Application of Formal Verification to the Lane Change Module of an Autonomous Vehicle

Application of Formal Verification to the Lane Change Module of an Autonomous Vehicle

Anton Zita[1], Sahar Mohajerani[1,2], Martin Fabian[2]

***Abstract*— For autonomous vehicles correct behavior is of the utmost importance, as unexpected incorrect behavior can have catastrophic outcomes. However, as with any large-scale software development, it is not easy to get the system correct. As the system is made up of multiple sub-modules that interact with each other, unexpected behavior can arise from incorrect interactions when one module may have unfulfilled expectations on the other. This paper describes how formal verification was applied to the lane change module of the decision and control software (under development) for an autonomous vehicle. The module was manually modelled as an extended finite-state machine, as were some of the requirements. When applying the *Supremica* software to perform the formal verification, some bugs were discovered in the model. Setting up additional unit tests triggering the incorrect behavior showed that this behavior was also present within the actual code. For some of the errors, applied corrections resulted in the absence of the particular error, thus demonstrating the power of true formal verification.**

***Index Terms*— Extended finite-state machines, Model checking, Non-blocking, Compositional verification, Supervisory control theory, Discrete event systems.**

I. Introduction

When implementing complex software systems, typically with multiple interacting sub-modules, even highly skilled programmers may introduce logical errors. Thus, especially for safety critical systems it is important to find and eliminate errors before they propagate into the implementation. This calls for tools to support the programmers.

One way of trying to eliminate logical errors is to use formal verification. This means to create a formal model of the system, and use mathematically proven methods to find errors. These types of *formal methods* [1] can show not only presence, but also *absence* of logical errors according to a given specification, assuming that the models capture the desired behavior in enough detail. However, formal verification has not yet found its place in industry to any significant extent, and it is still an active field of research.

One example of safety-critical software can be found in the automotive industry, which is rapidly moving towards autonomous vehicles that are capable of maneuvering themselves without any human input. Most of the major car companies are developing self-driving cars for the near future. To be able to drive autonomously, the vehicles are equipped with a number of different sensors to gather data about their surroundings. This data is then processed onboard, and depending on the current situation the software takes different actions. Since the vehicles will be in real traffic, software problems could potentially be disastrous, and great effort is therefore made to guarantee the correctness of the system.

Guaranteeing correctness is not easy, though. The vehicles operate within an unpredictable environment and creating test cases to cover all possible scenarios is impossible. As noted by [2], if a fleet of 100 autonomous vehicles drives 24 hours/day, 365 days/year, at an average speed of 25 mph, it takes 400 years to demonstrate with 95% confidence that their failure rate is 20% better than the human driver failure rate of 1.09 fatalities per 100 million miles (US, 2013).

To further illustrate this, in 2007, CalTech built an autonomous car for the DARPA Urban Challenge [3], named Alice. All the participant's vehicles were required to navigate through an urban-like environment. The tasks included parking, driving while adhering all traffic safety, etc. In part of the route during the competition Alice's behaviour was unsafe and almost led to a collision. After examination it was discovered that the reason for the failure was bad interactions between the reactive obstacle avoidance subsystem and the reacting path planner. This failure only happened in a very specific situation and no matter how many test cases were designed, it was very unlikely for the bug to be found [4].

One of the leading companies in autonomous driving is Volvo Car Corporation, VCC. VCC has an ongoing project called *Drive Me*. In this project a number of XC90's with autonomous driving capability will be launched to VCC-customers in 2017, to drive autonomously on some of the main roads around Gothenburg. Within this project, *formal verification* techniques were applied on a small part of the *Drive Me* software to show that this could indeed help in raising the confidence of code correctness. For this, the existing MATLAB [5] code was manually translated into *Extended Finite State Machines* (EFSM) [6], [7], which were then loaded into the formal verification (and synthesis) tool *Supremica* [8]–[10]. Existing verification techniques [11] were then applied to check whether the EFSM was able to reach any bad states, which would signify not fulfilling the given specifications. As it turned out, such bad states were found.

This paper is structured as follows. Section II collects the preliminary background of EFSMs, and Section III illustrates by an example how source code can be modeled in the EFSM formalism. Then, Section IV gives an overview of the overall system together with the model of it. Section V then presents

*This work has been performed with support from the Swedish Research Council (VR) under grant number 2016-00529.

[1] Anton Zita and Sahar Mohajerani are with Volvo Car Corporation, Gothenburg, Sweden `firstname.lastname@volvocars.com`

[2] Sahar Mohajerani and Martin Fabian are with the Department of Signals and Systems, Chalmers University of Technology, Gothenburg, Sweden `{mohajera,fabian}@chalmers.se`

two specifications that the systems should fulfill. The actual verification process is then described in Section VI where also the found errors are discussed. The paper concludes with Section VII where things are summed up and some future work is suggested.

II. Preliminaries

For modeling the kind of logical behavior dealt with here, where continuous dynamics can be disregarded and floating point data is used mainly for comparisons, it is beneficial to think in terms of *states* and *events*, where states represent situations where certain properties hold, and events are associated to transitions between the states that effect changes of those properties. A typical formalism for this is *finite-state machines* (FSM) [12], [13].

A. Extended Finite-State Machines

Extended finite-state machines (EFSM) are similar to conventional finite-state machines, but augmented with *updates* associated to the transitions [6], [7]; formulas constructed from variables, integer constants, the Boolean literals *true* and *false*, and the usual arithmetic and logic connectives.

A *variable* v is an entity associated with a bounded discrete domain $\mathrm{dom}(v)$ and an initial value $v^\circ \in \mathrm{dom}(v)$. Let $V = \{v_0, \ldots, v_n\}$ be the set of variables with domain $\mathrm{dom}(V) = \mathrm{dom}(v_0) \times \cdots \times \mathrm{dom}(v_n)$. An element of $\mathrm{dom}(V)$ is called a *valuation* and is denoted by $\hat{v} = (\hat{v}_0, \ldots, \hat{v}_n)$ with $\hat{v}_i \in \mathrm{dom}(v_i)$, and the value associated to variable $v_i \in V$ is denoted $\hat{v}[v_i] = \hat{v}_i$. The *initial valuation* is $v^\circ = (v_0^\circ, \ldots, v_n^\circ)$.

A second set of variables, called *next-state variables*, denoted by $V' = \{\, v' \mid v \in V\,\}$ with $\mathrm{dom}(V') = \mathrm{dom}(V)$, is used to describe the values of the variables after execution of a transition. Variables in V are referred to as *current-state variables* to differentiate them from the next-state variables in V'. The set of all update formulas using variables in V and V' is denoted by Π_V.

For an update $p \in \Pi_V$, the terms $\mathrm{vars}(p)$ and $\mathrm{vars}'(p)$ denote the sets of all variables, and all next-state variables, respectively, that occur in p. For example, if $p \equiv x' = y + 1$ then $\mathrm{vars}(p) = \{x, y\}$ and $\mathrm{vars}'(p) = \{x\}$. Here and in the following, the relation $\equiv$ denotes syntactic identity of updates to avoid ambiguity when an update contains the equality symbol $=$. An update p without any next-state variables, $\mathrm{vars}'(p) = \emptyset$, is called a *pure guard*.

One way to rewrite updates is *substitution*, which performs syntactic replacement of subformulas.

Definition 1: A *substitution* is a mapping $[z_1 \mapsto a_1, \ldots, z_n \mapsto a_n]$ that maps variables z_i to terms a_i. Given an update $p \in \Pi_V$, the *substitution instance* $p[z_1 \mapsto a_1, \ldots, z_n \mapsto a_n]$ is the update obtained from p by simultaneously replacing each occurrence of z_i by a_i.

For example, $(x' = x + y)[x' \mapsto 1, x \mapsto 0] \equiv 1 = 0 + y$.

With slight abuse of notation, updates $p \in \Pi_V$ are also interpreted as predicates over their variables, and they evaluate to $\mathbf{T}$ or $\mathbf{F}$, i.e., $p\colon \mathrm{dom}(V) \times \mathrm{dom}(V') \to \{\mathbf{T}, \mathbf{F}\}$. For example, if $V = \{x\}$ with $\mathrm{dom}(x) = \{0, 1\}$, then the update $p \equiv x' = x + 1$ means that the value of the variable x in the next state will be increased by 1 over its current-state value. Its predicate $p(x, x')$ evaluates as $p(0, 1) = \mathbf{T}$ and $p(1, 1) = p(0, 0) = p(1, 0) = \mathbf{F}$

Definition 2: An *extended finite-state machine (EFSM)* is a tuple $E = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$, where Σ is a set of events, the *alphabet*; Q is a finite set of *locations*; $\rightarrow\, \subseteq Q \times \Sigma \times \Pi_V \times Q$ is the *conditional transition relation*; $Q^\circ \subseteq Q$ is the set of *initial locations*; and $Q^\omega \subseteq Q$ is the set of *marked locations*.

The expression $q_0 \overset{\sigma:p}{\rightarrow} q_1$ denotes the presence of a transition in E, from location q_0 to location q_1 with event σ and update p. Such a transition can occur if the EFSM is in location q_0 and the update p evaluates to $\mathbf{T}$, and when the transition occurs, the EFSM changes its location from q_0 to q_1 while updating the variables in $\mathrm{vars}'(p)$ in accordance with p; variables not contained in $\mathrm{vars}'(p)$ are unchanged.

For example, let x be a variable with domain $\mathrm{dom}(x) = \{0, \ldots, 5\}$. A transition with update $x' = x + 1$ changes the variable x by adding 1 to its current value, if it currently is less than 5. Otherwise (if $x = 5$) the transition is disabled and no update is performed. An update $x = 3$ disables a transition unless $x = 3$ in the current state, and the value of x in the next state is not changed. Differently, the update $x' = 3$ always enables its transition, and the value of x in the next state is forced to be 3.

Given an EFSM $E = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$, its *variable set* is $\mathrm{vars}(E) = \bigcup_{(q_0,\sigma,p,q_1)\in\rightarrow} \mathrm{vars}(p)$, and it contains all the variables that appear on some transitions of E.

Usually, EFSM models consist of several interacting components. Such a model is called an *EFSM system.*

Definition 3: An *EFSM system* is a collection of interacting EFSMs,

$$\mathcal{E} = \{E_1, \ldots, E_n\}\ . \tag{1}$$

The alphabet of the system $\mathcal{E}$ is $\Sigma_{\mathcal{E}} = \bigcup_{E\in\mathcal{E}} \Sigma_E$, and the variable set of $\mathcal{E}$ is $\mathrm{vars}(\mathcal{E}) = \bigcup_{E\in\mathcal{E}} \mathrm{vars}(E)$.

Component interaction in an EFSM systems is modeled by synchronous composition, where shared events are executed in a "lock-step" fashion, while non-shared events are interleaved. In addition, updates of transitions labeled by shared events are combined by conjunction.

Definition 4: Given two EFSMs $E_1 = \langle \Sigma_1, Q_1, \rightarrow_1, Q_1^\circ, Q_1^\omega \rangle$ and $E_2 = \langle \Sigma_2, Q_2, \rightarrow_2, Q_2^\circ, Q_2^\omega \rangle$, the *synchronous composition* of E_1 and E_2 is $E_1 \parallel E_2 = \langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \rightarrow, Q_1^\circ \times Q_2^\circ, Q_1^\omega \times Q_2^\omega \rangle$, where:

$$\begin{aligned}
(x_1, x_2) \xrightarrow{\sigma:p_1\wedge p_2} (y_1, y_2) \quad &\text{if } \sigma \in \Sigma_1 \cap \Sigma_2,\ x_1 \xrightarrow{\sigma:p_1}_1 y_1, \\
&\text{and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ ; \\
(x_1, x_2) \xrightarrow{\sigma:p_1} (y_1, x_2) \quad &\text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } x_1 \xrightarrow{\sigma:p_1}_1 y_1\ ; \\
(x_1, x_2) \xrightarrow{\sigma:p_2} (x_1, y_2) \quad &\text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } x_2 \xrightarrow{\sigma:p_2}_2 y_2\ .
\end{aligned}$$

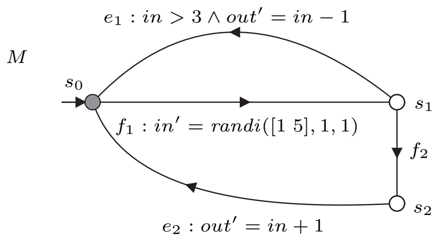
Using Def. 4, the global behaviour of a system $\mathcal{E} = \{E_1, \ldots E_n\}$ is given by $E_1 \parallel \cdots \parallel E_n$.

In an EFSM, the current state of the system is given by the current location together with the current values of all the variables. Since the variables are discrete and bounded,

```
function[out] = f1()
    in = randi([1 5],1,1);
    if in>3
        out=in-1;
    else
        out=f2(in);
    end
end

function[out]=f2(in)
    out=in+1;
end
```

Fig. 1. MATLAB-code for the example in Section III



M
s_0
$e_1 : in > 3 \wedge out' = in - 1$
s_1
$f_1 : in' = randi([1\ 5], 1, 1)$
f_2
s_2
$e_2 : out' = in + 1$

Fig. 2. EFSM model of the MATLAB-code shown in Fig. 1. The initial location is identified by the small arrow pointing into it, and marked locations are shaded.

the EFSM can be *flattened*, which introduces states for the combinations of locations and variable values [1].

Definition 5: Let $E = \langle \Sigma, Q, \rightarrow, Q^\circ, Q^\omega \rangle$ be an EFSM with variable set $\mathrm{vars}(E) = V$. The *monolithic flattening* of E is $U(E) = \langle \Sigma, Q_U, \rightarrow_U, Q_U^\circ, Q_U^\omega \rangle$ where

- $Q_U = Q \times \mathrm{dom}(V)$;
- $(x, \hat{v}) \xrightarrow{\sigma}_U (y, \hat{w})$ if E contains a transition $x \xrightarrow{\sigma:p} y$ such that $p(\hat{v}, \hat{w}) = \mathbf{T}$;
- $Q_U^\circ = Q^\circ \times \{v^\circ\}$;
- $Q_U^\omega = Q^\omega \times \mathrm{dom}(V)$.

$U(E)$ is the FSM representation of the EFSM, where all the variables have been removed and their values $\hat{v}$ embedded into the state set Q_U. This ensures the correct sequencing of transitions in the FSM.

III. Example

This section illustrates by an example how a function call in MATLAB [5] code can be modeled as an EFSM. The approach of the modelling is not to be considered as a general method that could, or should, be applied to all types of MATLAB-code, but rather how the modelling needed for the specific code of interest was performed.

Consider the MATLAB code shown in Fig. 1. The code consists of two functions $f1$ and $f2$. When $f1$ is called the variable in is assigned a random value between one and five, $randi([1\ \ 5], 1, 1)$. Then, if in is larger than 3, the output variable out is assigned $in - 1$, and if in is smaller or equal to 3, $f2$ is called where the variable out is assigned $in + 1$. Fig. 2 shows the EFSM model of the code. It uses the variables in and out with domain $\{1, 2, 3, 4, 5\}$ and $\{0, 1, 2, 3, 4\}$ respectively to represent the MATLAB variables. The updates on the transitions match the statements in the code. The transitions labeled with f_1 and f_2 represent calling functions $f1$ and $f2$, respectively. The variable in is assigned its random value in the update on the f_1 transition.

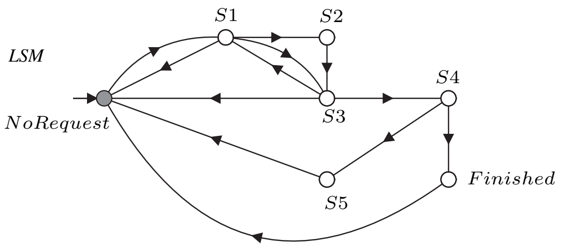


Fig. 3. The high level lateral state manager; the locations keep track of where in the process of the lane change the car is.

IV. System Description and Modeling

This paper focuses on a part of the lane change module called the *lateral state manager*. The implementation of the lane change module is written in object oriented MATLAB-code and simulations are made in the MATLAB/Simulink environment.

A. Planner

The lane change module is implemented with the use of several classes, all with different responsibilities. In this paper, the class *Planner* is considered. *Planner* has the responsibility to decide and control how the lane change should be done. *Planner* is cyclically updated at a high frequency with the current status of the vehicle, surrounding traffic situation, and current reference signals. The reference signals hold for example the current lane change request. With this information, *Planner* returns a path and required control signals to make a lane change in a safe and efficient way. Since the task of *Planner* is to calculate a path for the current inputs there is, with the exception of the lateral state manager (see below), no need to use data from previous updates.

B. Lateral state manager

Contrary to *Planner*, though, the Lateral State Manager (*LSM*) has to keep track of where in the process of lane change the car currently is, and thus it is implemented as a state machine as depicted in Fig. 3.

This state machine consists of seven locations, two of which are *NoRequest* and *Finished*. Each time that *Planner* is updated, the current state of *LSM* is changed based on the current location. Note that at most one transition can be fired each time *Planner* is updated.

When no lane change is requested *LSM* should be in *NoRequest*, which is also the initial state. Once a request comes, *LSM* moves to S1, and from there to either S2, S3 or back to *NoRequest* depending on the situation. Finally, when the lane change is done *LSM* will be in *Finished* and from there, on the next update, transit back to *NoRequest*.

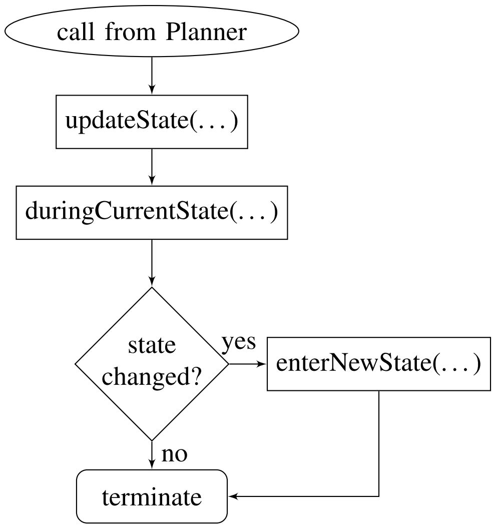


Fig. 4. Flow chart showing the execution sequence of methods used in lateral state manager.

C. Implementation

The implementation of *LSM* is done with a set of methods and variables in *Planner*, where the current state is one of the variables.

LSM consists of three different types of methods. The first type, of which there is only one implementation, is called *updateState*. This method is called from *Planner* every time that *Planner* is updated. The purpose of the *updateState* method is to call the current state's *duringUpdate* method, which is the second type of method; each state has its own *duringUpdate* method, named like *duringNoRequest*. The *duringUpdate* methods are where the decisions are taken and different parts of the code are executed depending on the inputs. Before the *duringUpdate* method terminates, it will either change the state or keep the current state. In the latter case, the same *duringUpdate* method will be called in the next update. If the state is changed by the *duringUpdate* method, the new state's *enterUpdate* method is called, which is the third type of method, before *updateState* is finished. In contrast to the *duringUpdate* methods which are executed repeatedly on each update while the state stays current, the *enterUpdate* methods are executed only once when the transition into the state occurs. This is illustrated by the flow chart in Fig. 4.

Modeling *LSM*'s 223 lines of MATLAB-code in *Supremica* was done manually, similar to what is described in Section III. This resulted in a single EFSM with 75 locations, 123 transitions, and 17 variables (14 Boolean, 2 three-valued, and a 7-valued variable holding the current location). The flattening of this EFSM plant model has 1 443 104 states (using *Supremica*'s per-event compiler), since combinations of the seventeen variables' values in the locations are represented as different states in the FSM.

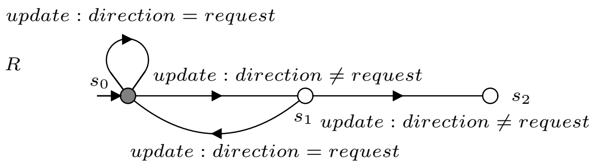

$update : direction = request$
R
s_0
$update : direction \neq request$
s_1
s_2
$update : direction \neq request$
$update : direction = request$

Fig. 5. Specification describing that the *direction* variable should always on each update be equal to the input parameter *request*.

V. Specifications

Two properties of the code modeled in the previous section will be specified; that a lane change is always made to the side, left or right, that is requested by *Planner*, and that some variables that should be incremented at most once each cycle are actually never incremented more than once.

A. Lane change request

LSM receives requests from *Planner* to change lane to either left or right. Thus, it does make sense to make sure that when a lane change is performed, it accords with the currently active request. In practice this means to check if the value of an internal variable, *direction*, and the incoming *request* parameter could be different from each other for two consecutive updates. A difference for one update would probably be sufficient, but a difference that persists over several updates would indicate that this was not only related to variable initialization in the model, but actually a property of the MATLAB-code. Fig. 5 shows R, the EFSM model of the lane change request specification.

When R of Fig. 5 is in its initial state s_0 and an update occurs, then if *direction* and *request* have the same value, R remains in s_0, whereas if the values differ, R transits to s_1. When in s_1, if on the update the variables now have the same values, R transits back to s_0. However, if the variables again have different values, then R transits to s_2. Now, from s_2 it is not possible to come back to the initial state, and so R blocks in s_2. The task of verification is to find whether this blocking state is reachable from the initial state or not, when R is synchronized with *LSM*.

B. Timer increment

In *LSM* there are three variables that are used as timers: *timer*$_1$, *timer*$_2$ and *timer*$_3$. These "timers" are really counters that keep track of during how many consequtive updates certain Boolean variables have been true, one timer for each such variable. When the value of the variable is false, the respective timer will be set to zero. When the value is true, the timer should be incremented on each update.

To check that a timer is incremented at most once per update, specifications for this were formulated for each of the timers. Fig. 6 shows T, the EFSM model of the timer specification. Just as in Fig. 5, the event *update* represents the update call from *Planner*. The *timerInc* event represents that the timer is incremented. If the timer is incremented and an *update* occurs before the next increment, everything is fine. But if two timer increments occur without an intermediate *update* the specification blocks.

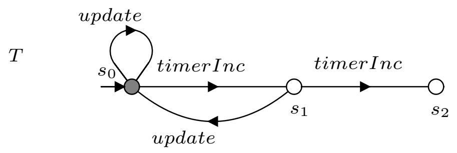


Fig. 6. Specification describing that a timer should be incremented not more than once per update.

VI. Verification

Given the *LSM* model and the specifications R and T, the task of verification is now to determine whether the bad states can be reached in $LSM \,||\, R$ and $LSM \,||\, T$, respectively. Due to the way that the specifications are formulated, the bad states are unreachable if and only if the system is non-blocking. Thus we are to verify whether $LSM \parallel R$ and $LSM \parallel T$ are non-blocking or not. However, due to the state-space explosion exhaustive enumeration of the states is time consuming (though possible), so instead *Supremica*'s compositional abstraction-based conflict check [11] is used.

A. Compositional approach

The notion of *non-blocking* concerns being able to always reach some marked state. This is formally defined for FSM [12], [13], but using the flattening of an EFSM (Def. 5), the notion can be directly applied also to EFSM.

Definition 6: An EFSM E is non-blocking if its monolithic flattening $U(E)$ is non-blocking.

Similarly, non-blocking can be defined for an EFSM system $\mathcal{E} = \{E_1, \dots, E_n\}$ (Def. 3), as its monolithic flattening $U(\mathcal{E}) = U(E_1 \parallel \cdots \parallel E_n)$ being non-blocking.

In this paper, there are two EFSM systems, $\mathcal{E}_R = \{LSM, R\}$ and $\mathcal{E}_T = \{LSM, T\}$, and since *LSM* is itself non-blocking, if $\mathcal{E}_R$ or $\mathcal{E}_T$ would be blocking, then it is known that the blocking states are introduced by the specification, and hence the specification is not fulfilled. However, calculating the monolithic flattening of the EFSM system when checking non-blocking leads to an (in the worst case) exponential increase in the number of states; the monolithic flattening of $\mathcal{E}_R$ has $3\,191\,904$ states, and $\mathcal{E}_T$ has $1\,256\,608$. Thus, this should be avoided.

The compositional abstraction-based approach [11] avoids the flattening of an EFSM system. Instead, at each step the individual EFSMs are *abstracted*. When no more abstraction is possible, either some components are composed (Def. 4) or some variables are *partially unfolded*, substituting (Def. 1) some variables with possible values and creating new EFSMs called *variable EFSMs* [11]. This procedure is continued until it leads to a single FSM, which is an abstract representation of the system, simple enough to be verified monolithically by standard means.

Abstraction means removing redundancy, and an abstracted component has less locations and/or transitions compared to the original component. This requires that the abstracted components relate properly to the original components. For compositional non-blocking verification this requires that the non-blocking property of the system does not change after the abstraction. To that end, *conflict equivalence* was introduced in [14], and [11] defines a variety of conflict equivalent abstraction methods for EFSMs. These all guarantee that the final abstracted system is non-blocking if and only if the original system was.

When *Supremica*'s compositional non-blocking verification algorithm (called *Conflict check* in the *Supremica* GUI) finds a blocking state, it generates a *counterexample*. This is a trace of events from the initial state to the blocking state. The counterexample can then be investigated in *Supremica*'s simulator, where it can also be simulated event by event, by clicking on the events of the trace.

B. False positives

As the model is built manually, errors not present in the code may be introduced into the model. A counterexample for such an error is called a *false positive*. Thus, when a counterexample is found, it must first be made sure that this is not a false positive. One way to do this is to write manually a unit-test that aims to trigger the issue in the actual code. However, this may fail, maybe because some coupling between variables is unknown, and then the validity of the model has to be carefully checked. For the specifications in Fig. 5 and Fig. 6 counterexamples were found that could be shown to exist in the actual code.

C. Verification of lane change request

With the lane change request specification of Fig. 5, the global system $LMS||R$ has $3\,191\,904$ reachable states. However, using the compositional approach the largest EFSM encountered during the compositional approach has 102 locations and 315 transitions, and the algorithm terminates in 0.07 seconds. The non-blocking verification showed that the specification is violated. In fact, there are $1\,033\,440$ blocking states, each of which represents a situation where the direction of lane change has been different from the current request for at least two consecutive updates. *Supremica* returned a 34 events long counterexample to one of these bad states, which indicated that when a first request was issued to go, say right, and then a second request to go left was issued in a later update, when *LSM* was in a specific region of its state-space and without cancelling the first request in-between, then the system entered a blocking state.

Since *LSM* is only a part of the lane change module it was not obvious what this discrepancy between the variable values could possibly lead to. Therefore, *Supremica*'s counterexample guided the design of a unit-test. The unit-test could also be visualized in a simulation environment used by the developers to run and test the actual code. The unit-test and simulations showed that for the counterexample scenario the vehicle checked for traffic in the (left) lane that it was supposed to change to according to the second request, but when the traffic was clear on that side, the vehicle in fact changed lane to the other (right) side in accordance with the first request. Thus was shown the existence of the issue in the actual code.

From the counterexample it seemed like that the problem could possibly be solved by adding an extra transition from S2 to *NoRequest*. Such a transition was added to the model. Unfortunately, verifying the system again showed that the problem had now moved to another region of the state-space. Since this indicated that the problem was not just a slip in one part, but probably a logical error that required an extensive over-haul of the code, no further investigation was made about finding solutions.

D. Verification of timers

With the timer increment specification, Fig. 6, the global system has 1 256 608 reachable states. However, using the compositional approach, for each of the three timers, the largest EFSM encountered during the compositional approach has 102 locations and 169 transitions, and the algorithm terminates in 0.06 seconds. The verification showed that two of the timers, $timer_1$ and $timer_2$, were indeed incremented not more than once on each update, but $timer_3$ could in certain situations be incremented twice. In the worst case, $timer_3$ could thus hold a value representing double the time that its corresponding Boolean variable was true.

With the help of the counterexample in *Supremica* it could be shown that the root of the problem was that $timer_3$ was incremented not only in a *duringUpdate* method, but also in an *enterUpdate* method. Thus, the increment of $timer_3$ in the *enterUpdate* method was removed in the *Supremica* model, and the verification was performed once again. This showed that the problem was solved and the code could easily be corrected in the same way.

VII. Conclusion

Using an EFSM model of a part of the lane change module for an autonomous vehicle, it was shown that unexpected interactions between that module and the higher level tactical decision module, such as first issuing a request and then issuing a different request without cancelling the first request in-between, resulted in incorrect behavior of the vehicle. This error was shown to exist also in the actual code. Furthermore, verification of suggested corrections to the problem showed that the issue was not straightforwardly resolved but moved to other parts of the system. Though this means that absence of the error could not be shown, it still gave valuable input as different fixes could be tried on the model before eventually (and hopefully) finding one that did resolve the issue.

In addition it was shown that a variable counting the number of updates, and which should be incremented only once per cycle, could in fact be incremented more than once, thus holding incorrect values. Here, formal verification did show that the suggested correction for the counter indeed removed that erroneous behavior.

Research is currently ongoing regarding how to incorporate these types of formal techniques into the daily engineering work flow. The two main obstacles being how to get the model from the code, and finding the specifications, many of which are not written down explicitly.

Future work includes the application of *synthesis* techniques, where the model is algorithmically adjusted to be guaranteed to fulfill the given specifications. The compositional synthesis feature of *Supremica* can do this, but the result is due to the abstractions incomprehensible to humans, and very hard (practically impossible) to relate back to the original model. Thus, more research is needed here concerning how to use the synthesis result in practice.

Further details, and other issues revealed by the verification process, are reported in [15].

VIII. Acknowledgments

The authors thank Pontus Peterson for the collaboration with the modeling and verification. We also thank Mohammad Ali and Roozbeh Kianfar at Volvo Car Corporation and Zenuity for insightful comments and giving us access to the relevant parts of the code.

References

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[2] N. Kalra and S. M. Paddock, *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, 2016. [Online]. Available: http://www.jstor.org/stable/10.7249/j.ctt1btc0xw

[3] DARPA, "DARPA Urban Challenge," http://archive.darpa.mil/grandchallenge/, accessed 2017-03-10.

[4] T. Wongpiromsarn, "Formal methods for design and verification of embedded control systems : application to an autonomous vehicle," Ph.D. dissertation, California Institute of Technology, 2010. [Online]. Available: http://resolver.caltech.edu/CaltechTHESIS:05272010-153304667

[5] MathWorks, https://www.mathworks.com/products/matlab.html.

[6] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of the 30th International Design Automation Conference*, ser. DAC '93. New York, NY, USA: ACM, 1993, pp. 86–91. [Online]. Available: http://doi.acm.org/10.1145/157485.164585

[7] M. Sköldstam, K. Åkesson, and M. Fabian, "Modeling of discrete event systems using automata with variables," in *Proceedings of the 46th IEEE Conference on Decision and Control*, 2007, pp. 3387–3392.

[8] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems," in *Proceeding os the 8th Workshop on Discrete Event Systems (WODES'06), Ann Arbor, MI, USA*, 2006, pp. 384–385.

[9] R. Malik, M. Fabian, and K. Åkesson, "Modelling large-scale discrete-event systems using modules, aliases, and extended finite-state automata," in *IFAC Proceedings Volumes. 18th IFAC World Congress, Milano, 28 August - 2 September 2011*, 2011, pp. 7000–7005.

[10] K. Åkesson, "Supremica," http://www.supremica.org/, 2016, online, accessed 2016-04-29.

[11] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional nonblocking verification of extended finite-state machines," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 26, pp. 33–84, 2016.

[12] P. J. G. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[13] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems, 2nd Edition*. Springer, 2010.

[14] R. Malik, "On the set of certain conflicts of a given language," in *Proceedings of the 7th International Workshop on Discrete Event Systems (WODES'04).* IFAC, Sep. 2004, pp. 277–282.

[15] P. Petersson and A. Zita, "Logical modelling and formal verification of decision and control functions for autonomous vehicles," Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.