore, L. Guo, R. Passerone, A. L. Sangiovanni-Vincentelli, A. Simalatsar, and Q. Zhu, "METROII: A design environment for cyber-physical systems," *ACM Transactions on Embedded Computing Systems*, vol. 12, no. 1s, pp. 49:1–49:31, March 2013.

[13] Y. Kakiuchi, A. Kitajima, K. Hamaguchi, and T. Kashiwabara, "Automatic monitor generation from regular expression based specifications for module interface verification," in *Proc. of the International Symposium on Circuits and Systems (ISCAS05)*, May 2005.

[14] J.-Y. Brunel, M. Di Natale, A. Ferrari, P. Giusto, and L. Lavagno, "Softcontract: an assertion-based software development process that enables design-by-contract," in *Proc. of the conference on Design, automation and test in Europe (DATE04)*, 2004.

[15] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Formal Methods in System Design*, vol. 34, no. 3, June 2009.

[16] W. Aldrich, "Coverage analysis for model based design tools," in *Proc. of the* 18^{th} *International Conference and Exposition on Testing Computer Software*, ser. TCS'2001, Washington, DC, June 18-22 2001.

[17] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *European Journal of Control*, vol. 18, no. 3, pp. 217–238, 2012.

[18] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects,* 6^{th} *International Symposium (FMCO 2007)*, October 24–26 2008, vol. 5382.

[19] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, 2011.