Combining Requirement Mining, Software Model Checking and Simulation-Based Verification for Industrial Automotive Systems

Tomoya Yamaguchi and Tomoyuki Kaga
TOYOTA MOTOR CORPORATION
{tomoya_yamaguchi,tomoyuki_kaga}@mail.toyota.co.jp

Alexandre Donzé and Sanjit A. Seshia
University of California, Berkeley
{donze,sseshia}@berkeley.edu

***Abstract*—The verification and validation of industrial closed-loop automotive systems still remains a major challenge. The overall goal is to verify properties of the closed-loop combination of control software and physical plant. While current software model-checking techniques can be applied on a software component of the system, the end result is not very useful unless the interactions with the physical plant and other software components are captured. To this end, we present an industrial case study in which we combine requirement mining, software model-checking, and simulation-based verification to find issues in industrial automotive systems. Our methodology combines the the scalability of simulation-based verification of hybrid systems with the effectiveness of software model-checking at the unit level. We presents two case studies: one on a publicly available Abstract Fuel Control System benchmark and another on an actual production SiLS (Software in the Loop Simulator) benchmark. Together these case studies demonstrate the practicality of the proposed methodology.**

I. Introduction

In recent years, functional requirements for automotive control systems have become far more sophisticated, leading to the development of more complex and larger scale control software. This in turn has increased the importance of verification and validation (V&V) processes in the automotive industry since software can affect the integrity of the automotive system as a whole.

The industry authors of this paper have been part of a team which, for more than a decade, has attempted to use verification techniques such as a model checking [1] on production automotive systems. The strong guarantees provided by model checking make it attractive for the automotive industry. Unfortunately, even with impressive tools, these attempts have proved to be time-consuming, generally requiring considerable person-hours and expertise to be applied, with little or no conclusive results and many false alarms. A major factor is that most tools can handle only small, unit-level components, whereas to be truly useful, one needs to map an issue found at the unit level to a system-level issue that an engineer can confirm.

In order to apply model-checking at the software component-level and deduce results at the system-level, one has to make the right assumptions on the interfaces of modules (pre- and post-conditions). To this end, this paper proposes to leverage recently-developed simulation-based verification techniques for cyber-physical systems (e.g. [2], [3]) that can be used for falsifying temporal logic properties as well as to mine specifications from simulation traces [4]. Such methods have proven to scale well and able to provide useful information about cyber-physical systems of industrial size and complexity. We show how requirement mining, simulation-based verification, and software model checking can be combined to (1) obtain more precise pre-conditions for software modules in order to reduce the number of false-positives from model checkers, and (2) to guide the search for concretizing probable issues at the system levels when they do exist. We present results on a case study of an Abstract Fuel Control System benchmark [5] as well as on an actual production powertrain design in a SiLS (Software in the Loop Simulator) setting. We show that the resulting V&V methodology is more scalable than software verification and provides better guarantees than simulation-based verification.

II. Overview of Our Approach

In order to address the V&V problem identified in the preceding section, we identified two tasks which can help:

- Finding good pre-conditions for unit level software components, which characterize the states they can reach in the closed-loop system.
- Mapping counterexamples found at the unit level to "system-level" counterexamples, i.e., concretizing the unit-level counterexample on the closed-loop system.

Right now, the first item is performed manually. The second item is also performed manually, but only incompletely — the unit level counterexample is validated but typically not extended to a system level counterexample. In our experience, *finding good pre-conditions* takes up to 20% of total model checking person-hours and *validating a unit-level counterexample* takes 50% of total person-hours [6] [12].

We therefore propose a methodology that combines simulation-based verification of the closed-loop system with software model checking at the unit level. Software model checking is exhaustive, and simulation-based verification is scalable to the system level: therefore, their combination allows us to find corner-case issues in the code that generate counterexamples at the system level.

More specifically, this methodology combines requirement mining, software model checking and simulation-based verification in a complementary fashion. The key steps in this methodology are as follows (see flowchart in Fig. 1):

1. *Pre-condition (range) mining:* Using a system for mining requirements of closed-loop cyber-physical systems [4], we generate pre-conditions for a software component in terms

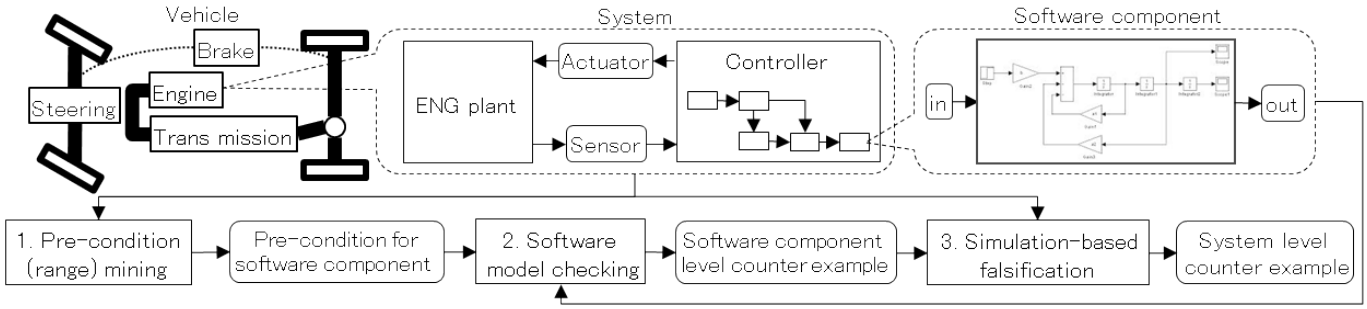

Fig. 1. Composition of vehicle system & proposed method

of ranges of values that selected interface variables must always lie in.

In general, the requirements are specified in signal temporal logic [7] [12]. For the purpose of this step, the STL specification is parametrized, and has the syntactic form below:

$$\begin{aligned} &\boldsymbol{x} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \\ &\pi_{min} = (\pi_{min\ 1}, \ldots, \pi_{min\ n}) \\ &\pi_{max} = (\pi_{max\ 1}, \ldots, \pi_{max\ n}) \\ &\square \left(\bigwedge_{i=1}^{n} ((\pi_{min\ i} \leq \boldsymbol{x}_i) \wedge (\boldsymbol{x}_i \leq \pi_{max\ i})) \right) \end{aligned} \tag{1}$$

where $\boldsymbol{x}$ are input variables of the target software component, and π_{min} and π_{max} are parameters to be mined. (Our technical report[12] shows how the tool operates.)

2. *Software model-checking:* Given the generated pre-conditions, we run a software model checker to check assertions or post-conditions for a unit-level software component. If any unit-level counterexamples are found, then we go to the next step. Otherwise, we can mark this component as verified.

   We use off-the-shelf software verifiers. For the case studies, we used Simulink Design Verifier (SLDV, [8]) and the C Bounded Model Checker (CBMC, [9]). (Our technical report [12] includes background on these tools. If the software verifier generates a counterexample, then we map this counterexample back to an assignment to the input variables, and denote this assignment by the vector $\widehat{\boldsymbol{x}}$.

3. *Simulation-based Falsification:* Given a counterexample for a unit-level software component, we try to extend it to a system-level counterexample by the use of simulation-based verification [2], [3]. The use of these tools requires the unit-level counterexample to be encoded into a suitable property expressed in signal temporal logic [7]. If the tool succeeds in finding a system-level counterexample, then this is passed on to engineers who then cross-check whether this issue can indeed occur. Otherwise, we go back to the previous step and try to find more unit-level counterexamples. (Our technical report [12] describes the working of the falsifier (Breach [2])). Briefly, we formulate an STL property that states that the value of the input variables of the software component $\boldsymbol{x}$ always remain at least an $\epsilon > 0$ distance away from the counterexample assignment $\widehat{\boldsymbol{x}}$, as shown below:

$$\varphi(\boldsymbol{x}) = \square \left(\sqrt{\sum_{i=1}^{n} (\boldsymbol{x}_i(t) - \widehat{\boldsymbol{x}}_i)^2} \geq \epsilon \right) \tag{2}$$

Algorithm 1 AF target decision

1: **if** $60.0 \leq throttleAngle \leq 62.0$ and $-2.0 \leq waterTemp \leq 2.0$
2: and $((airFlow$[g/s]$< 0.0)$ or $(10.0 \leq airFlow \leq 11.0))$ **then**
3: $\quad airFuelRatioTarget \leftarrow 12.5$ ▷ injected fault
4: **else**
5: $\quad airFuelRatioTarget \leftarrow 14.7$ ▷ original code
6: **end if**

We refer to this property as Property Eq. 2. Breach uses numerical optimization to search for a counterexample. If it finds one, this is a system-level counterexample showing how $\widehat{\boldsymbol{x}}$ can be extended to the entire closed-loop system.

III. Case Study 1: Abstract Fuel Control System

In this section, we present an evaluation of our methodology on an Abstract Fuel Control System (AFC) model [5]. This model was proposed by Toyota researchers [5] as a synthetic challenge problem representative of some of the key verification challenges faced (see [12] for details).

Description. This fuel control model is a subsystem of gasoline engine and implemented in Simulink [10]. The purpose of this model is to control the engine air-fuel ratio so as to meet emissions targets — an important control functionality in a gasoline engine. The model contains the air-fuel controller and a mean value model of the engine dynamics, such as the throttle and intake manifold air dynamics. Inputs of this system model are $throttleAngle$, $engineSpeed$ and $waterTemp$. Outputs are $airFuelRatio$, $airFuelRatioTarget$ and $controllerMode$.

Injected fault: We revised this model to inject a rare case malfunction into one of the software components. This software component has 3 inputs: $airFlow$, $throttleAngle$ and $waterTemp$ and one output:$airFuelRatioTarget$. The injected malfunction sets $airFuelRatioTarget$ to 12.5 under a rare condition (see Alg. 1). The post-condition for this model is that $airFuelRatioTarget \geq 13.0$.

The injected fault is highly representative of true issues that come up during production. This model is a functional approximation of a more complex A/F reference decision unit that would include a latent malfunction. It is desirable to find such issues early in the development cycle. However, such a rare case malfunction is difficult to find using random testing or simulation-based methods in general.

Experimental Results: The input variables $\boldsymbol{x}$ are $airFlow$, $throttleAngle$, and $waterTemp$. The result of range mining provides $airFlow$[g/s]$= [2.6, 34.0]$, $throttleAngle$[deg]$= [0.0, 90.0]$ and $waterTemp$[°C]$= [-30.0, 100.0]$. Note that $airFlow$ is the only intermediate variable whose range was unknown before range mining. We applied SLDV with and

without the mined input ranges as a pre-condition. In both cases, SLDV finds counterexamples that violate the post-condition described above, but they are different. We show the counterexamples in Table I. We indicate the pre-condition ranges in the absence of range mining as no-condition, indicating that there is no constraint on the value of that input variable. The counterexample obtained without range mining turns out to not be feasible due to the negative value for $airFlow$ which is not possible when accounting for the physical plant dynamics.

In our methodology, we take the counterexample obtained with range mining, namely, $\widehat{\boldsymbol{x}} = [10.0, 60.0, -2.0]$ and run Breach to falsify Property Eq. 2. Breach finds a system-level counterexample that violates the post-condition. (Our technical report [12] shows this system-level counterexample.)

IV. Case Study 2: Production Powertrain System

Our second case study is a production powertrain system which is one of the production models under development. It is based on SMiL [11] which is an in-house SILS (Simulation-in-the-Loop-Simulation) environment developed at Toyota [12].

Description: This power train model comprises engine and transmission sub-systems as well as the entire controller code in C. The model has 5 external inputs: $pedalAngle$, $brakeAngle$, $shift$, $waterTemp$ and $airTemp$. $shift$ position is always fixed as "D" (drive) in this evaluation.

Injected Fault: Motivated by an actual issue that occurred during development, dealing with a malfunction triggered under a very specific combination of conditions in the C code, we injected a fault into this model. Alg. 2 shows the injected fault in code that decides a control target in a closed loop and has 8 input variables:

- $waterTemp$[°C]: Temperature of engine coolant.
- $atmosphericPressure$[hPa]: Atmospheric pressure.
- $gear$: Current gear position in transmission.
- $gearHoldFlag$: Status of lock-up.
- $idlFlag$: Status of engine idling.
- $catalystTempHIGHflag$: Turned ON when catalyst temperature becomes high.
- $fuelCutFlag$: Status of fuel cut, triggered when negative torque is required e.g. braking.
- $engRpm$[rpm]: Rotational speed of engine.

The post-condition of this code is $target < 150.0$. We now discuss how we attempt to find a system-level counterexample that violates this post-condition.

Experimental Results: We applied the methodology of Sec. II to this case study. Once again, Breach was used for the range mining and falsification steps, while, in this case, CBMC [9] was used as the software verification tool. As a reference, we also applied software model checking without range mining. Table II shows the counterexamples obtained at the unit level with and without range mining.

The counterexample obtained without range mining is not a true system-level counterexample, because, e.g., it assigns $atmosphericPressure$ to be greater than 2.0 hPa.

TABLE I
Counterexamples from SLDV with and without range mining. "ce" indicates the counterexample values.

input variable	with mining		no mining	
	range	ce	range	ce
$airFlow$[g/s]	[2.6, 34.0]	10.0	no-condition	-0.5
$throttleAngle$[deg]	[0.0, 90.0]	60.0	no-condition	60.0
$waterTemp$[°C]	[-30.0, 100.0]	-2.0	no-condition	-2.0

Algorithm 2 Injected issue on power train model

```
1: if waterTemp > WARMINGUP
2: and atmosphericPressure > THRESHOLD
3: and ((4th ≤ Gear ≤ 6th) or (gearHoldFlag = OFF))
4: and idlFlag = OFF and fuelCutFlag = OFF
5: and catalystTempHIGHflag = ON then
6:     if 2600.0 ≤ engRpm ≤ 2610.0
7: and 89.0 ≤ waterTemp ≤ 91.0 then
8:         target ← 150.0                    ▷ injected fault
9:     else
10:        target ← orignalTarget            ▷ original code
11:    end if
12: end if
```

However, when combined with range mining using Breach, our methodology can be used to lift CBMC's counterexample to the system level. For this, we once again use Breach's falsification feature to find a violation of Property Eq. 2 where $\widehat{\boldsymbol{x}} = [90.0, 1.0, 6, 0, 0, 1, 2605.0]$.

The system-level counterexample is visualized in Fig. 2. The triangles show that the post-condition is violated at around 21.0 sec. This violation occurs through a sequence of events involving both continuous signals in the physical plant and changes in discrete variables. We trace the sequence of events backwards (see Fig. 2). For the post-condition to be violated, $engRpm$ must reach 2605.0 and $catalystTempHIGHflag$ must be set to True. The latter condition occurs when high temperature of exhaust gas are present, which occurs in turn when a high value of $pedalAngle$ and heavy engine load ($engRpm$) are kept on for a certain amount of time. Further, to reach $engRpm$[rpm] = 2605.0, the system must start from low rotation such as idle mode and start mode. In addition, the $gear$ must change in a specified pattern based on the current $gear$, $engRpm$ and the vehicle speed. Finding such a complex sequence of events involving physical plant signals and software variables requires an approach such as ours that analyzes the closed-loop system.

To summarize, our methodology combines the unit-level exhaustiveness of software model checking with the system-level scalability of simulation-driven requirement mining and falsification. The system-level counterexamples obtained greatly enhance the productivity with which issues arising the development process can be debugged and fixed. In our experience, this approach significantly eliminates the manual effort in finding good preconditions (20% of total person hours of an engineer trained in formal methods) and validating a counterexample (50% of total person hours).

V. Discussion

We conducted another set of experiments to check whether using simulation-driven falsification directly to violate the unit

TABLE II
Counterexamples from CBMC with and without range mining. "ce" indicates the counterexample values.

input variable	with mining		no mining
	range	ce	ce
$waterTemp$[°C]	[-30.0, 100.0]	90.0	89.4
$atmosphericPressure$[hPa]	[0.0, 1.0]	1.0	3.5
$gear$	[0,6]	6	5
$gearHoldFlag$	0	0	0
$idlFlag$	[0,1]	0	0
$catalystTempHIGHflag$	[0,1]	1	1
$fuelCutFlag$	[0,1]	0	0
$engRpm$[rpm]	[0.0, 5310.9]	2605.0	2600.0

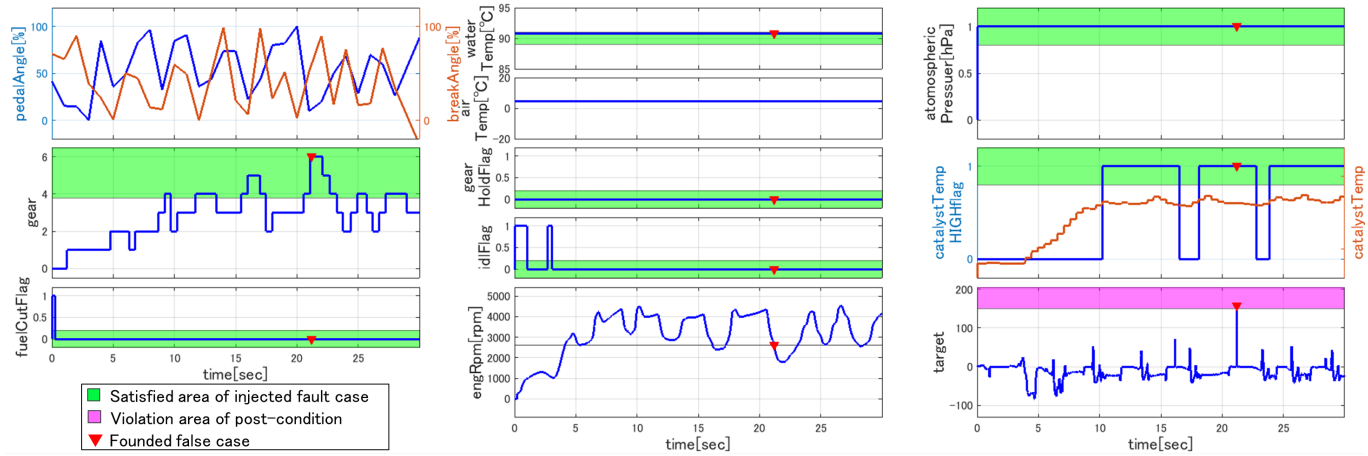

Fig. 2. System-level counterexample ("false case") on production powertrain system model (larger version in our technical report [12].)

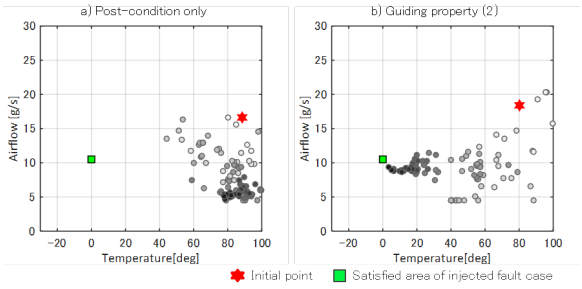

Fig. 3. Comparison between direct falsification of post-condition (left plot) and falsification guided using a counterexample and mined pre-conditions (right) for the AFC model. Each circle is an input found during falsification.

level post-condition, without the use of model checking, can be as effective as using software model checking first and then simulation to falsify Property Eq. 2. Specifically, we re-ran just the falsification step for 100 different trials on the AFC model with different initial input values, varying the property between one that tries to directly violate the post-condition and Property Eq. 2. Note that simulation-based falsification can be sensitive to the choice of initial input values, since it performs numerical optimization from this initial valuation.

We found that the combined approach could find the fault (and a system-level counterexample) 59.0% of the time, while a pure simulation-based approach could only find the fault 17.0% of the time. Further, as seen in Fig. 3, we see the visualization of one pair of trials for the different properties. The red star denotes the initial input valuation and the green box indicates the unit level counterexample to be hit. We can see that if we directly try to violate the post-condition, the optimizer gets stuck in a local minimum in the parameter space away from the fault region; whereas Property Eq. 2 is effective at guiding the search towards the unit level counterexample.

We also compared these options on the production power train model, and found that, on average, using Property Eq. 2 can find the injected malfunction faster than just using post-condition. The data is presented in our technical report [12].

In conclusion, in this paper we have shown that a combination of simulation-driven requirement mining, software model checking, and simulation-based falsification can be significantly more effective than using just software model checking or just simulation-based verification.

Going forward, we plan to expand the adoption of this methodology and also consider more complex requirements to be mined at the interface between the software components and the physical plant.

Acknowledgments

We thank the anonymous referees for their comments. We are grateful to Jyotirmoy Deshmukh, Xiaoqing Jin, James Kapinski, Hisahiro Ito, Arthur Wu and Ken Butts from Toyota Motor Engineering & Manufacturing North America, Inc. (TEMA) for their insightful comments and suggestions. We thank James Kapinski for providing the Abstract Fuel Control Model. We acknowledge the support on CBMC from Daniel Kroening, Martin Brain and Peter Schrammel. The UC Berkeley authors were supported in part by Toyota through the CHESS center.

References

[1] Edmund M. Clarke, Orna Grumberg, and Doron Peled. Model Checking. MIT Press, 2000.
[2] Donzé, A. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. CAV 2010: 167-170.
[3] ANNPUREDDY, Yashwanth, et al. S-taliro: A tool for temporal logic falsification for hybrid systems. Springer Berlin Heidelberg, 2011.
[4] Jin, X., Donzé, A., Deshmukh, J. V., & Seshia, S. A. Mining requirements from closed-loop control models. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2015, 34.11: 1704-1717.
[5] JIN, Xiaoqing, et al. Powertrain control verification benchmark. In: Proceedings of the 17th international conference on Hybrid systems: computation and control. ACM, 2014. p. 253-262.
[6] Tomoya Yamaguchi, et al. A model checking application to software development of automobile control systems. Embedded System Symposium 2012, 2012. p. 188-196. (Japanese)
[7] Maler, Oded; Nickovic, Dejan; Pnueli, Amir. Checking temporal properties of discrete, timed and continuous behaviors. In: Pillars of computer science. Springer Berlin Heidelberg, 2008. p. 475-505.
[8] http://www.mathworks.com/products/sldesignverifier
[9] Kroening, Daniel and Tautschnig, Michael. CBMC bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2014. p. 389-391.
[10] http://www.mathworks.com/products/simulink
[11] Fukuoka Koji, et al. Development of CRAMAS-VF. In: Fujitsu Ten technical report, 2014, 31.1: 15-20.
[12] Tomoya Yamaguchi, Tomoyuki Kaga, Alexandre Donzé and Sanjit A. Seshia. Combining Requirement Mining, Software Model Checking, and Simulation-Based Verification for Industrial Automotive Systems. EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2016-124, June 30, 2016