Monitor-Based Run-Time Contract Verification of Distributed Systems

Orlando Ferrante, Roberto Passerone, Alberto Ferrari,
Leonardo Mangeruca, Christos Sofronis, Massimiliano D'Angelo
Advanced Laboratory on Embedded Systems S.r.l., via Barberini 50, Rome, Italy
e-mail: name.surname@ales.eu.com

***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 to ensure that the global level requirements are satisfied. We address this issue using an approach based on the notion of contract and simulation-based verification. In particular, we extend traditional contract verification methods to target distributed systems, which require an asynchronous communication paradigm. We use a pattern-based language for requirement definition, from which we generate a set of contract monitors implemented in the Simulink framework to observe the underlying system execution and flag violating behaviors. In the paper, we discuss in particular the aspects related to handling the asynchronous interaction between components and their relation to the contract monitors. An automatic towing system case study demonstrates 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, 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 both the issue of requirement definition and their verification in the context of a distributed system using run-time verification and a simulation infrastructure. Because of the distributed nature of the design, which makes use of network services for the communication among components, as well as the distributed nature of the organization, it is essential to provide a precise specification of the properties that each component should satisfy, and develop a faithful model of the component interactions. 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. Our specific contributions extend in two directions. First, we introduce a pattern-based contract specification mechanism that allows the user to map, visualize and compose his/her requirements into high level formalized concepts in an intuitive manner. Contracts 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. In addition, we develop the theoretical background that allows contracts to be adapted to distributed development, such as supporting an interleaving asynchronous semantics, which is more appropriate in a distributed settings where components run at independent speeds. At the end of the simulation, designers are able to check for any violation of the system properties, and take appropriate corrective actions.

Related Work. Monitor techniques, or *observers*, have been used extensively in system design through simulation. Typically, monitors are obtained by synthesis from higher level specifications, expressed in the form of some logic or regular expression, which includes temporal properties, conveniently translated into a state-based intermediate format, before being compiled to executable form. Balarin et al. make use of the Property Specification Language (PSL) to represent the properties, which can then be translated into automata, and eventually compiled as Verilog or C code, for both hardware and software monitoring [1]. In MetroII, constraints and constraint solvers are built into the modeling language to express properties that can be directly related to the design to enforce the satisfaction of logical and time-based conditions [2]. Kakiuki et al. [3] apply monitors to the verification of hardware designs, while Brunel et al. [4] apply a technique similar to the logic of constraints to the design of embedded software, where requirements are in the form of contracts. Tripakis et al. [5] also discuss how to generate a diagnoser from a timed automaton specification. We approach this problem by focusing on the specification of formal requirements using a *pattern* based language [6]. The structured nature of the patterns allows for efficiently synthesizing monitors, reducing the complexity of the monitoring problem.

Damm et al. formalize a pattern-based language for the specification of safety contracts called Contract Specification Language (CSL) [7]. This language provides the user a number of parameterizable concepts (patterns), which are translated into the underlying model [8]. Several shortcomings make this inadequate for our purposes. First, the underlying fully synchronous semantics is too constraining for distributed systems. In addition, the notion of time (or even logical step) is not explicitly part of the CSL language, which inherits the implicit time defined by the underlying model. Finally, there is no means in CSL to compose assertions, limiting its usability.

II. Methodology and Specification Language

The specification of the system starts from a set of informal requirements which are formalized as *contracts*, expressed using a pattern-based 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 [9], [7], [10]. The 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. We are interested in verifying 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. When this is the case, we say that the contracts (and the underlying components) are *compatible*. A satisfaction check can be used to establish that components satisfy their contract. Establishing that components satisfy their contracts, and that contracts are mutually compatible, is sufficient to establish the correctness of the design with respect to the specification.

Our approach to contract formalization and verification 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. Blocks have both a textual representation and a graphical Simulink representation. A pattern is a structured 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. 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. Patterns are built using a layered specification. The first layer is concerned with the definition of events. An *event expression* is constructed from signals by identifying the occurrences of specific conditions. One can turn any boolean expression **C** into an event, using the operator **ev(C)**. Signal condition blocks capture 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, where k represents time. The textual representation of this event is **re(u, v)**. A *falling edge* condition is defined dually, and represented as **fe(u, v)**. The second layer is used to construct atomic patterns. Each pattern is composed of a *condition* and of a *temporal specification*, which defines when the condition must be true. The condition of a pattern can be either the *occurrence of an event* or a *boolean expression* built on signal values. The pattern **[E] happens** constrains 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. Every signal x can be evaluated at time instant k with the expression $x[k]$, which form the basic terms of the expressions. Temporal constraints are used to identify the safety or liveness nature of the specification. Each constraint takes an assertion ϕ and defines an interval over which ϕ must hold. The simplest patterns refer to the validity of ϕ in the time interval that goes from the current time to indefinitely in the future. An example of interval constraint is the **[ϕ] always** fragment that represents a safety constraint on ϕ that imposes the formula to be true at every time instant. Patterns may also specify a defined time interval **I**. Formally, an interval is the set of time instants between a start event and an end event, each defined using one of the event blocks described in the previous paragraphs. Patterns make use of temporal constraints as follows: **[ϕ] during [I]** constrains ϕ to be true during the time instants identified by the interval; and **[ϕ] at least once within [I]** constrains ϕ to be true in at least one time instant in **I**. A key element of the proposed language is the capability to compose patterns to derive more complex ones. The third layer provides a number of *pattern composition* blocks that are used to derive a composite pattern C out of two patterns A and B. The standard composition patterns are available in the language: $A \,\&\, B$ is an assertion that identifies a set of behaviors satisfying both assertion A and B; $A \mid B$ identifies the set of behaviors satisfying assertion A or assertion B; $A \Rightarrow B$ identifies the set of behaviors that do not satisfies A or that satisfies both A and B; and $\sim A$ is an assertion that identifies the set of behaviors that do not satisfy A (complement set).

The top layer is concerned with contracts, which are simply defined as pairs of assertions (namely the assumption and the guarantee). 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 [8]. The textual assertion specification language described in the previous section is useful in formalizing the requirements using signals, patterns and their operators. For our implementation, we have chosen the Matlab/Simulink software[1], 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.

III. Implementing asynchronous contract monitors

An asynchronous interaction can be obtained by adding "dummy" self-loop transitions to every state, which are taken whenever a *stuttering* behavior of the component is desired. From the infrastructure point of view, the difference is therefore limited to letting components sit idle when necessary. The scheduler, however, must be able to handle the different situations, and activate the components whenever they share a signal and must therefore synchronize. Components may synchronize by either sharing signals, or by using alternate synchronization paradigms denoted by rich connectors [11]. In addition, data may be delayed and/or buffered along a communication channel. Whereas the difference between synchronous and asynchronous execution is fundamental, and therefore

[1]http://www.mathworks.it/products/simulink/index.html

requires a change in the simulation strategy, different communication paradigms can be handled from a behavioral point of view, by adding specific extra components that mediate the communication. In particular, a purely asynchronous semantics is unable to represent delays, since components must be speed independent. Delays must therefore be represented by explicit quantities, that bound the unrestricted non-determinism that results from the use of asynchronous components.

At the semantics level, the asynchronous interaction differs from the synchronous case only for those transitions which independently generate some new events. If a component resides in a state from which all transitions are guarded by an input event, then the component will necessarily have to follow the transitions that are enabled according to the presence/absence of the input events, and will stutter (i.e., stay in the current state) only if none of the input events is enabled. In this situation, therefore, no interleaving is possible, and the state machine behaves as if it had a synchronous interaction with the rest of the system. This condition is particularly interesting in the context of monitor-based verification. A monitor, in fact, seen as a black box is essentially a passive element driven by changes in the signals it observes, and has no outputs. A monitor is therefore not capable of initiating any interaction with the component it is attached to. As a consequence, transitions are either not guarded at all, and do not produce any visible effect outside the monitor, or are guarded by some input event, leading to the above situation. For this reason, it makes no difference whether we consider the monitor to be synchronous or asynchronous with its corresponding component, since transitions are in any case synchronized. Therefore, for the purpose of this work,

> monitors are always synchronous with the monitored component.

We stress that this does not imply that contracts necessarily interact synchronously among themselves. Instead, because monitors interact exclusively with the component they are attached to, and because they do so synchronously, their mutual interaction is mediated by the interaction paradigm of the corresponding component. In other words, contract composition is delegated to the composition of the corresponding component implementations. This delegation mechanism is particularly convenient, as changes in the interaction between components (for instance by the addition of delays or buffering, or through the use of a rich connector) are immediately reflected at the level of the contracts. Thus, our implementation approach is that of adding a minimum of extra capabilities to the simulation infrastructure in order to support asynchronous interactions. The other orthogonal aspects are instead handled from outside the infrastructure using supportive components added to the system to account for the intended synchronization behavior.

IV. Automatic Towing System

The Automatic Towing System (ATS)[2] is a complex distributed system composed of several agents that automate road service calls. The system is composed of tow-bots, i.e., self-driving tow trucks that provide support to user vehicles in a predefined service area orchestrated by a command, control, communications and intelligence subsystem (C4I). To illustrate our methods, and to apply contracts and contract analysis to the ATS case study, we restrict our analysis of the system to the interaction between the C4I, a single damaged user vehicle and a tow-bot. The top level view of the system, modeled in MATLAB/Simulink, is shown in Figure 1. The ATS is

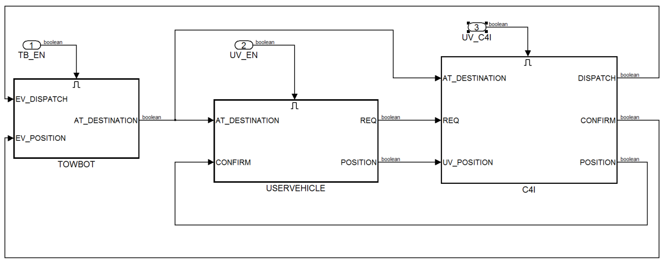

Fig. 1. ATS system block diagram

an asynchronous, distributed system whose communication of messages relies on wired and wireless network infrastructures. In order to model an asynchronous communication between the components, we adopted an approach similar to the one described by Miller et al. [12]. Each component is a subsystem that exposes a functional interface (the set of signals corresponding to the events exchanged with the environment) and an additional control Boolean signal. When the additional signal is active (TRUE), the block performs its computation as usual; when it is inactive (FALSE), the block is disabled (stuttering). The ATS systems should satisfy several requirements to ensure safe user-vehicle towing under all operational circumstances. Some of these requirements, formalized using the BCL pattern language, are represented in Table I identifying for each of them what is the constrained component and whether the requirement is an assumption or a guarantee. Requirements

TABLE I. Formalized BCL requirements for the ATS system

ID	Requirement	Component	Type
R01	**Everytime** [evDispatch] **then** [uvPosition] **follows**	Tow-Bot	A
R02	[evDispatch] **implies** [atDestination] **eventually**	Tow-Bot	G
R03	[evDispatch] **implies** [atDestination] **eventually**	C4I	G
R04	[req] **implies** [uvPosition] **follows**	C4I	A
R05	[evDispatch $\wedge$ uvPosition] **always**	C4I	G
R06	[evReq] **implies** ![uvPosition] **within** (evReq, atDestination)	UV	G
R07	[evReq] **implies** [evReqConfirm] **eventually**	UV	A

R01 and R02 ensure that the tow-bot successfully reaches a car when a dispatch command is sent from the C4I, assuming that the car does not change its position in the meantime. Requirements R03, R04 and R05 require the C4I to properly communicate the dispatch of a tow-bot to a requesting car and guarantee the information are sent to the Tow-Bot. Finally, R06 and R07 ensure a correct behavior of the car.

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 incompatible if at least one of the assumptions is violated. Incompatible contracts cannot be safely composed and intuitively expose logical incoherence between the requirements. On the other hand, in order to verify compatibility of closed (and possibly

[2]http://www.sprint-iot.eu/industrialcase.php

non-deterministic) systems, it is sufficient to verify that a state of the system in which any of the assumptions is violated is not reachable using exhaustive verification. The ATS model is logically a closed system since all input ports are driven by a corresponding output port. Nonetheless, the model has free inputs corresponding to the enable signals of the subsystems used to model the asynchronous execution of components in MATLAB/Simulink. In this case, the environment is the asynchronous scheduler that is capable of enabling and disabling the execution of the subsystems, modeling the non-deterministic behavior of an asynchronous system. Checking incompatibility for this scenario consists of evaluating the presence of a system execution, i.e., a component interleaving, leading to an assumption failure. Conversely, checking compatibility amounts to verifying that all the assumptions are verified under all admissible sub-system executions interleaving. We use executable monitors for incompatibility verification in a run-time context. In the synchronous case, we again use run-time verification, but also employ a formal verification framework [13] to deduce an appropriate schedule. In practice, each system will require the implementation of a scheduler that restricts the non-determinism by enforcing specific timing constraints. Synchronous composition of components and contracts can be obtained by constraining all the enable signals of the components to always occur simultaneously. Using this approach, we found the contracts of the ATS to be incompatible under synchronous composition. To solve this problem we must relax requirement R05 using the following formulation: **Everytime** [evDispatch] **then** [uvPosition] **follows** and the implementation of the C4I has been modified accordingly. For the new model, the compatibility has been successfully verified. To verify compatibility in the case of asynchronous composition, the enable signals of the components have been left free. Each component maintains its state when the enable signal is FALSE and proceeds with its execution when it is TRUE. The formal engine returns a counter-example that shows the incompatibility. For instance, the Tow-Bot assumption can be violated by a specific execution schedule, as depicted in Figure 2. The reason of the violation is related to

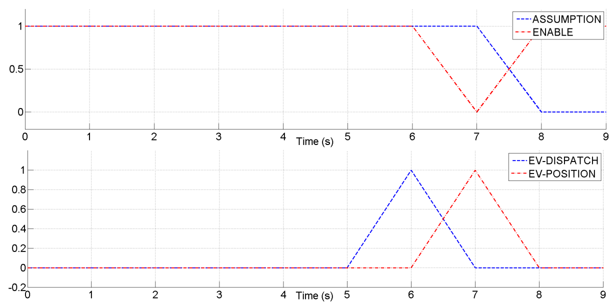

Fig. 2. Incompatibility of asynchronous composition

the formalization of the Tow-Bot assumption that imposes the position signal to be asserted exactly after the reception of the dispatch, without taking into account the possible stuttering of the component. In the case of the Tow-Bot, the incompatibility can be solved relaxing the R02 requirement using the following alternative formalization: [evDispatch] **implies** [atDestination] **eventually**. This alternative representation allows the components to be safely composed, and the formal verification engine returns a positive result.

V. Conclusion

We have presented a system verification methodology that uses contracts and run-time verification through monitors for the validation of distributed systems. Our contribution includes a pattern-based language, called BCL, for the specification of the system constraints, and the foundations and an implementation of an asynchronous interaction paradigm. Our technique for contract verification uses passive monitors which are always synchronous with the underlying components, which are delegated for the implementation of the asynchronous interaction. Our current work includes the development of a distributed simulation infrastructure supporting hardware in the loop integration and heterogeneous simulators.

Acknowledgment

The research leading to these results was partially funded by the SPRINT EU project (grant agreement no. 257909)

References

[1] F. Balarin and R. Passerone, "Functional verification methodology based on formal interface specification and transactor generation," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE06)*, Munich, Germany, March 6–10, 2006.

[2] 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.

[3] 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.

[4] 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.

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

[6] M. Dwyer, G. Avrunin, and J. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. of the Intern. Conf. on Software Engineering*, Los Angeles, CA, May 16-22 1999.

[7] 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.

[8] 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, September 23–25, 2008.

[9] 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.

[10] L. Mangeruca, O. Ferrante, and A. Ferrari, "Formalization and completeness of evolving requirements using contracts," in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, June 2013, pp. 120–129.

[11] S. Bliudze and J. Sifakis, "The algebra of connectors - structuring interaction in BIP," *IEEE Trans. on Computers*, vol. 57, no. 10, 2008.

[12] S. P. Miller, M. W. Whalen, M. P. Heimdahl, and A. Joshi, "A methodology for the design and verification of globally asynchronous/locally synchronous architectures," Tech. Rep. NASA/CR-2005-213912, 2005.

[13] O. Ferrante, L. Benvenuti, L. Mangeruca, C. Sofronis, and A. Ferrari, "Parallel NuSMV: a NuSMV extension for the verification of complex embedded systems," in *Proc. of the International Conference on Computer Safety, Reliability, and Security*, Magdeburg, Germany, 2012.