Counterexample Interpretation for Contract-Based Design

Chapter · September 2020

DOI: 10.1007/978-3-030-58920-2_7

4 authors:

Arut Prakash Kaleeswaran
Bosch

Arne Nordmann
Bosch

Thomas Vogel
Humboldt-Universität zu Berlin

Lars Grunske
Humboldt-Universität zu Berlin

Counterexample Interpretation for Contract-Based Design

Arut Prakash Kaleeswaran[1,2], Arne Nordmann[1], Thomas Vogel[2], and Lars Grunske[2]

[1] Bosch Corporate Sector Research, 71272 Renningen, Germany
FirstSecond.Lastname@de.bosch.com
[2] Humboldt-Universität zu Berlin, Berlin, Germany
First.Lastname@informatik.hu-berlin.de

Abstract Contract-based design (CBD) is an emerging paradigm for complex systems, specifying the input-output behavior of a component by defining what the component guarantees, provided its environment satisfies the given assumptions. Under certain circumstances, it is possible to verify the decomposition of contracts to conclude the correctness of the top-level system requirements. Verification is performed by using model checkers. If the decomposition of the contract is found to be incorrect, a model checker generates a counterexample. However, the challenging task is to understand the counterexample, which usually is lengthy, cryptic, and verbose. In this paper, we propose an approach to derive an understandable error explanation for counterexamples in CBD. In addition, we highlight the erroneous variables and erroneous states in the counterexample, which reduces the effort to identify errors. Therefore, our approach supports error comprehension of the original counterexample. Our approach is evaluated based on two industrial use cases, the Bosch Electronic Power Steering (EPS) and a redundant sensor system.

Keywords: Contract-based design · counterexample comprehension

1 Introduction

When software-intensive systems are used in safety-critical domains such as automotive and avionics, their malfunction might lead to severe damages or even loss of lives. Thus, safety is of paramount importance and systems have to be developed according to safety standards such as IEC 61508 or ISO 26262. These standards require safety methods such as Failure Mode and Effects Analysis (FMEA), Fault Tree Analysis (FTA), or Hazard and Operability (HAZOP) that analyze the safety of a system based on a model of the system [14,15,30,28].

In previous work, we presented Model-Based Safety Analysis (MBSA) methods for FMEA [24], FTA [25], and HAZOP [18] by combining and linking safety analysis with Model-Based System Development (MBSD). These methods help in analyzing safety goals: If safety goals are violated, engineers derive safety requirements to satisfy these goals. However, the analysis to identify the faults is performed manually by engineers, which should be ideally automated. In recent years, formal methods have been developed to analyze and verify complex systems [33], and as such, for instance, formal verification would have revealed the exposed defects in Ariane-5, Mars Pathfinder, Intel's Pentium II processor, and the Therac-25 therapy radiation machine [1].

One example of a formal method is *model checking* [1,11], where a *model checker* verifies whether a provided *property/specification* φ is satisfied by a given state-based *system model* K, that is, $K \vDash \varphi$. Otherwise, if φ is not satisfied by K, a model checker generates a *counterexample* describing an execution path in K that leads from the initial system state to a state that violates φ [1], whereas each state consists of variables (atomic propositions) with their values. Engineers can use the counterexample to find the fault in K that causes the violation of φ. According to Clarke [9, p. 3], "It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use model checking just for this feature."

Nevertheless, a counterexample is only the symptom of a fault, and understanding a counterexample to actually identify a fault in the system model is complicated, error-prone, and time-consuming because of the following problems: (**P1**) a counterexample is cryptic and can be lengthy [26], (**P2**) not all the states in a counterexample are relevant to an error [4], (**P3**) not all the variables in a state are related to the violation [4], (**P4**) the debugging of a system model using a counterexample is performed manually [2,26], and (**P5**) a counterexample does not explicitly highlight the source of the error that is hidden in the model [2]. Thus, an automated method for explaining counterexamples that assists engineers in localizing faults in their models is highly desirable [21].

Contract-based design (CBD) is an emerging paradigm for designing complex systems using components and contracts [3]. The components are defined by a component diagram, and each of them is associated with a contract that precisely specifies the expected behavior of the component by assumptions and guarantees [8]. If a component is refined to sub-components, its contract is also refined and assigned to its sub-components. Thus, all of the sub-components should satisfy the expected behavior of the parent component. This corresponds to the correctness and consistency of the refined contracts and can be verified by a model checker, which is known as a *refinement check*.

In this paper, we present an approach that supports engineers in comprehending a counterexample and locating the fault in a CBD that does not pass the refinement check. The contribution of our approach is the automated identification of (*i*) the *erroneous specification* as those parts of a CBD specification that cause the violation of the refinement, (*ii*) the *erroneous states*, that is, those states in a counterexample that are relevant to an error, (*iii*) the *erroneous variables* as those variables in a counterexample that are related to the violation, and (*iv*) the *erroneous component* that causes the violation and particularly, whether its assumptions or guarantees are erroneous. As the *erroneous states* distinguish relevant and irrelevant states in a counterexample, the length of a counterexample to be investigated by engineers is reduced. The *erroneous specification* and *variables* support engineers in relating the CBD specification with the counterexample by focusing on the erroneous parts. Finally, the *erroneous component* helps engineers with identifying the hidden error in the component model of a CBD. Thus, our approach highlights the erroneous states, specification, variables, and component rather than showing the raw specification and counterexample to engineers, which aims for reducing the complexity, error-proneness, and costs of debugging, and supporting error comprehension.

For this purpose, our approach takes a CBD, translates the component model to a system model (K), specifically a Kripke Structure (KS), and the contracts to a refinement check formula defined in Linear Temporal Logic (LTL) [31], and verifies the refinement of contracts using the NuSMV [7] model checker. If the refinement is violated, the resulting counterexample and the violated LTL specification is processed. First, to identify the *erroneous specification*, we extend the work of Narizzano et al. [27]. It identifies inconsistent LTL sub-specifications from the whole violated LTL specification. Second, to identify the *erroneous states* in the counterexample, for which we adapt the idea of Barbon et al. [2], where the full system behavior of a Labeled Transition System (LTS) is simulated. Adapting the work of Barbon et al. for KS, we face two challenges: (**Ch1**) We specify the behavior of a system by contracts (LTL specifications) in contrast to an LTS. Thus, we cannot simulate the behavior in terms of contracts. (**Ch2**) To simulate the full system behavior, all of the possible initial states needs to be collected, which is not possible due to limitations of NuSMV. To overcome **Ch1**, once we find the *erroneous specification*, we remove the erroneous specification from the complete LTL specification and translate the remaining specification to the system model KS that can be simulated by NuSMV. To address **Ch2**, we consider each state from the counterexample as an initial state and simulate the behavior for all such states. Further, by comparing the counterexample trace with the simulation trace, the *erroneous states* in the counterexample are identified. Third, to identify the *erroneous variable* in the counterexample, the variables that belong to *erroneous specification* are extracted and highlighted in the counterexample. Fourth, to identify the component to which the *erroneous specification* belongs and whether the assumptions or guarantees of the component's contract are erroneous, we use information from the refinement check and the *erroneous variables*. We evaluate the proposed approach by using two industrial use cases. To the best of our knowledge, the presented approach is the first one to identify the *erroneous specification*, *erroneous states* and *erroneous variables* from counterexamples in CBD.

2 Contract-Based Design and Motivating Example

Contract-Based Design (CBD) exploits the contracts for compositional reasoning, step-wise refinement, and reuse of components that are already pre-designed or designed independently [8]. CBD consists of a component model and contracts; particularly, a contract C is a pair of assumptions α and guarantees β, thus $C = (\alpha, \beta)$. According to Kaiser et al. [17, p. 70], "the assumption specifies

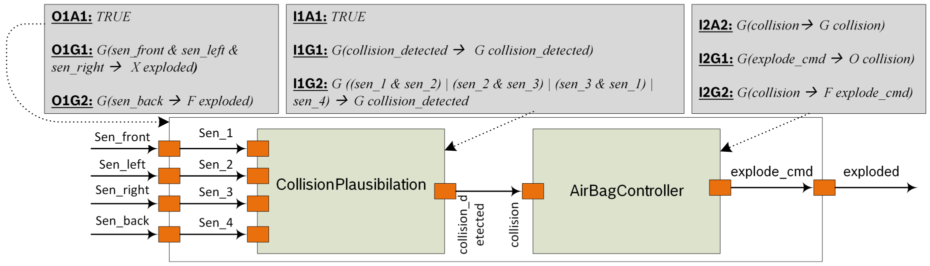

Figure 1: CBD with its component model and contracts of the airbag system.

how the context of the component, i.e., the environment from the point of view of the component, should behave. Only if the assumption holds, then the component will behave as guaranteed". To illustrate CBD and our approach, we model and use the airbag system from Ratiu et al. [32] as a running example. The corresponding CBD with its component model and contracts are shown in Figure 1. The input of the airbag system is the detection signal taken from four sensors: *sen_front*, *sen_left*, *sen_right*, and *sen_back*, and the output signal *exploded* activates the airbag system. The system consists of two sub-components: *CollisionPlausibilation* and *AirBagController*. *CollisionPlausibilation* detects the collision or crash from the input sensor signal. The *AirBagController* processes the output from *CollisionPlausibilation* and controls the activation of the airbag.

In Figure 1, a contract of a component is labeled by $O_N A_N$ and $I_N G_N$, whereas O and I indicate whether the component is an outer or an inner one, A means assumption (α), G means guarantee (β), and the number N is used for enumeration. The contract of the top-level component (in this case, the system) has two guarantees. **O1G1**: whenever the detection signal is given by all of the sensors *sen_front*, *sen_left*, and *sen_right*, then the *exploded* signal will be activated in the next time step. **O1G2**: whenever the detection signal is given by *sen_back*, then the *exploded* signal will be activated in a future time step.

Considering the contracts of a parent/outer component $C_T = (\alpha, \beta)$ and a sub-/inner component $C_S = (\alpha_S, \beta_S)$, the refinement relations between C_T and C_S are $\alpha \subseteq \alpha_S$ and $\beta_S \subseteq \beta$ [17]. In the airbag system, α is $O_N A_N$, α_S is $I_N A_N$, β is $O_N G_N$, and β_S is $I_N G_N$. Cimatti and Tonetta [8, Section 4.D] describe the formulae to verify the correctness of a refinement, which is also used with minor modifications in [32]. These *refinement check formulae* R are shown in Equations (1) and (2), in which the *wire* construct connects the input and output ports of the components. R is restricted to the standard propositional LTL in order to be able to perform verification by standard model checking [8].

$$\underbrace{(\alpha \wedge \bigwedge_{1 \leq j \leq n, j \neq i} (\alpha_{\mathrm{j}} \implies \beta_{\mathrm{j}})) \wedge wire}_{antecedent} \implies \underbrace{\alpha_{\mathrm{i}}}_{consequent} \tag{1}$$

$$\underbrace{\alpha \wedge ((\alpha_1 \implies \beta_1) \wedge ... \wedge (\alpha_{\mathrm{n}} \implies \beta_{\mathrm{n}})) \wedge wire}_{antecedent} \implies \underbrace{\beta}_{consequent} \tag{2}$$

Equation (1) is to verify whether the assumption of each sub-component (α_{i}) holds true whenever the contracts of all of the other sub-components ($\alpha_{\mathrm{j}} \implies \beta_{\mathrm{j}}$) and the assumption of the parent component (α) holds. Equation (2) is to verify whether the guarantee of the parent component (β) holds true whenever the contracts of all n sub-components ($\alpha_{\mathrm{n}} \implies \beta_{\mathrm{n}}$) and the assumption of parent component (α) holds. In the case that α_{i} or β fails to hold, R is violated and the verification by a model checker returns a counterexample. Applying R to the airbag systems results in the following formulae, whereas Equations 3a and 3b instantiate Equation (1), and Equation 4 instantiates Equation 2.

$$(O1A1 \wedge (I2A1 \implies I2G1 \wedge I2G2) \wedge wire) \implies I1A1 \tag{3a}$$

$$(O1A1 \wedge (I1A1 \implies I1G1 \wedge I1G2) \wedge wire) \implies I2A1 \tag{3b}$$

$$\begin{aligned} &(O1A1 \wedge (I1A1 \implies I1G1 \wedge I1G2) \wedge \\ &(I2A1 \implies I2G1 \wedge I2G2) \wedge wire) \implies (O1G1 \wedge O1G2) \end{aligned} \tag{4}$$

Equation (3a) verifies whether the assumption of *CollisionPlausibilation* ($I1A1$) holds true, whenever the contract of the *AirBagController* and the assumption of the system ($O1A1$) hold. Similarly, Equation (3b) verifies the assumption of the *AirBagController* ($I2A1$). Finally, Equation (4) verifies whether the system guarantees ($O1G1$ and $O1G2$) hold true, whenever the contracts of *CollisionPlausibilation* and *AirBagController* and the system's assumption ($O1A1$) hold.

3 Approach

To reduce the complexity, error-proneness, and costs of debugging, and thus to support error comprehension when checking a refinement in a CBD, our approach highlights relevant parts of the CBD and counterexample rather than showing the raw CBD and counterexample. For this purpose, our approach processes a CBD and a counterexample in several steps as shown in Figure 2.

Step1 comprises the modeling of a CBD and the verification of the refinement formulae R for this CBD (cf. Section 2) using the NuSMV model checker. If any formula of R is violated, the refinement check fails so that NuSMV generates a counterexample. This indicates that there must be a fault in the CBD. Using the CBD encoded in a NuSMV file (FSMV), the violated formula/specification R, and the counterexample, *Step2* finds the erroneous (sub-)specification E_{spec} from R, which consequently identifies as well the correct (sub-)specification C_{spec}. This step is based on the work of Narizzano et al. [27]. To enable simulation of the system behavior in terms of the LTL-based C_{spec}, we translate C_{spec} to a Kripke structure and combine this structure with the Kripke structure of FSMV, which results in the NuSVM model SMV_{new}. This step adapts the work of Barbon et al. [2] by addressing challenge Ch1 (cf. Section 1). Simulating the SMV_{new} model, *Step4* generates traces of correct behavior, which are then compared to the counterexample trace. This comparison identifies the *erroneous states* E_{state} in the counterexample. This step is based on the work of Barbon et al. [2] by addressing challenge Ch2 (cf. Section 1). From the erroneous specification E_{spec}, *Step5* extracts the variables as they are related to the violation of R, thus being the *erroneous variables* E_{var}. Based on E_{spec} and E_{var}, *Step6* identifies the

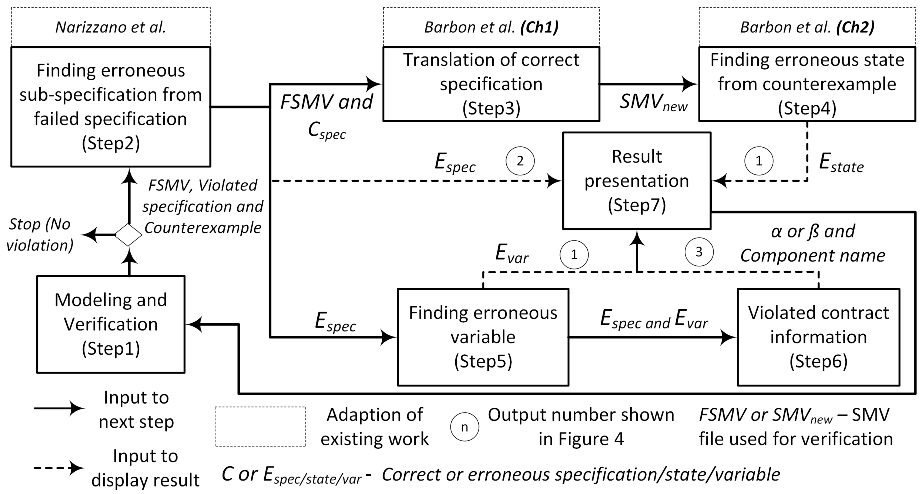

Figure 2: Overview of the approach.

component with its contract that causes the violation, and particularly whether the assumptions or guarantees of the contract are erroneous. Finally, *Step7* collects all of the results from the other steps to highlight the erroneous specification E_{spec}, states E_{state}, variables E_{var}, component in the CBD and counterexample, thus to display the result of our approach. The highlighted elements support an engineer in understanding the counterexample and finding the fault in the CBD. Then, the engineer can correct the CBD by remodeling the CBD and reverifying the refinement (Step1). In the following, we discuss each step in detail.

3.1 Step 1: Modeling and verification

This step comprises the modeling of a CBD and verification of the refinement for the CBD. We use FASTEN [32] to specify the component model and contracts. FASTEN is an extensible platform for modeling and model checking safety-critical systems. Specifically, it translates the component model to a system model (K) being a Kripke structure (KS), and the contracts to an LTL specification (φ) based on the refinement formulae R (see the generic Equations (1) and (2)). The output of the translation is an SMV file ($FSMV$), which is then used by the NuSMV model checker for verification, that is, to perform the refinement check in terms of the correctness of R. If one of the formulae in R is violated, NuSMV generates a *concrete counterexample* (C_C).

For the airbag system in Figure 1, FASTEN creates the formulae of Equations 3 and 4, which instantiate the generic Equations (1) and (2), respectively. Verifying this specification, a C_C is generated for violating Equation (4) due to the inconsistency between the guarantees $O1GA$ and $I2G2$. This is because the variable *exploded_cmd* in $I2G2$ holds true for a future time step but *exploded* in $O1G1$ holds true immediately for next time step.

Algorithm 1 Algorithm to identify the erroneous specification.

INPUT: R - violated specification and C - concrete counterexample
OUTPUT: E_{spec} - erroneous sub-specification and C_{spec} - correct sub-specification

1: E_{spec}, global list1
2: C_{spec}, global list2
3: **function** FINDINCONSISTENCY(R, C)
4: $\quad (antecedent, consequent) \leftarrow$ SPLIT(R)
5: $\quad$ **if** SIZEOF$(consequent) == 1$ **then**
6: $\quad\quad E_{spec} \leftarrow consequent$
7: $\quad$ **else**
8: $\quad\quad$ **for** $n \leftarrow 0$ to SIZEOF$(consequent)$ -1 **do**
9: $\quad\quad\quad consequent \leftarrow consequent \setminus consequent(n)$
10: $\quad\quad\quad C_{new} \leftarrow$ RUNNUSMV$(antecedent\ +\ "\implies"\ +\ consequent)$
11: $\quad\quad\quad$ **if** $C_{new}.equals(C)$ **then**
12: $\quad\quad\quad\quad C_{spec} \leftarrow C_{spec} \cup consequent(n)$
13: $\quad\quad\quad\quad n \leftarrow n - 1$
14: $\quad\quad\quad$ **else**
15: $\quad\quad\quad\quad E_{spec} \leftarrow E_{spec} \cup consequent(n)$
16: $\quad\quad\quad\quad consequent \leftarrow consequent \cup consequent(n)$

3.2 Step 2: Identifying the Erroneous Specification

Using *FSMV*, C_C, and the violated specification R from *Step 1*, this step identifies the violated (*erroneous specification* E_{spec}) and correct sub-specifications (*correct specification* C_{spec}) from R, whereas a sub-specification can be one of the assumptions α or guarantees β of the contracts. For this purpose, we adapt the work of Narizzano et al. [27], which identifies inconsistency between functional requirements, to contracts as shown in Algorithm 1. The algorithm performs two tasks: *(Task1)* Split the violated R, and *(Task2)* identify C_{spec} and E_{spec} in R. Referring to Equation (5), φ fails whenever the *antecedent* holds $true$ and the *consequent* holds $false$. Applying the same rule to Equations (1) and (2): if one of the R fails, a counterexample is generated for the violated *consequent*. Thus, `line 4` performs *Task1* by splitting R into two lists: *antecedent* and *consequent*.

$$\varphi ::= p \implies q \text{ is false iff } p = true \text{ and } q = false \tag{5}$$

Task2 identifies E_{spec} and C_{spec} in the *consequent* list. This is achieved by removing one consequent at a time, and repeat the verification to see which consequent makes the formula fail. If the removed consequent belongs to E_{spec}, the counterexample generated during the verification will differ from C_C. On the other hand, if the counterexample is same as C_C, it is considered as C_{spec}. In detail, if the size of the *consequent* list is one (`line 5`), then there is only one consequent that is then added to the E_{spec} list. In this case, *Task2* is finished. Otherwise, the iterative approach is performed from `line 8` to `line 16`. In the following, we refer to each sub-specification in the *consequent* list as *consequent(n)*, where n is the number of the sub-specification. In `line 9`, *consequent(n)* is removed from the *consequent* list and verification is performed by running NuSMV in `line 10`. For verification, the *antecedent*, $\implies$, and the *con-*

sequent list are concatenated as a propositional LTL property. The generated counterexample is assigned to C_{new} (`line 10`) and compared with C_C (`line 11`). If $C_{new} \equiv C_C$, *consequent(n)* is added to the C_{spec} list (`line 12`) and n is subtracted by one in `line 13`. Otherwise, when $C_{new} \neq C_C$, *consequent(n)* is added to the E_{spec} list (`line 15`) and *consequent(n)* is added back to the *consequent* list (`line 16`).

For the airbag system, we know from *Step1* that Equation (4) is the violated R. The *antecedent* of this equation is $(O1A1 \wedge (I1A1 \implies I1G1 \wedge I1G2) \wedge (I2A1 \implies I2G1 \wedge I2G2))$ and the *consequent* is $(O1G1 \wedge O1G2)$. The guarantee $O1G2$ is added to C_{spec} as it is consistent with the *antecedent*, while the guarantee $O1G1$ is added to E_{spec} since it is inconsistent with guarantee $I2G2$.

3.3 Step 3: Translating the LTL Specification to a System Model (K)

This step is required to apply the idea by Barbon et al. [2] to identify the erroneous states in the counterexample using simulation of the correct behavior, which is done in the subsequent step. In our case, the behavior of the system is only implicitly modeled in the form of contracts (φ), and not explicitly as a system model (K). Thus, we cannot simulate the behavior directly (cf. challenge Ch1 in Section 1). To overcome this challenge, we translate φ to K.

LTL2SMV [10] is an independent component of NuSMV, which takes an LTL specification as input and produces a corresponding system model in SMV format as output. We take *FSMV*, the *consequent*, and C_{spec} from *Step 2* as input, combine the *consequent* and C_{spec} along with *implies* as the LTL specification $\varphi_{new} := \textit{consequent} \implies C_{spec}$. Then, φ_{new} is provided as input to LTL2SMV that will generate a new SMV file ($SMV_{LTL2SMV}$) consisting of K corresponding to φ_{new}. Finally, $SMV_{LTL2SMV}$ is integrated with $FSMV$, i.e., $SMV_{new} = prod(SMV_{LTL2SMV}, FSMV)$.

During the generation of $SMV_{LTL2SMV}$, additional variables are generated by LTL2SMV. Therefore, verification is performed once again with SMV_{new} and the counterexample C_M is generated. Due to space constraints, $SMV_{LTL2SMV}$ and additional variables that are generated by LTL2SMV are not shown in this paper. If additional variables are removed from the counterexample (C_M), then C_M is same as the concrete counterexample, i.e., $C_M \equiv C_C$.

3.4 Step 4: Identifying Erroneous States in the Counterexample

In this step, we identify the *erroneous* (E_{state}) and *correct states* (C_{state}) in the counterexample. For this purpose, this step takes C_M and SMV_{new} from *Step2* as its input. Using the approach by Barbon et al. [2], we face challenge **Ch2** (cf. Section 1) since it is not feasible to simulate the full system behavior due to limitations in NuSMV. Particularly, we are not interested in all possible initial states I but just the states from the counterexample in order to simulate them. Therefore we use all states with their variables and values of the counterexample as initial states I.

Thus, to identify E_{state} and C_{state} in the counterexample, each state from the counterexample is taken as an initial state. Behavior of the system is simulated

Algorithm 2 Algorithm for erroneous state identification

INPUT: SMV_{new} - SMV file integrating $SMV_{LTL2SMV}$ and $FSMV$, C_M - counterexample generated by using SMV_{new}, I -initial states
OUTPUT: E_{state} - erroneous state and C_{state} - correct state in the counterexample

1: E_{state}, global list1
2: C_{state}, global list2
3: **function** FINDERRORSTATE(C_M, SMV_{new}, I)
4:     **for** $n \leftarrow 0$ to SIZEOF(I) -1 **do**
5:         $S_{trace} \leftarrow$ RUNNUSMV($SMV_{new}, I(n)$)
6:         **if** ISEMPTY(S_{trace}) **then**
7:             $E_{state} \leftarrow E_{state} \cup C_M(n).stateLabel$
8:         **else**
9:             **for** $j \leftarrow 0$ to SIZEOF(S_{trace}) -1 **do**
10:                 **for** $k \leftarrow 0$ to SIZEOF(C_M) -1 **do**
11:                     **if** $S_{trace}(j).equals(C_M(k))$ **then**
12:                         $C_{state} \leftarrow C_{state} \cup C_M(i).stateLabel$

for every initial state, and each state from the simulation trace is compared with the counterexample trace. If a state from the counterexample is found in the simulation trace, then we consider it as C_{state}. If the simulation trace is empty, we consider the selected initial state as E_{state}.

This approach is defined in detail in Algorithm 2. From `line 4` to `line 12`, we iterate over all initial states I to get the simulation trace for each of these states. The number n of initial states is equal to the number of states in the counterexample C_M. Provided every initial state from I, that is, $I(n)$, and SMV_{new} as input to NuSMV, the simulation trace S_{trace} is generated (`line 5`). If the simulation trace is empty, NuSMV returns the message "the set of initial state is EMPTY" (see S3 in Figure 3). In `line 6`, we check whether S_{trace} is empty. If this is the case, we add the *state label* of the selected $C_M(n)$ to the E_{state} list (`line 7`). Otherwise, from `line 9` to `line 12`, we iterate over all states in the S_{trace} and $C_M(n)$ to compare each state from S_{trace} with every state in C_M (`line 11`). If any of the state from S_{trace} is found in C_M, the *state label* is added to the C_{state} list (`line 12`).

For our running example, the counterexample C_M shown in Figure 3 is generated because the variable *exploded_cmd* from guarantee $I2G2$ holds true for future time step but *exploded* from guarantee $O1G1$ holds true immediately for next time step. Selecting *State 1.1* as $I(n)$ from C_M, we get S_{trace} S1 and S2. Comparing S_{trace} with C_M, {*State 1.2, State 1.3, State 1.5*} are found as E_{state} and {*State 1.1, State 1.4*} are found as C_{state}. In Figure 3, the states marked with E resulted from empty traces due to the NuSMV result S3, while the states marked with C match with the simulation trace S1 or S2.

3.5 Step 5: Identifying Erroneous Variables in the Counterexample

A system is also defined in terms of variables. By finding the *erroneous variables* E_{var} in the counterexample, we isolate the correct variables. Thus, we can focus

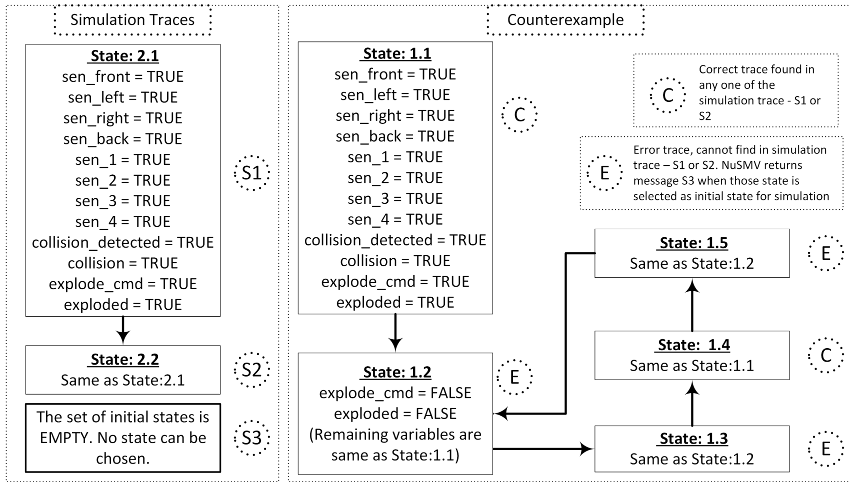

Figure 3: Simulation and Counterexample traces.

on highlighting E_{var} to support error comprehension and debugging since these variables are related to the violation of the refinement formulae. To identify E_{var}, we use E_{spec} obtained in *Step2*. In general, an LTL specification consists of temporal operators, mathematical symbols, variables, and values of variables. Removing temporal operators, mathematical symbols, and values from a specification, the variables can be identified. The same principle is applied to E_{spec}, which is an LTL specification, to identify E_{var}. For the airbag system, the guarantee $O1G1$, that is, $G(sen\_front \ \& \ sen\_left \ \& \ sen\_right \implies X \ exploded)$ is identified as E_{spec}. Removing the temporal operators, mathematical symbols, and values, we obtain *sen_front, sen_left,sen_right*, and *exploded* as E_{var}.

3.6 Step 6: Identifying Violated Contract Details

In this step, we identify whether the erroneous specification E_{spec} belongs to an assumption or guarantee, along with the respective component of the system. E_{spec} found in *Step2* belongs to a contract, that is, it will be either an assumption or a guarantee. It is possible to recognize whether the identified E_{spec} belongs to one or the other based on Equations (1) and (2). In Equation (1), the *consequent* contains only the assumptions of sub-components, whereas in Equation (2), the *consequent* contains only guarantee of the top-component. Furthermore, the respective component can be identified by name from E_{var}. During the generation of *FSMV*, FASTEN avoids duplicate variable names by prefixing the variable names of a sub-component with the name of the sub-component, wheres the variable names of the parent component remain unchanged. However, the top-component name is used as the *Module* name in *FSMV*. Thus, we can use the variable names from E_{var} to identify the related components by name.

In the airbag system, Equation (4) is violated (see *Step1*), which is formulated using Equation (2). Thus, we know that the erroneous specification belongs

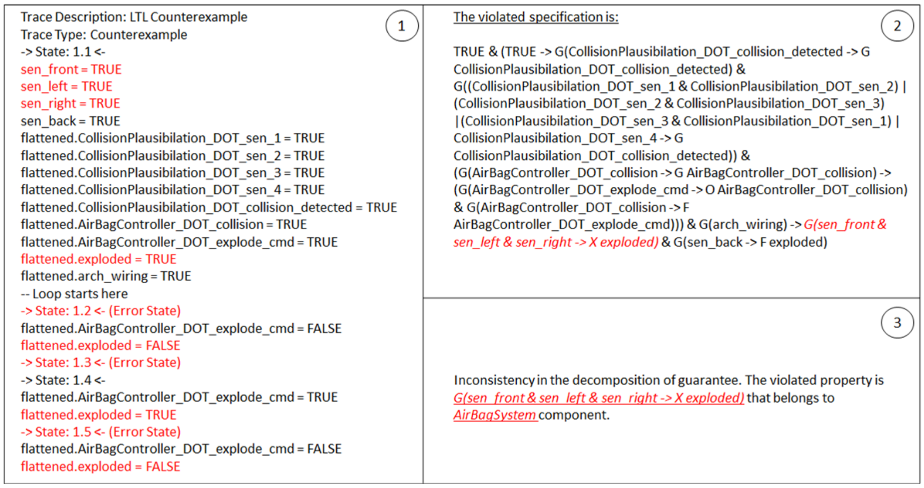

Figure 4: Result of our approach for the airbag system.

to a *guarantee*. The E_{var} found in *Step5* are *sen_front, sen_left,sen_right*, and *exploded*. These are not prefixed by a component name so that the respective component is the parent component, being the "AirBagSystem".

3.7 Step 7: Presentation of Results

As shown in Figure 2, the outcome of individual steps of our approach are used to support error comprehension by presenting three types of error information to an engineer: ① E_{state} and E_{var} in the counterexample found by *Step4* and *Step5*, ② E_{spec} in R found by *Step2*, and ③ Violated contract information found by *Step2* and *Step6*. This information is highlighted when showing the counterexample and failed specification to an engineer as exemplified by Figure 4 for the airbag system. Particularly, E_{var} (i. e., {*sen_front, sen_left, sen_right, exploded*}) and E_{state} (i. e., {*State 1.2, State 1.3, State 1.5*}) are highlighted in the complete counterexample to distinguish erroneous variables and states from non-erroneous variables and states ①. Similarly, the E_{spec} as the erroneous sub-specification is highlighted in the failed specification R ②. Information about the violated contract ③ consists of three elements: (i) that E_{spec} in R belongs to a *guarantee*, (ii) the concrete formulae of E_{spec} being $O1G1$, i. e., $G(sen\_front \ \& \ sen\_left \ \& \ sen\_right \implies X \ exploded)$, and (iii) that E_{spec} belongs to the "AirBagSystem" component.

4 Evaluation

Besides the airbag system introduced in Section 2, we evaluate our approach with two industrial systems: a Bosch electronic power steering for highly-automated driving [5] and a redundant sensor system [22]. An overview on the three use

Table 1: Overview of the three use cases

#	System	**Component Parent/Sub**	**Specifications** α / β	**Variables (i/o ports)**	**States in Counterexample**
1	Airbag System [32]	1 / 2	3 / 6	12	5
2	Redundant Sensor [22]	1 / 6	7 / 9	42	4
3	Electronic Power Steering [5]	1 / 8	9 / 24	26	5

Table 2: Evaluation results for the three use cases

#	System	**Variables/E_{var} (i/o ports)**	**States/E_{state} in Counterexample**	**Run-time (sec.)**	
				NuSMV	**NuXMV**
1	Airbag System [32]	12/4	5/3	1.528	1.622
2	Redundant Sensor [22]	42/2	4/1	109.347	110.973
3	Electronic Power Steering [5]	26/2	5/3	57.003	65.226

cases with the number of components, specifications (assumptions α and guarantees β), variables, and the size of the resulting counterexample is shown in Table 1. To obtain counterexamples for evaluation, specification errors were introduced in the use cases, that is, we modify either assumptions or guarantees of sub-components or vice versa for the top-component to cause an inconsistency between the contracts of the sub-components and the top-component.

Redundant Sensor System This system is an industrial use case developed by the European Space Agency project FoReVer[3]. It monitors the sensor outcomes to detect and possibly isolate failures to keep the output value reliable. It has 1 parent component, 6 sub-components, 7 assumptions, and 9 guarantees.

Electronic Power Steering (EPS) The EPS system is an industrial product from Bosch for highly-automated driving (HAD) vehicles. To cope with the availability demands of HAD, the system has two redundant channels: primary and secondary channel communication. The output from each channel is *master, slave,* or *passive* mode. The mode transition switches from *master* to *slave*, and *master* or *slave* to *passive*. The nominal behavior of the system is that either one of the channel should be *master* and the other one should be *slave*; in this case the system provides torque to motor for steering. The EPS system has 1 parent component, 8 sub-components, 9 assumptions, and 24 guarantees.

Results Looking at Figure 4, the complete information labeled with 1 and 2 but without the highlighted parts is returned by NuSMV. It is cryptic and does not give any explicit error information or explanation, which makes error understanding even harder. To overcome this issue and to improve the error understanding and usability, the *erroneous states*, *variables*, *specification* and *violated contract information* are highlighted and presented to engineers. This helps the engineers in debugging to identify the cause of error.

The results provided in Table 2 indicate that our approach can help engineers in identifying the error in the counterexample compared to analyzing the concrete and complete counterexample given by a model checker. For the redundant sensor system, the number of *erroneous states* found in the counterexample is 1.

[3] `https://es-static.fbk.eu/projects/forever/`

Therefore, an engineer can focus on this single state to identify the error instead of understanding 4 states in the counterexample. In addition, the approach also highlights that only 2 variables are responsible for the violation. Therefore, the user can focus only on 2 variables while ignoring all other 42 variables. Similarly for EPS, out of 5 total states in the counterexample, 3 are *erroneous states* and from 26 variables, 2 variables are found to be *erroneous variables*. This shows that our approach is able to reduce number of states and particularly of variables that need to be understood and investigated to find the error.

Implementation The seven steps described in Sect. 3 are developed as a script in Java that runs NuSMV in batch mode and triggers LTL2SMV. NuXMV [6] is used as an alternative to NuSMV, to evaluate the performance of verification (Table 2), although our goal was not to improve the run-time of model checking.

Constraints of our approach During the evaluation, we identified some constraints for the application of our approach. If a port or variable is declared but its behavior is not defined by contracts, it affects the generated simulation traces and the result is not reliable. Further, our approach cannot be applied if a specification is inconsistent by itself. For example, $G(A \implies \neg A)$ is inconsistent by itself and our approach cannot handle such a scenario.

5 Related Work

The main motivation of this work is to identify the *erroneous specification*, *states*, and *variables* from the violated specification and counterexample. Existing work addresses the identification of *erroneous specification* and *states*.

The approach by Langenfeld et al. [20] verifies inconsistencies for real-time requirements and is evaluated with industrial use cases. Crapo et al. [13] and Moitra et al. [23] use the Requirements Analysis Engine (RAE) of ASSERT that accept formal requirements in an easily understandable syntax by making use of a domain ontology. Further, RAE of ASSERT analyze an incomplete set of requirements and localizes the error by identifying the responsible requirements with an error marker. For finding *erroneous states* or traces in a counterexample, Jin et al. [16] presented an enhanced error trace that explicitly distinguishes fated and free segments. Fated segments show unavoidable progress towards the error while free segments show the choices that, when avoided, might have prevented the error. Hence, demarcation into segments highlight the critical events.

Pakonen et al. [29] presented a method that assists with identifying the root of the failure in both the model and the specification, by animating the model of the function block diagram as well as the LTL property. The counterexample visualization and explanation from Pakonen et al. addresses both aspects: finding the root cause of failure in the model and finding the failure in the trace.

6 Conclusion and Future Work

To the best of our knowledge, the presented approach is the first one to identify the *erroneous specification*, *states* and *variables* from counterexamples in CBD.

The presented approach aids in finding erroneous specification in the provided contracts, and erroneous states in the counterexample, which improves error comprehension and usability aspects of CBD. Our approach is evaluated with one example and two industrial systems of different size and complexity. This shows that the approach scales up to the size of an industrial product.

"*Researchers in and educators in formal methods, we should strive to make our notations and tools accessible to non-experts.*" — Edmund Clarke [12, p. 638]

There are several open points and options to enhance the presented approach. Currently, the result is presented separately from the component model. In the future, the result will be lifted back to original component model, where it can be integrated and linked to model elements. The presented approach can identify the *erroneous specification* but not the specification or contract which is inconsistent with the *erroneous specification*. Finding the inconsistent specification along with the *erroneous specification*, improves error comprehension. While this paper supports error comprehension and understanding of counterexamples for experts, future work will focus on supporting interpretation of counterexamples for non-experts, e. g., through natural language like format by using domain terminology supported by ontologies as sketched in earlier work [19].

References

1. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
2. Barbon, G., Leroy, V., Salaun, G.: Debugging of behavioural models using counterexample analysis. IEEE Transactions on Software Engineering pp. 1–14 (2019)
3. Benveniste, A., Caillaud, B., Passerone, R.: A generic model of contracts for embedded systems. CoRR **abs/0706.1456** (2007)
4. van den Berg, L., Strooper, P.A., Johnston, W.: An automated approach for the interpretation of counter-examples. ENTCS **174**(4), 19–35 (2007)
5. Bozzano, M., Munk, P., Schweizer, M., Tonetta, S., Vozárová, V.: Model-based safety analysis of mode transitions. In: Proc. of SAFECOMP (2020), (In press)
6. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuxmv symbolic model checker. In: 26th Intl. Conference on Computer Aided Verification CAV. pp. 334–342 (2014)
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Intl. Conf. on Computer Aided Verification CAV. pp. 359–364 (2002)
8. Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: 38th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2012. pp. 21–28 (2012)
9. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking - History, Achievements, Perspectives. pp. 1–26 (2008)
10. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. Formal Methods in System Design **10**(1), 47–71 (1997)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
12. Clarke, E.M., Wing J. M., Formal methods: State of the art and future directions, ACM Comput. Surv. 28 (4) (1996) 626–643.
13. Crapo, A.W., Moitra, A.: Using OWL ontologies as a domain-specific language for capturing requirements for formal analysis and test case generation. In: 13th IEEE International Conference on Semantic Computing, ICSC. pp. 361–366 (2019)

14. Fenelon, P., McDermid, J.A.: An integrated tool set for software safety analysis. J. Syst. Softw. **21**(3), 279–290 (1993)
15. Grunske, L.: Towards an integration of standard component-based safety evaluation techniques with saveccm. In: Quality of Software Architectures. pp. 199–213. Springer (2006)
16. Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. STTT **6**(2), 102–116 (2004)
17. Kaiser, B., Weber, R., Oertel, M., Böde, E., Nejad, B.M., Zander, J.: Contract-based design of embedded systems integrating nominal behavior and safety. CSIMQ **4**, 66–91 (2015)
18. Kaleeswaran, A.P., Munk, P., Sarkic, S., Vogel, T., Nordmann, A.: A domain specific language to support HAZOP studies of sysml models. In: Model-Based Safety and Assessment - 6th International Symposium, IMBSA. pp. 47–62 (2019)
19. Kaleeswaran, A.P., Nordmann, A., ul Mehdi, A.: Towards integrating ontologies into verification for autonomous driving. In: ISWC 2019 Satellite Tracks (Posters & Demonstrations, Industry, and Outrageous Ideas). pp. 319–320 (2019)
20. Langenfeld, V., Dietsch, D., Westphal, B., Hoenicke, J., Post, A.: Scalable analysis of real-time requirements. In: 27th IEEE International Requirements Engineering Conference, RE. pp. 234–244 (2019)
21. Leue, S., Befrouei, M.T.: Counterexample explanation by anomaly detection. In: 19th International Workshop on Model Checking Software SPIN. pp. 24–42 (2012)
22. Marcantonio, D., Tonetta, S.: Redundant sensors. `https://es-static.fbk.eu/tools/ocra/download/RedundantSensors.pdf` (2014)
23. Moitra, A., Siu, K., Crapo, A.W., Durling, M., Li, M., Manolios, P., Meiners, M., McMillan, C.: Automating requirements analysis and test case generation. Requir. Eng. **24**(3), 341–364 (2019)
24. Munk, P., Abele, A., Thaden, E., Nordmann, A., Amarnath, R., Schweizer, M., Burton, S.: Semi-automatic safety analysis and optimization. In: 55th ACM/ESDA/IEEE Design Automation Conference (DAC) (2018)
25. Munk, P., Nordmann, A.: Model-based safety assessment with sysml and component fault trees: application and lessons learned. SoSyM. pp. 1–22 (2020)
26. Muram, F.U., Tran, H., Zdun, U.: Counterexample analysis for supporting containment checking of business process models. In: BPM. pp. 515–528 (2015)
27. Narizzano, M., Pulina, L., Tacchella, A., Vuotto, S.: Property specification patterns at work: verification and inconsistency explanation. ISSE **15**(3-4), 307–323 (2019)
28. Ortmeier, F., Thums, A., Schellhorn, G., Reif, W.: Combining formal methods and safety analysis – the formosa approach. In: Integration of Software Specification Techniques for Applications in Engineering. pp. 474–493. Springer (2004)
29. Pakonen, A., Buzhinsky, I., Vyatkin, V.: Counterexample visualization and explanation for function block diagrams. In: INDIN. pp. 747–753 (2018)
30. Papadopoulos, Y., McDermid, J., Sasse, R., Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Reliability Engineering & System Safety **71**(3), 229–247 (2001)
31. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57 (1977)
32. Ratiu, D., Gario, M., Schoenhaar, H.: FASTEN: an open extensible framework to experiment with formal specification approaches: using language engineering to develop a multi-paradigm specification environment for nusmv. In: FormaliSE@ICSE. pp. 41–50. IEEE / ACM (2019)
33. Sharvia, S., Papadopoulos, Y.: Integrating model checking with hip-hops in model-based safety analysis. Reliab. Eng. Syst. Saf. **135**, 64–80 (2015)