EVA: a Tool for the Compositional Verification of AUTOSAR Models

Alessandro Cimatti[1], Luca Cristoforetti[1], Alberto Griggio[1], Stefano Tonetta[1], Sara Corfini[2](✉), Marco Di Natale[2,4], and Florian Barrau[3]

[1] Fondazione Bruno Kessler, Trento, Italy
[2] Huawei Pisa Research Center, Pisa, Italy
s.corfini@huawei.com
[3] Huawei Grenoble Research Center, Grenoble, France
[4] Scuola Superiore Sant'Anna, Pisa, Italy

Abstract. We present EVA, a framework for the integration of modern verification tools in the context of AUTOSAR, a widely-used open standard for the development of automotive software systems. Our framework enables the automatic end-to-end verification of system-level properties using a compositional approach. It combines software model checking techniques for the verification of software components at the code level with a contract-based analysis for verifying their correct composition. In this paper, we present the tool through its application on a representative automotive case study, discussing the main functionalities provided and the results obtained.

1 Introduction

AUTOSAR [1] is a worldwide consortium of car manufacturers and component or service providers in the automotive domain, with the main goal of providing a standardized software architecture for the development and execution of software components. One of the fundamental challenges in designing software for the AUTOSAR platform is ensuring safety. To this end, the application of formal methods – and in particular automatic (or semi-automatic) techniques based on model checking and theorem proving – is receiving significant interest as a complement to more traditional V&V techniques. In this paper we present EVA, a framework for the integration of modern verification tools in the context of AUTOSAR. EVA adopts a model-based compositional verification that founds on the contract-based methodology in [8]. The tool allows the automatic end-to-end verification of system-level properties, and combines software model checking techniques for the verification of software components at the code level with a contract-based analysis for verifying their correct composition. EVA also implements all the features that are required for usability in a typical industrial context, including a front-end integrated in a standard AUTOSAR development environment [2] with a user-friendly (formal) property editor, the automatic generation of code stubs and other views and forms to help the user manage verification in an AUTOSAR environment.

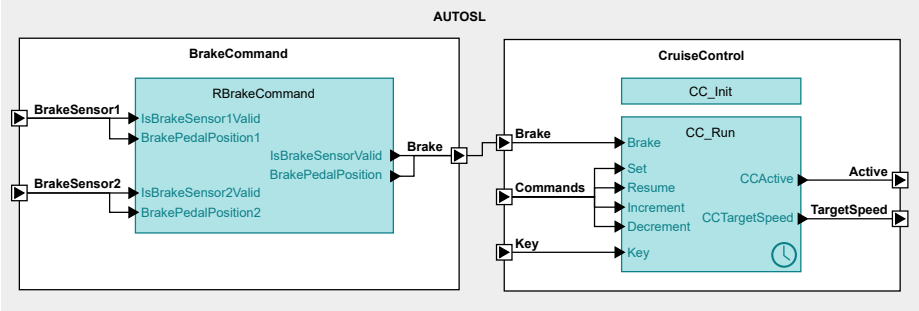

Fig. 1. BrakeCommand and CruiseControl components.

We present EVA through its application on a representative case study, which describes a simplified active safety automotive system containing some of the typical safety functions available in the modern vehicles (such as lane departure warning, cruise control and a fault-tolerant brake pedal system). The example is meant to show the potential of the tool as a driver for a more widespread adoption of formal methods and contract-based verification in the industrial automotive context. Specifically, we introduce the case study in §2 and we describe the typical verification workflow followed by a user of EVA in §3. Finally, in §4 we discuss the main verification results obtained.

2 A Case Study for Verification in AUTOSAR

AUTOSAR defines the reference architecture for the development of automotive systems and provides the language (meta-model) for describing their architectural models. An AUTOSAR application consists of a hierarchy of components connected through ports. *Provide* ports represent output ports and *require* ports correspond to the input ports. Connectors represent data flow from one port to another. An AUTOSAR port can be classified as sender-receiver or client-server and sender-receiver communications can be queued or non-queued (i.e., with no buffering and the receiver always accesses the last sent data). In this paper we assume that all ports are sender-receiver and non-queued.

An *atomic* software component consists of a set of runnables. A *runnable* is a sequence of operations started by the Run-Time Environment (RTE). The runnable is configured so that is triggered by an event that can be timing, data sent or received, operation invoked, return of a server call, mode switching or external events. A special *init* event is used for runnables that are executed when the RTE starts and initializes the software components.

We illustrate the basic notions above by means of a simple but representative case study, that we shall use to present the main features of EVA. Figure 1 overviews (a section of) the architecture of the sample application. It collects 22 atomic components (including sensors, controllers and actuators) plus one

composite component (AUTOSL) that represents the whole system, and implements some of the typical safety functions available in the modern vehicles such as autonomous emergency braking, lane departure warning, crash preparation and cruise control. We implemented (the runnables of) 9 components, 7 have been coded manually and 2 have been generated from a Simulink model using the Embedded Coder Support Package for AUTOSAR. The other components are considered as stubs because their data come from lower levels (hardware sensors) and we assume that the values they provide are correct.

The case study considers various safety properties, both at the level of the whole system and at the level of the implementation of individual components or runnables. As an example, we describe here two properties, a system-level one and a component-level one, both concerning the behaviour of the cruise controller. Specifically, the cruise controller is expected to react to a brake input by disengaging itself within two execution steps. At the implementation level, the requirement relates the input and output ports of the CruiseControl periodic runnable, stating that whenever the CruiseControl CCActive port is true and the Brake input port is true, then the CCActive output port must become false in at most two steps. At the system level, instead, the same requirement relates the behaviour of the components BrakeCommand and CruiseControl, stating that the cruise control shall be disengaged if the user brakes, even when one of the two brake pedal sensors is faulty.

3 EVA Verification Workflow

EVA integrates the verification engines Kratos2 [6] and OCRA [5] into an analysis AUTOSAR toolchain. The ultimate goal is to automate the verification of formal properties (contracts) on AUTOSAR models. In its default configuration, EVA uses a portfolio of different state-of-the-art SAT- and SMT-based symbolic model checking algorithms (implemented in Kratos2 and OCRA) which include different variants of bit-level IC3 [10,12], IC3 with implicit abstraction [7], bounded model checking [3] and K-induction [11].

The typical workflow of the tool is sketched in Figure 2. At the beginning, the user creates an analysis project providing as input the AUTOSAR configuration of the system. The tool transforms the AUTOSAR configuration into an internal set of analysis models. Since the AUTOSAR standard deals neither with requirements nor with formal properties and their verification, EVA adopts the extended AUTOSAR metamodel defined in [4] to support such concepts.

The user then completes the configuration of the system and provides:

source code: the user imports into the analysis project the source code of the runnables and associates each runnable with its source files.

requirements: the user defines the (informal) properties of the system and their relationships. Specifically, the user can assign a requirement to a component, or to the system (modeled by a composite component) and refine it into other requirements. Considering the following examples of informal

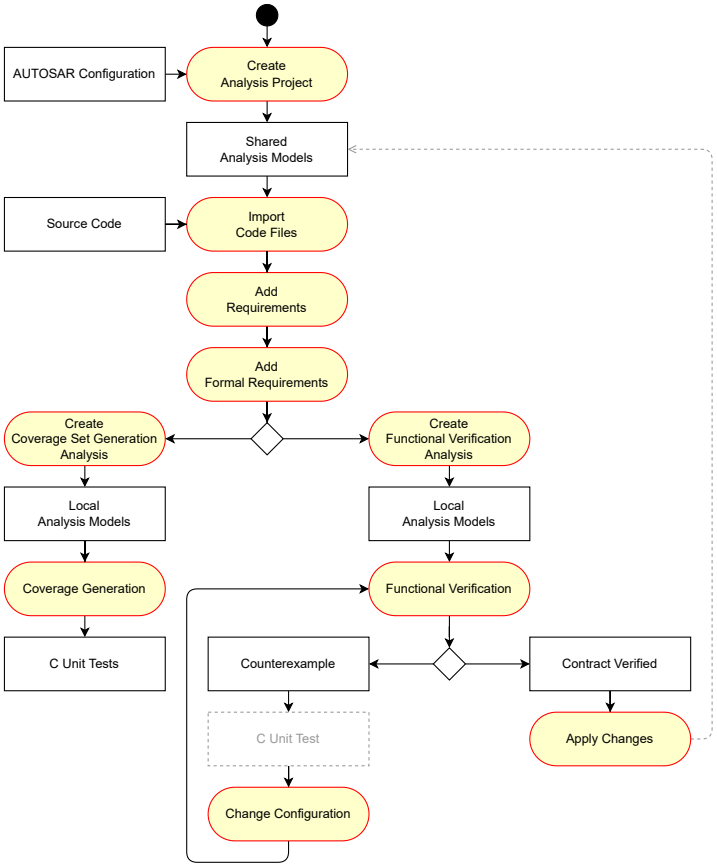

Fig. 2. The analysis workflow.

requirements for the case study of §2:

If the user brakes, the cruise control shall disengage within 2 steps (1)

The signals of the brake pedal sensors shall be merged (2)

Even if at most one brake pedal sensor is faulty if the user brakes, the cruise control shall disengage (3)

(1) and (2) are component-level requirements assigned to CruiseControl and BrakeCommand respectively, while (3) is a system-level requirement assigned to the composite AUTOSL and refined by (1) and (2).

contracts: the user formalizes the requirements into contracts. Precisely, a contract consists of (optional) assumptions (properties that shall be satisfied by the environment) and assertions (properties that the owner of the contract shall satisfy), expressed as formulas in Linear Temporal Logic (LTL) with some metric extensions (interpreted over discrete time). The user can assign

a contract either to a runnable or to a (composite) component.

```
in the future within [2,2]                                          (4)
it shall always_be that
    (CCActive and Brake is_greater_than 0) implies
        in the future within [0,2] (not next(CCActive))
holds_true
```

Contract (4) is the formal representation of requirement (1) and it is assigned to the periodic runnable of the CruiseControl component[5]. It is worth noting that EVA provides a *smart* contract editor that assists the user with context completion, syntax highlighting and error detection. Also, to aid readability of contracts, EVA uses some syntactic sugar to represent temporal operators, such as in the future for F or it shall always_be for G.

The user can create a new functional verification analysis, allowing to perform:

code verification: the user can check whether (the source code of) a runnable satisfies one of the contracts assigned to it. Let us consider again the periodic runnable of the CruiseControl component. The user can run code verification to check whether that runnable satisfies its assigned contract (4).

compositional verification: the user can check whether a contract assigned to a (composite) component is correctly refined by the contracts of the sub-components. Intuitively, the user can run compositional verification to check whether the system-level contract derived from requirement (3) and assigned to the composite AUTOSL, is refined by the contracts derived from requirements (1) and (2) and assigned to the runnables of components CruiseControl and BrakeCommand.

The result of both analyses can be that the contract is verified or violated. In case of contract violation, EVA returns a counterexample (and the corresponding test case, if the performed analysis is code verification). The user can fix the code or change the system configuration (refine requirements or scheduling runnables) and then execute the analysis again. The user can optionally apply local changes to the shared analysis models (typically after a contract has been verified).

In addition to the main features above, two further analyses are provided:

contract validation: the user can verify the consistency (and absence of logical contradictions) of the contracts of a component and of its sub-components.

coverage set generation: it combines model checking and random simulation to automatically generate unit tests (using the CUnit [9] framework) trying to cover all the branches of the `C` code of a given runnable.

4 Experimental Evaluation

In order to evaluate the effectiveness and performance of EVA, we applied it to the verification of all the 43 requirements (10 system-level, 33 component-level)

[5] We omit the contracts derived from (2) and (3) for lack of space (their formalization shall be included in the artifact accompanying this submission).

of the case-study application described in §2. Due to lack of space, we cannot report the results in detail and we shall limit our analysis to some qualitative considerations about the overall performance of EVA and the usefulness of the produced outputs. Full details on the obtained results will be included in the submitted artifact.

Performance considerations. We verified all the requirements on a PC running Ubuntu Linux 20.04, with a 2.6 GHz Intel Core I7-66000U CPU and 20 Gb of RAM. EVA was able to successfully perform 42 out of 43 verification tasks within the timeout (set to 1 hour), requiring less than one second in nearly half of the cases for component-level properties, and requiring less than one minute for all the remaining component-level tasks except one. For such problems, the main bottlenecks identified during the case study involved the use of complex floating-point operations, which are still handled inefficiently by the verification backend. Also the verification of the 10 system-level properties could be completed relatively efficiently, with EVA requiring less than one minute in 7 cases, and approximately 30 minutes for the hardest one. In this case, the main factor affecting performance (besides the expected ones such as the number of involved contracts and their complexity and length) are the constraints on the composition of components defined in the input model. In particular, performance is affected significantly in cases in which the contract under analysis involves periodic components with very different activation periods. The presence of periods that range from few milliseconds to seconds poses a conceptual/theoretical challenge because the reasoner must explore a large number of small steps of the more frequent tasks for each step of the slow ones. Optimizations targeting this issue are left as research directions for future works.

Issues discovered. During verification, several counterexamples have been discovered. Most of them turned out to be due to incorrect formalizations of requirements or missing environment assumptions, which could be easily fixed by examining the produced counterexamples. The analyses however revealed also a number of real bugs in the implementations of some of the software components as well as two issues due to wrong scheduling of components. The first was caused by a mismatch between the Simulink description of the CruiseControl periodic runnable and its C implementation in the AUTOSAR application. Specifically, the mismatch was due to different assumptions about the rate of execution of the step of the cruise control with respect to the rate of the change of the inputs, which caused the input values to be read only at even steps of the cruise controller. The second issue regarded the scheduling of the BrakeCommand runnable, which was set to be executed only upon changes in the input pedal positions. A counterexample in the contract refinement showed that the validity of these input signals could change value without the BrakeCommand running so that the pedal position was not propagated to the CruiseControl. The model was fixed by adding a trigger of the BrakeCommand also associated to the valid signal of the pedal positions. In both cases, the bugs could be fixed by analyzing the counterexamples generated by EVA.

5 Data Availability Statement

The artifact described in the paper is not publicly available due to internal policy. Any requests can be directed to the corresponding author.

References

1. https://www.autosar.org
2. Artop: The AUTOSAR Tool Platform, http://www.artop.org
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) 5^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 1579, pp. 193–207. Springer (1999)
4. Cimatti, A., Corfini, S., Cristoforetti, L., Di Natale, M., Griggio, A., Puri, S., Tonetta, S.: A Comprehensive Framework for the Analysis of Automotive Systems. In: Syriani, E., Sahraoui, H.A., Bencomo, N., Wimmer, M. (eds.) ACM/IEEE 25^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS). pp. 379–389. ACM (2022)
5. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: A Tool for Checking the Refinement of Temporal Contracts. In: Denney, E., Bultan, T., Zeller, A. (eds.) 28^{th} IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 702–705. IEEE (2013)
6. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos - A Software Model Checker for SystemC. In: Gopalakrishnan, G., Qadeer, S. (eds.) 23^{rd} International Conference on Computer Aided Verification (CAV). LNCS, vol. 6806, pp. 310–316. Springer (2011)
7. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state Invariant Checking with IC3 and Predicate Abstraction. Formal Methods in System Design **49**(3), 190–218 (2016)
8. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Science of Computer Programming **97**, 333–348 (2015)
9. CUnit: A Unit Testing Framework for C, cunit.sourceforge.net
10. Griggio, A., Roveri, M.: Comparing Different Variants of the ic3 Algorithm for Hardware Model Checking. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **35**(6), 1026–1039 (2016)
11. Sheeran, M., Singh, S., Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver. In: Hunt, W.A., Johnson, S.D. (eds.) 3^{rd} International Conference on Formal Methods in Computer-Aided Design (FMCAD). LNCS, vol. 1954, pp. 108–125. Springer (2000)
12. Vizel, Y., Gurfinkel, A.: Interpolating Property Directed Reachability. In: Biere, A., Bloem, R. (eds.) 26^{th} International Conference on Computer Aided Verification (CAV). LNCS, vol. 8559, pp. 260–276. Springer (2014)