OCRA: A Tool for Checking the Refinement of Temporal Contracts

Alessandro Cimatti, Michele Dorigatti, Stefano Tonetta
{cimatti,mdorigatti,tonettas}@fbk.eu
FBK-irst, Trento, Italy

***Abstract*—Contract-based design enriches a component model with properties structured in pairs of assumptions and guarantees. These properties are expressed in term of the variables at the interface of the components, and specify how a component interacts with its environment: the assumption is a property that must be satisfied by the environment of the component, while the guarantee is a property that the component must satisfy in response. Contract-based design has been recently proposed in many methodologies for taming the complexity of embedded systems. In fact, contract-based design enables stepwise refinement, compositional verification, and reuse of components. However, only few tools exist to support the formal verification underlying these methods.**

OCRA (Othello Contracts Refinement Analysis) is a new tool that provides means for checking the refinement of contracts specified in a linear-time temporal logic. The specification language allows to express discrete as well as metric real-time constraints. The underlying reasoning engine allows checking if the contract refinement is correct. OCRA has been used in different projects and integrated in CASE tools.

I. Introduction

Contract-based design, first conceived for software specification [21] and now also applied to embedded systems [5], [25], [19], [15], [3], [13], [14], [6], structures the component properties into contracts. A contract specifies the properties assumed to be satisfied by the component environment (assumptions), and the properties guaranteed by the component in response (guarantees). There are several points supporting the idea of contract-based reasoning. The first one is that it provides a clean framework for compositional verification of global properties of an existing system: the contracts are used as landmarks for the proof, so that in the end it is possible to obtain the guarantee for the global property out of the proof that each of the components satisfies its contracts, and that the individual contracts entail the global property. The second is that it supports stepwise refinement, so that when a component is decomposed, the corresponding specification is decomposed at the same time. This means for example that the allocation of functions to subcomponents is decided and proved correct at the moment of decomposition, i.e. way before the behavioral descriptions are provided. The third reason is the support of component reuse: the proof of refinement holds for any component implementation satisfying the contracts of the component leaves.

In this paper we present OCRA, a new tool that provides automated support for contract-based design with temporal logics. OCRA relies on the contract framework originally proposed in [13], where assumptions and guarantees are specified as temporal formulas. Checking the correctness of contracts refinement is supported by generating a set proof obligations. These are temporal logic formulas, obtained from assumptions and guarantees, that are valid if and only if the contracts refinement is correct.

A distinguishing feature of the tool is its high degree of expressiveness: the underlying temporal logic, HRELTL [12], is a variant of LTL where formulas represent sets of hybrid traces, mixing discrete- and continuous-time steps, and is therefore amenable to model properties of timed and hybrid systems. When restricted to the propositional fragment, the proof obligations can be proved valid using BDD- or SAT-based model checking techniques for LTL. In the general case of HRELTL, reasoning relies on Satisfiability Modulo Theory (SMT). Since logical entailment for HRELTL is undecidable, bounded model checking techniques are used to find counterexamples to the contract refinement [12], [13].

OCRA has been developed within the European project SafeCer [26], focusing on the compositional certification of embedded systems. The tool is currently in use by the industrial partners of the project, and has also been integrated within a UML-based modeling environment for aerospace [17]. The tool is publicly available [23].

II. Related work and tools

Many related works focus on methods and tools for assume-guarantee reasoning. In many cases, assumptions and guarantees are expressed in temporal logics (including LTL) [1], [20], [24], [14]). The semantics of the assumption-guarantee pair is however different from the one used in contract-based design. The role of an assumption in assume-guarantee reasoning is to define the context in which the component implementation satisfies the guarantee (these are also called "weak" assumptions in contract-based design). Thus, the semantics of the assume-guarantee pair in most works is simply the implication "if the assumption holds, also the guarantee holds". In this context, many works focused on finding the right assumptions to enable the compositional reasoning, given the implementations of the components (see, for example, [18]).

In contract-based design [21], as well as in OCRA, contracts instead must be satisfied by both the environment and the component. If the environment does not satisfy the assumption, the component is not compatible with that architecture. For

this reason, as for interface automata [16], the refinement is not given by a simple trace inclusion, but has a contravariant fashion: guarantees must be strengthened and assumptions must be weakened. However, instead of relying on alternation simulation between the interface automata, we exploit the less expensive contract refinement that can be reduced to multiple checks of trace inclusion.

To the best of our knowledge, OCRA is the first tool that supports the verification of refinement of contracts for component-based systems specified with a temporal logic. The most related tools are the following: MIO [4], which checks the refinement of contracts specified with a modal variant of interface automata; Ticc [2], which checks the compatibility of interface automata; AGREE [14], which uses temporal logics to apply assume-guarantee reasoning on architectural models, but with the "weak" notion of assumptions.

III. Contract-based formal framework

We consider both *discrete* and *hybrid* traces. A discrete trace is a sequence of assignments to the variables in V. In a hybrid trace, continuous variables evolve continuously when time elapses, while discrete variables only change at discrete steps (see [12] for a more formal definition). Given a set V of variables, we denote with Π_V the set of all possible traces over V (either discrete or hybrid depending on the context).

The interface of a component is defined in terms of ports, i.e. the visible variables with which the component interacts with the environment. We denote with V_S such set of variables, which encompass input/output data/event ports. Given a set $\mathcal{S}$ of components, $V_{\mathcal{S}}$ is defined as $V_{\mathcal{S}} = \bigcup_{S \in \mathcal{S}} V_S$.

An *implementation* M of a component S is defined as a transition system such that each run of M corresponds to a trace over V_S, i.e., the language $L(M)$ of M is a subset of Π_{V_S}. An *environment* E of S is defined in the same way and, therefore, the language $L(E)$ is a subset of Π_{V_S}.

A decomposition ρ of a component S defines a set of subcomponents Sub and a mapping γ of ports. The implementation of a decomposed component S consists of the composition of the implementations of its sub-components Sub through the mapping γ. Similarly, the environment of a subcomponent $U \in Sub$ is composed by the environment of S and by the sibling subcomponents $Sub \setminus \{U\}$.

Given a component S, a *contract* for S is a pair $\langle A, G \rangle$ of assertions over V_S representing respectively an *assumption* and a *guarantee* for the component. Every assertion ϕ is associated with the set of traces that satisfy it, denoted $[\![\phi]\!]$. Let $C = \langle A, G \rangle$ be a contract of S. Let M and E be respectively an implementation and an environment of S. We say that M is an implementation satisfying C iff $M \models A \rightarrow G$ (i.e., $L(M) \cap [\![A]\!] \subseteq [\![G]\!]$). We say that E is an environment satisfying C iff $E \models A$ (i.e., $L(E) \subseteq [\![A]\!]$). We say that M is receptive to C iff M is compatible with E (i.e., for any reachable state of the composition, for every transition of E on an input port of C, there exists a matching transition of M). Finally, we say that M realizes C iff $M \models C$ and M is receptive to C.

Since the decomposition of a component S into subcomponents induces a composite implementation of S and composite environment for the subcomponents, it is necessary to prove that the decomposition is correct, i.e., that the composite implementation of S satisfies S's contracts and that the composite environment of each subcomponent U satisfies U's contracts. Contracts are used to prove these facts compositionally and thus independently from the specific implementation of the subcomponents and environment of the composite component.

Given a component S and a decomposition $\rho = \langle Sub, \gamma \rangle$, a contract C of S, and a set of contracts $\mathcal{CS}$ of the subcomponents of S, we say that $\mathcal{CS}$ is a refinement of C, written $\mathcal{CS} \leq_\rho C$, iff 1) the correct implementations of the sub-contracts form a correct implementation of C; and 2) 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'' (see [13] for a formal definition).

The following theorem defines the *proof obligations* for contracts refinement, i.e. a set of assertions that are valid iff the refinement is correct.

Theorem 1 ([13]): Consider a component S, a decomposition $\rho = \langle Sub, \gamma \rangle$, and $\mathcal{CS} = \{\langle A_1, G_1 \rangle, \ldots, \langle A_n, G_n \rangle\}$. $\mathcal{CS} \leq_\rho C$ iff the following conditions hold:

- $(\exists V_{Sub}((\neg A_1 \vee B_1) \wedge \ldots \wedge (\neg A_n \vee B_n) \wedge \gamma)) \rightarrow (\neg A \vee B))$;
- for all $U \in Sub$, $(\exists V_S, \exists V_{Sub \setminus \{U\}}(A \wedge \bigwedge_{1 \leq j \leq n, j \neq i} (\neg A_j \vee B_j) \wedge \gamma)) \rightarrow A_i$.

IV. OCRA: functionalities and design

The main functionality of OCRA is the verification of the contract refinement. OCRA checks if a given contract refinement is correct generating the corresponding set of proof obligations and checking their validity. OCRA takes as input a textual description of the components interfaces and their decomposition into sub-components, the components' contracts and their refinement with the contracts of the subcomponents (refer to the documentation of the tool in [23] for a formal definition of syntax and semanitics). The OCRA input file, also called OSS (OCRA System Specification), describes a tree of components (given by the decomposition into subcomponents), which represents the system architecture.

The contracts are specified in Othello [11], a human-readable language which can be mapped to temporal formulas in the HRELTL [12], and thus represent sets of hybrid traces. The contract refinement is however independent from the nature of the traces and OCRA provides an option to interpret the contracts over discrete-time traces restricting the input to forbid continuous ports and allow LTL contracts only.

OCRA can be used via an interactive shell or via scripts, and provides the following six main commands:

- OCRA_CHECK_REFINEMENT, which checks the contract refinement of the OSS. Traversing the system architecture starting from the system root component, for each refined contract, it generates the proof obligations, and checks if they are all valid. If the check finds that the refinement is not correct, OCRA provides one or more counterexamples (one for each invalid proof obligation).

- OCRA_CHECK_IMPLEMENTATION, which, given an OSS, the name of a component in the OSS, and an SMV file representing an implementation of the component, verifies if the SMV model satisfies the contracts of the component defined in the OSS. This is restricted to discrete-time LTL contracts. Given a finite state machine modeled in the SMV language, it verifies if the machine satisfies the contracts defined in the OSS.
- OCRA_CHECK_RECEPTIVENESS, which takes the same input of OCRA_CHECK_IMPLEMENTATION, but verifies if the SMV model is receptive to the contracts of the specified component. Together with OCRA_CHECK_IMPLEMENTATION, it checks if the SMV model realizes the component contracts.
- OCRA_PRINT_SYSTEM_IMPLEMENTATION, which, given an SMV model for each basic component and the OSS, builds the corresponding SMV model of the system. This can be used for a monolithic verification to be compared with the compositional verification provided by OCRA.
- OCRA_CHECK_CONSISTENCY, which checks the consistency of each assertion (assumption or guarantee of a contract) of the OSS. This command provides a helpful form of contract validation, that allows to trap some errors in the specification of contracts.
- OCRA_CHECK_SYNTAX, which performs syntactic and type checking on an input OSS file.

In order to prove the validity of the proof obligations deriving from contract refinement, OCRA interacts with NuSMV3 [22], a model checker built on top of NuSMV2 [9] that is able to deal with various forms of temporal logics. NUSMV3 provides the functionality to either prove that the formulas are valid, or to find counterexamples, which can be inspected by the user in order to find the bugs in the contract refinement. When the contracts are written in the standard discrete-time LTL, to prove or disprove the validity of the proof obligations the BDD-based engine is used. In the general case, instead, NuSMV3 relies on the satisfiability procedure described in [12]. This procedure is based on the MathSAT SMT solver [10] and, albeit incomplete, it can be very useful to find errors in the refinement [13].

The tool is written in C, and consists of about 9000 lines of code. About 1000 lines of code are dedicated to parsing; the core of the code provides the functions to manipulate the data structures, to perform some rewriting of formulas, the generation of the proof obligations, and the interaction with NuSMV3. Similarly to NuSMV2, OCRA has an option to run the C preprocessor on the input file, which allows to include other files and to define macros.

V. Contract-based design with OCRA

The advantage of contract-based design is the possibility to perform stepwise refinement, compositional verification, and reuse of components. OCRA supports concretely these techniques with the above commands.

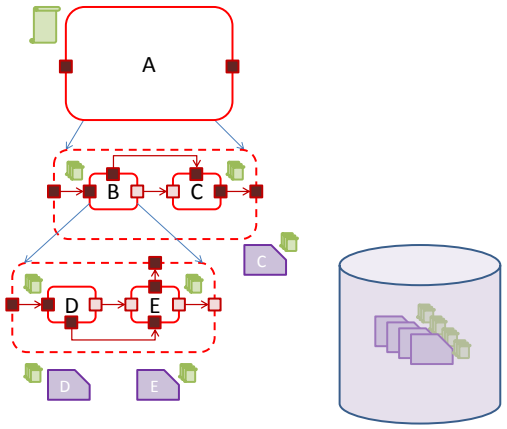

Fig. 1. Contract-based design example: the component A is decomposed into B and C; B is decomposed in D and E.

Figure 1 shows the idea of design flow that OCRA supports in terms of verification. Components are decomposed into collections of other interconnected components. For example, in Figure 1, component A is decomposed into the interconnection of B and C; B in turn is decomposed in the interconnection of D and E. The decomposition also defines how the ports of the component are delegated to ports of the sub-components. For example, the "upper" port of B is mapped onto the "upper" port of E. Each component in the hierarchy is associated with a set of contracts (the scroll shapes in the figure), specifying the acceptable behaviors for the component and its environment. Contracts can be refined, following the decomposition of components. For example, a contract for B may be refined by contracts for D and E. The decomposition of components can be increased step by step, adding every time the corresponding contracts refinement; OCRA can verify this refinement with the command OCRA_CHECK_REFINEMENT; this means that the architectural decomposition is proved correct much before the behavioral descriptions are provided.

In case of propositional LTL contracts, an SMV file can describe the behavior of components. In Figure 1, if the implementations (corner-less rectangles) are proved to satisfy the contracts of the leaf components (i.e. C, D, E), then the resulting system satisfies the top-level contract for A. OCRA can verify compositionally the global property by verifying that the refinement is correct (OCRA_CHECK_REFINEMENT) and that the behavioral descriptions satisfy the corresponding components' contracts with OCRA_CHECK_IMPLEMENTATION (see [13]). Similarly, it can verify that the system is receptive to its contracts compositionally by verifying that the refinement is correct (OCRA_CHECK_REFINEMENT) and that the behavioral descriptions are receptive to their corresponding components' contracts with OCRA_CHECK_IMPLEMENTATION.

Within this framework, reuse can be supported by maintaining a library of components together with their implementations. Instead of checking whether the implementation satisfies the leaf component contract (e.g. as a model checking problem), it may be easier to check whether it is refined by

the contracts stored in the library, that the implementation is known to satisfy. The library can be populated after running OCRA_CHECK_IMPLEMENTATION to check that it satisfies the contract and OCRA_CHECK_RECEPTIVENESS to check that it will be compatible with any context satisfying the assumption. In order to reuse a component from an existing library, the command OCRA_CHECK_REFINEMENT can be used to check that the refinement (and thus the reuse) is correct.

VI. Practical experience

OCRA has being developed within the ARTEMIS SafeCer project [26] and is publicly available at [23]. The CHESS tool [8] has been extended to specify Othello contracts on SysML system-level components or on UML software-level components and to interact with OCRA to check the contracts refinement. A similar integration is under development for the AutoFocus tool [7].

Within the SafeCer project, the tool will be used in a number of demonstrators such as an avionic use case on the airplane distance measuring equipment and a railway use case on the train control and monitoring system of doors. The tool has also been used in two case studies developed within the FoReVer project [17]. The first case study focused on the contract refinement of the requirements of the Fault Detection Isolation and Recovery (FDIR) component of a small virtual satellite system called EagleEye. The second case study was developed by an industrial partner who evaluated the FoReVer tool set (including CHESS) and methodology focusing on the Guidance Navigation and Control (GNC) system of a satellite. The feedback on OCRA was very positive, especially in terms of what can be expressed, and the time required for the refinement checking. It was also useful to improve usability issues such as the readability of the generated traces.

The most complete case study is taken from [15]. It describes a Wheel Braking System (WBS), which takes care of translating the brake signals of the braking pedals into physical brake of the wheel. For the same benchmark, we provided both a real-time and a propositional version. We also specified an SMV model as implementation of each basic component.

VII. Conclusions and future works

This paper presented OCRA, a new tool that supports the verification of contracts refinement for embedded systems. To the best of our knowledge, this the first tool in this context to support the contracts specified in a temporal logic. OCRA is publicly available and will be maintained at least until 2015. It is already integrated with CASE tools (see, e.g., [17] for the integration with CHESS). It has being used by industrial partners within the SafeCer and FoReVer projects. Interesting benchmarks have been already formalized and verified with OCRA. The tool showed to nicely scale with the number of decompositions (since the checks are local to each decomposition).

We are currently working to improve the support in different domain-specific aspects such as asynchronous, real-time, safety, and security contracts. In particular, we are integrating OCRA with the fault-tree analysis provided by NuSMV3.

Acknowledgment

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

References

[1] M. Abadi and L. Lamport. Conjoining Specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–534, 1995.
[2] B.T. Adler, L. de Alfaro, L.D. da Silva, M. Faella, A. Legay, V. Raman, and P. Roy. Ticc: A Tool for Interface Compatibility and Composition. In *CAV*, pages 59–62, 2006.
[3] 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.
[4] S.S. Bauer, P. Mayer, and A. Legay. MIO Workbench: A Tool for Compositional Design with Modal Input/Output Interfaces. In *ATVA*, pages 418–421, 2011.
[5] 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.
[6] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K.G. Larsen. Contracts for System Design. Technical Report RR-8147, INRIA, November 2012.
[7] M. Broy, F. Huber, and B. Schätz. AutoFocus - Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Inform., Forsch. Entwickl.*, 14(3):121–134, 1999.
[8] A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega. CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In *ASE*, pages 362–365, 2012.
[9] 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.
[10] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*. Springer, 2013.
[11] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Validation of Requirements for Hybrid Systems: a Formal Approach. *ACM Trans. Softw. Eng. Methodol.*, 21(4):22, 2012.
[12] A. Cimatti, M. Roveri, and S. Tonetta. Requirements Validation for Hybrid Systems. In *CAV*, pages 188–203, 2009.
[13] A. Cimatti and S. Tonetta. A Property-Based Proof System for Contract-Based Design. In *SEAA*, 2012.
[14] D.D. Cofer, A. Gacek, S.P. Miller, M.W. Whalen, B. LaValley, and L. Sha. Compositional Verification of Architectural Models. In *NFM*, pages 126–140, 2012.
[15] 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.
[16] L. de Alfaro and T.A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.
[17] FoReVer project. https://es.fbk.eu/projects/forever/.
[18] D. Giannakopoulou and C.S. Pasareanu. Learning-Based Assume-Guarantee Verification. In *SPIN*, pages 282–287, 2005.
[19] S. Graf, R. Passerone, and S. Quinton. Contract-Based Reasoning for Component Systems with Complex Interactions . In *TIMOBD'11*, 2011.
[20] K.L. McMillan. Circular Compositional Reasoning about Liveness. In *CHARME*, pages 342–345, 1999.
[21] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
[22] NuSMV3 website. https://es.fbk.eu/tools/nusmv3/.
[23] OCRA website. http://es.fbk.eu/tools/ocra.
[24] C.S. Pasareanu, M.B. Dwyer, and M. Huth. Assume-Guarantee Model Checking of Software: A Comparative Case Study. In *SPIN*, pages 168–183, 1999.
[25] S. Quinton and S. Graf. Contract-Based Verification of Hierarchical Systems of Components. In *SEFM*, pages 377–381, 2008.
[26] SafeCer project. http://www.safecer.eu.