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

Tomoya Yamaguchi
Tomoyuki Kaga
Alexandre Donze
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-124
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-124.html

June 30, 2016

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 present 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]–[3] on production automotive systems. 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 problem 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. [4], [5]) that can be used for falsifying temporal logic properties as well as to mine specifications from simulation traces [6]. 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 [7] 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. Background

A vehicle integrates multiple systems (See Fig. 3) including, e.g., the engine, transmission, steering and brakes. These systems comprise a controller and a physical plant (i.e., a physical component to be controlled). The controller is composed of software components typically implemented in Simulink [8] or C.

In this section we give more background on the software development process for automotive control systems, and the previous experience of the industry authors with verification and validation for such systems.

A. Automotive Control Software Development

Automotive control software development follows the usual V-model development process (see Fig. 1). However, since the software is written to control actual physical phenomena, the traditional method of verifying the software is to validate in actual usage environments, i.e., using physical prototypes. Consequently, "spiral-up" type processes are also incorporated into the system development process.

The main characteristics of the development process are as follows.

- System design: Iterative development of the software design, while testing operation of the control system in

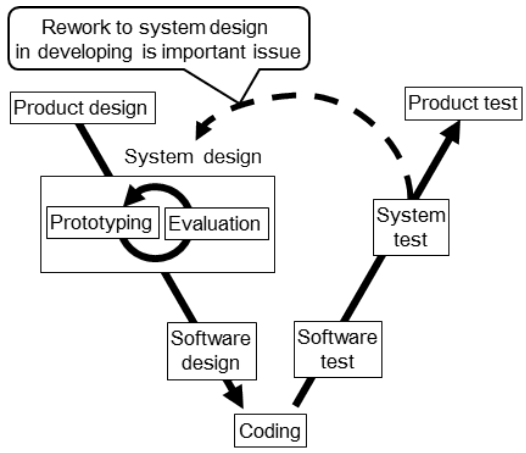

Fig. 1. Development process of a system controller

real environments. In particular, complex physical phenomenon involved in the engine system are considered and various tests are already applied;

- On the right side of V-model, various tests of the software on actual vehicles, engine test beds, HILS(Hardware-in-the-Loop-Simulation), and SILS (Simulation-in-the-Loop-Simulation) are applied hierarchically. Those are stronger and more exhaustive than at the system design phase.

This extensive testing, involving significant person-hours, is employed to achieve and maintain quality up to the final product test phase. The main cause of inefficiency is the need to return to an earlier phase of the development process to rework some aspect of the design. Additionally, software development tends to be incremental, focused on components, and re-using and extending legacy systems. When a new software component is designed or an existing one modified, it is critical that it be verified as extensively as possible to avoid undesirable interactions with other software components and with the physical world. Accordingly, the industry authors of this paper are part of a group that has been working for over a decade to investigate how software verification tools can be applied to production software.

B. Experience applying Software Model Checking

One approach to achieve exhaustive verification is to apply model checking at the unit level. Model checking is a method of formal verification based on an exhaustive exploration of the state space of a system [1]–[3]. Over a period of several years, various software model checking tools have been tried. However, this approach yielded little value while using considerable person-hours per experiment. In our experience, the biggest drawback of using model checkers was that, even after the steep learning curve for using these tools, they could only handle small unit level verification tasks, and when applied, would generate a very large number of false alarms. In Fig.2, we provide an example workflow, with corresponding work-hours. The most time-consuming task

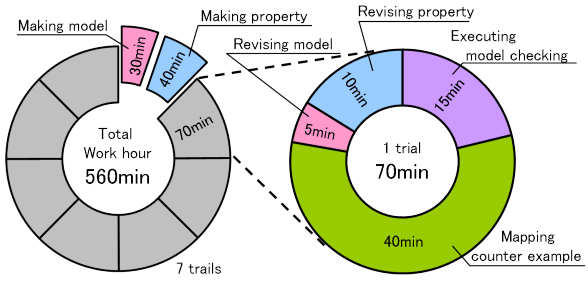

Fig. 2. Work-hours spent in each tasks while applying model checking.

consists in mapping counterexamples found at the unit-level to counter-examples at the system-level. Therefore, limiting the number of false positive by using more accurate pre-conditions and speeding-up the search for system-level counterexamples using simulation-based approaches, as suggested in this paper, can both contribute to significantly reduce the total time spent for verification.

Thus, in our experience, the effectiveness of software model checking is limited by false alarms arising because interactions with other components, in particular physical plants [9], are not captured in an open-loop setting. Moreover, model-checking the closed-loop system (software controller and plant) is impractical because of difficulties to model the plant and scalability issues.

III. Overview of Our Approach

In order to address the 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 [9].

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. 3):

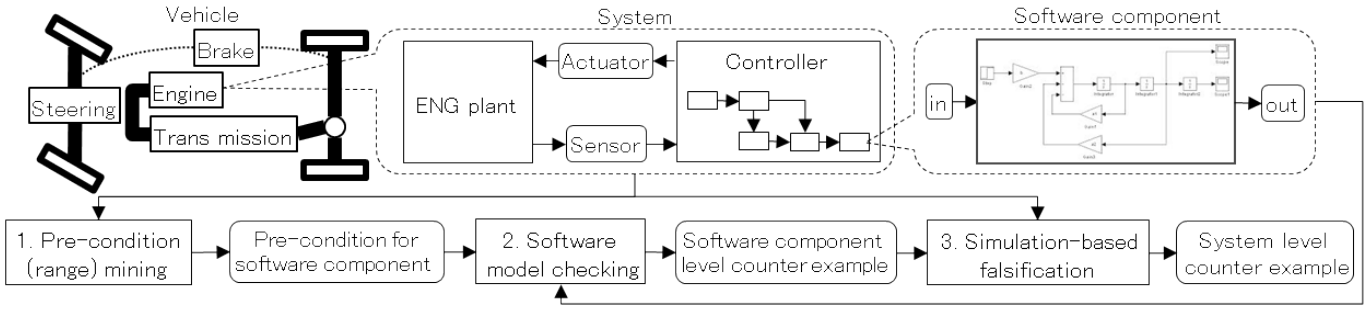

Fig. 3. Composition of vehicle system & proposed method

1. *Pre-condition (range) mining:* Using a system for mining requirements of closed-loop cyber-physical systems [6], we generate pre-conditions for a software component in terms of ranges of values that selected interface variables must always lie in.
   In general, the requirements are specified in signal temporal logic [10] (see Sec. III-A for background information about STL). 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. (See Fig. 5 in the section below showing 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 in this paper, we used Simulink Design Verifier (SLDV, [11]) and the C Bounded Model Checker (CBMC, [12]). We provide additional information on these tools below. 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 [4], [5]. The use of these tools requires the unit-level counterexample to be encoded into a suitable property expressed in signal temporal logic [10]. If the tool succeeds in finding a system-level counterexample, then this is passed on to engineers who then cross-check whether this problem can indeed occur. Otherwise, we go back to the previous step and try to find more unit-level counterexamples.
   We describe the working of the falsifier (Breach, [4]) later in this section. 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:

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

   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.

A. Background on Methods and Tools Used

The approach used in this paper was implemented using Breach [4] a Matlab/Simulink toolbox, and SLDV [11] or CBMC [12], both model-checking engines for Simulink models or C programs, respectively. In the following, we briefly describe the techniques implemented by these tools.

1) Signal Temporal Logic (STL): A signal $\boldsymbol{x}$ of dimension n is a function mapping any time value $t \in \mathbb{R}^+$ to a vector $\boldsymbol{x}(t) = (x_1(t), x_2(t), \ldots) \in \mathbb{R}^n$. In our context, signals are obtained from simulation of the closed-system combining a plant and a controller. Signal Temporal Logic (STL) is a language to specify constraints on signals. Atomic predicates are inequalities over the values of a signal, e.g.,$x_1(t) > 3$, $x_2(t) < -2$, etc. Arbitrary functions of $\boldsymbol{x}(t)$ are possible, e.g., $\sqrt{(x_1(t)-2)^2 + (x_2(t)+1)^2} > 0.1$.

STL Formulas can be combined using Boolean operators, e.g., $\neg(x_1(t) > 3)$, $x_2(t) < -2 \wedge x_4(t) > 4$, etc, or *temporal* operators. In this paper, we only use one: $\square$, aka "always". $\square\varphi$ means that the formula φ is true at all times t. Other STL temporal operators include $\Diamond$, aka "eventually", and $\mathcal{U}$, aka "until". Temporal operators can also be annotated with a time interval. E.g., $\square_{[2,4]}(\varphi)$ means that φ is true for all $t \in [2, 4]$.

STL admits a "quantitative" semantics ρ which maps a signal $\boldsymbol{x}$ and a formula φ to a value $\rho(\varphi, \boldsymbol{x})$ [6]. If this value is positive, then $\boldsymbol{x}$ satisfies φ, if it is negative, then $\boldsymbol{x}$ falsify φ. Its computation is based on the residual of atomic predicates, and on $\min$ (for conjunction and $\square$) and $\max$ (for disjunction and $\Diamond$) operators. E.g.,

- $\rho(x_1(t) > 2, \boldsymbol{x}) = x_1(t) - 2$,
- $\rho(x_1(t) < 1 \wedge x_2(t) > -2), \boldsymbol{x}) = \min(1 - x_1(t), x_2(t) + 2)$
- $\rho(\square_{[0,3]}(x_1(t) > 2, \boldsymbol{x}) = \min_{t \in [0,3]}(x_1(t) - 2)$,

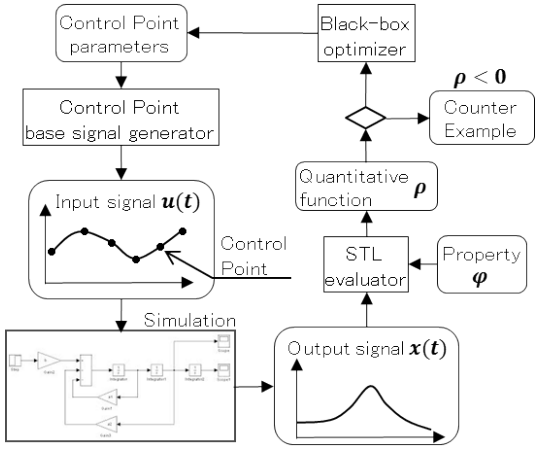

Fig. 4. Flowchart of simulation-based falsification. The optimizer minimizes ρ over possible control points which are converted into input signals $\boldsymbol{u}(t)$ using interpolation.

- etc.

Breach supports STL and the efficient computation of quantitative semantics for simulations obtained with Simulink.

2) Simulation-based Falsification: A system can be seen as a function taking an input signal $\boldsymbol{u}(t)$ and returning an output signal $\boldsymbol{x}(t)$. The falsification problem refers to the problem of finding an input signal $\boldsymbol{u}$ such that the corresponding output $\boldsymbol{x}$ falsifies a property φ. The tool Breach [4] solves this problem by parameterizing the input signals using control points (CP) and a choice of interpolation methods, and then formulating the problem as a minimization of the quantitative function ρ over the domain of values of the control points. If the minimization returns a negative objective function, then by definition of ρ, the corresponding signal $\boldsymbol{x}$ falsifies φ. The process is illustrated on Fig. 4. Different black-box optimizers can be used to solve the minimization problem, e.g., genetic algorithm (GA), simulated annealing (SA) or other commercial optimizers.

3) Requirement Mining: Requirement mining [6] is the problem of finding formulas that a system satisfies for all possible inputs. Breach solves a simplified version of this problem based on a parametric variant of STL, PSTL, and simulation-based falsification. Starting from a PSTL formula, i.e., an STL formula with unknown parameters, e.g., $\square(x(t) < p_1 \wedge x(t) > p_2)$, Breach first performs one simulation of the system, and find values for the parameters to satisfy the formula. Then, it solves a falsification problem for the instantiated STL formula. If falsification succeeds, the method is iterated until no counterexample can be found. The process is illustrated in Fig. 5.

4) Software Model Checking: The software verification problems considered in this study involve proving, for a given program written in C or Simulink, that a pre-condition of the program implies a specified post-condition. We use software model checking tools CBMC and SLDV for this purpose. A description of the techniques used in CBMC or SLDV is out of scope for this paper; however, the main point for our use case is that for both these tools, either the property can be proven, or a falsifying input (counterexample) is found. Both tools have been found to be quite efficient at the unit level.

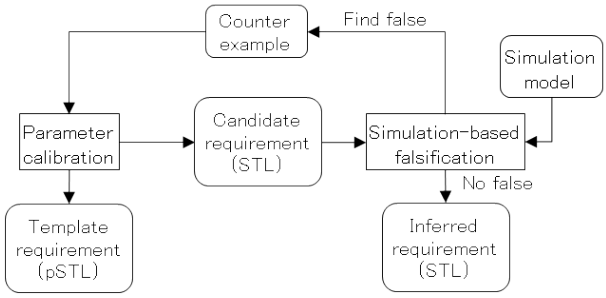

Fig. 5. Flowchart of counterexample guided requirement synthesis

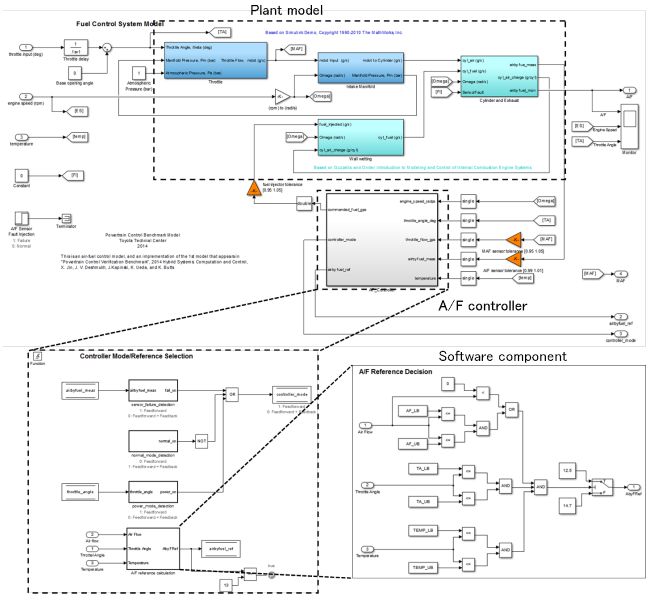

Fig. 6. Abstract Fuel Control System [7]

IV. 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 [7]. This model was proposed by Toyota researchers [7] as a synthetic challenge problem representative of some of the key verification challenges faced (see Fig. 6).

Description. This fuel control model is a subsystem of gasoline engine and implemented in Simulink. 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 $controllerMode$.

Here, we applied simulated annealing as a black-box optimizer and SLDV as BMC engine. In Table I, we give the control point (CP) settings for each input of the AFC model:

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: $airFuelRatioTarget \leftarrow 12.5$ ▷ injected fault
4: **else**
5: $airFuelRatioTarget \leftarrow 14.7$ ▷ original code
6: **end if**

number of CPs, range setting for each CP and interpolation methods. These settings are the same in both falsification and range mining phases.

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 $airFuelRatioTarget \geq 13.0$.

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; the ranges of the other two inputs are equal to the input ranges defined in Table I. 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 II. We indicate the pre-condition ranges in the absence of range mining as the interval [-∞, ∞], 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.

TABLE I
SETTING OF INPUT GENERATOR ON AF TARGET DECISION SOFTWARE COMPONENT

variable	#CP	range of CP	interpolation
$throttleAngle$[deg]	5	[0.0, 90.0]	Spline
$engineSpeed$[rpm]	5	[800.0, 3800.0]	Spline
$waterTemp$[˚C]	1	[-30.0, 100.0]	Spline

TABLE II
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	[-∞, ∞]	-0.5
$throttleAngle$[deg]	[0.0, 90.0]	60.0	[-∞, ∞]	60.0
$waterTemp$[˚C]	[-30.0, 100.0]	-2.0	[-∞, ∞]	-2.0

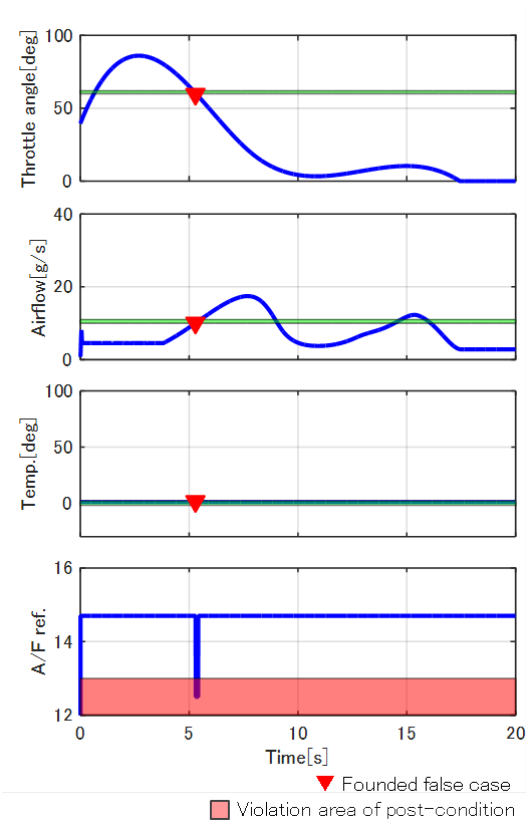

Fig. 7. System-level counterexample (false case) on Abstract Fuel Control System

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. This system-level counterexample is visualized in Fig. 7.

The red triangle shows how the signals $airFlow$, $throttleAngle$, and $waterTemp$ evolve until the post-condition property violation happens at nearly 5.5 sec. where the counterexample values $\widehat{\boldsymbol{x}} = [10.0, 60.0, -2.0]$ are obtained for those signals.

V. 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 [13], [14] which is an in-house SILS (Simulation-in-the-Loop-Simulation) environment developed at Toyota (see Fig. 8).

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.

For the more complex model, a commercial optimizer was used instead of standard simulated annealing. Also CBMC demonstrated better performance than SLDV. The control point settings for each input of the power train system model are

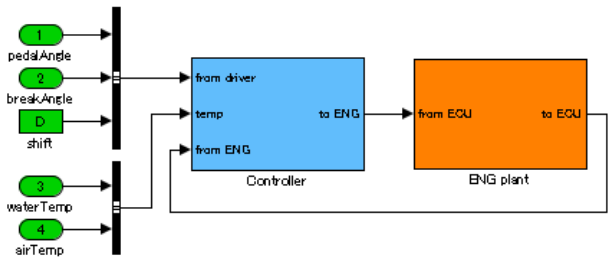

Fig. 8. Simulink model of a power train system model.

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
```

shown on Tab. III and are the same in both falsification and range mining phases. As simulation time was relatively long (about 36s. per simulation), we modified Breach native falsifier to support parallel computation, making it possible to compute multiple simulations simultaneously. Falsifying was stopped after 100 simulations. The software component in which we injected the malfunction has 8 inputs and one output. Range mining was run for 444 simulations with result shown on Tab. IV. The proposed combined methodology was able to find a falsifying case depicted in Fig. 9.

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. III to this case study. Once again, Breach was used for the range mining and falsification steps, while, in this case, CBMC [12] was used as the software verification tool. As a reference, we also applied software model checking without range mining. Table IV 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.

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. 9. 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. 9). 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 falsi-

TABLE III
SETTING FOR INPUT GENERATOR ON THE POWER TRAIN MODEL

variable	#CP	range of CP	interpolation
$pedalAngle$[%]	30	[0.0, 100.0]	linear
$brakeAngle$[%]	30	[0.0, 100.0]	linear
$waterTemp$[°C]	1	[-30.0, 100.0]	previous
$airTemp$[°C]	1	[-30.0, 40.0]	previous

$shift$ position is fixed as "D" range.

TABLE IV
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

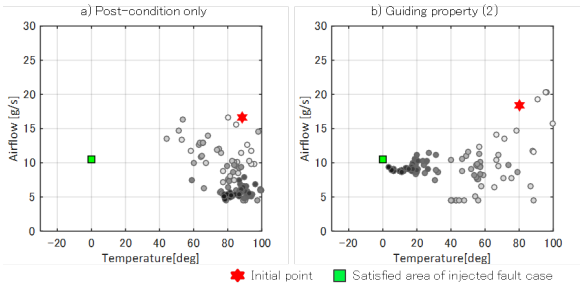

Fig. 10. Comparison between direct falsification of post-condition (left plot) and falsification guided using a counterexample and mined pre-conditions (right) for the AFC model. Each circle is an input found during falsification.

fication. 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) and validating a counterexample (50% of total person hours).

VI. Discussion

We conducted another set of experiments to check whether using simulation-driven falsification directly to violate the unit level post-condition, without the use of software 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. 10, 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 (malfunction).

We compared execution times of the proposed combined methodology vs post-condition only falsification. The results on AFC is shown in Tab. V and in Tab. VI and Tab. VII. All experiments were run on a desktop PC with i7-3770, 3.4GHz. with 4 cores.

The results show that on average, the total time of combined methodology is smaller than post-condition only falsification. Also, from Tab. VI, we see than in most cases and in average, the falsifying case is found using fewer (often many fewer) simulations than with the combined methodology.

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 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] Biere, Armin; Heule, Marijn; Van Maaren, Hans (ed.). Handbook of satisfiability. IOS press, 2009.
[3] Holzmann, Gerard J. The model checker SPIN. IEEE Transactions on software engineering, 1997, 23.5: 279.
[4] Donzé, A. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. CAV 2010: 167-170.
[5] Annpureddy, Yashwanth, et al. S-TaLiro: A tool for temporal logic falsification for hybrid systems. Springer Berlin Heidelberg, 2011.

TABLE V
Comparison of execution time on AFC

	Pre-con mining	Software MC	Sim-based falsification	total
Combined methodology	5[min]	few seconds	5[min]	10[min]
Post-condition only falsification	-	-	15[min] >(timeout)	15[min] >(timeout)

TABLE VI
Comparison of number of simulation on power train model for 10 trials

trial	1	2	3	4	5	6	7	8	9	10	ave
Combined methodology	205	161	80	12	80	48	80	72	89	85	91.2
Post-condition only falsification	325	8	1409	4	877	56	2805	80	24	1145	673.3

TABLE VII
Comparison of execution time on power train model

	Pre-con mining	Software MC	Sim-based falsification	total
Combined methodology	66.6[min]	few seconds	14.3[min] (average)	80.9[min]
Post-condition only falsification	-	-	105.9[min] (average)	105.9[min]

[6] 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.
[7] 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.
[8] http://www.mathworks.com/products/simulink
[9] Tomoya Yamaguchi, et al. A model checking application to software development of automobile control systems. Embedded System Symposium 2012, 2012. p. 188-196. (Japanese)
[10] 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.
[11] http://www.mathworks.com/products/sldesignverifier
[12] Kroening, Daniel; Tautschnig, Michael. CBMC bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2014. p. 389-391.
[13] Yasutaka Fujiwara, Hisahiro Ito, Harunaga Uozumi, and Koji Fukuoka. Development of Next Generation SILS. In: Proceeding No.122-12, published by Society of Automotive Engineers of Japan (JSAE), 2012. p. 1-4.
[14] Fukuoka Koji, et al. Development of CRAMAS-VF. In: Fujitsu Ten technical report, 2014, 31.1: 15-20.

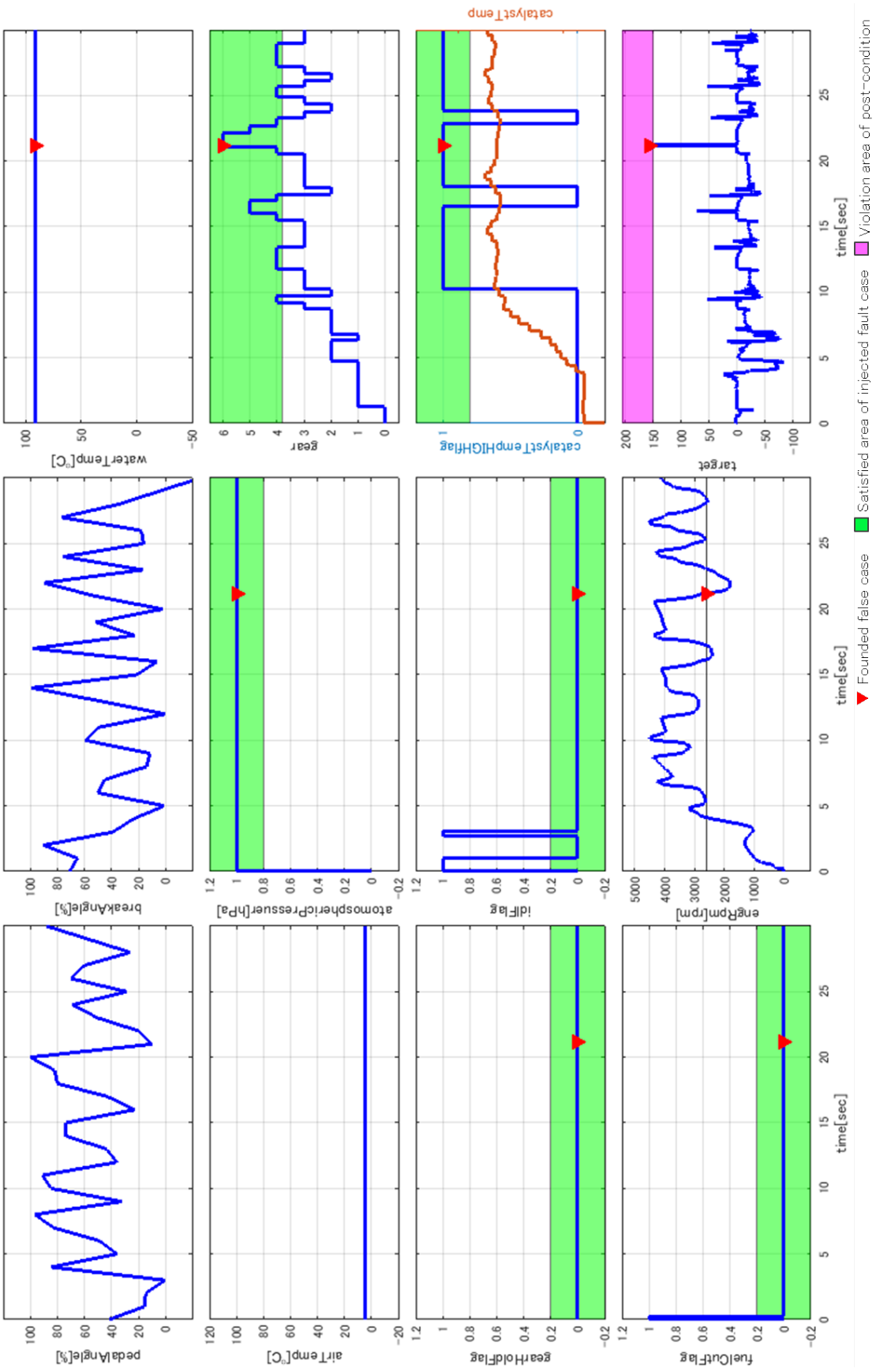

Fig. 9. System-level counterexample (“false case”) on production power train system model