Modeling and Synthesis of the Lane Change Function of an Autonomous Vehicle

Downloaded from: https://research.chalmers.se, 2020-04-06 06:49 UTC

Citation for the original published paper (version of record):
Modeling and Synthesis of the Lane Change Function of an Autonomous Vehicle
IFAC-PapersOnLine, 51(7): 133-138
http://dx.doi.org/10.1016/j.ifacol.2018.06.291

N.B. When citing this work, cite the original published paper.
Modeling and Synthesis of the Lane Change Function of an Autonomous Vehicle

Jonas Krook* Anton Zita Roozbeh Kianfar*
Sahar Mohajerani** Martin Fabian**

* Zenuity, Göteborg, Sweden
(e-mail: firstname.lastname@zenuity.com)
** Department of Electrical Engineering, Chalmers University of Technology, Göteborg, Sweden
(e-mail: {mohajera,fabian}@chalmers.se)

Abstract: Unexpected incorrect behavior of autonomous vehicles can have catastrophic outcomes. But, as with any large-scale software development, correctness of the system is not easily guaranteed. As the system is made up of multiple sub-modules that interact with each other, unexpected behavior can arise from incorrect interactions between the modules. In a previous paper, formal verification was applied to the lane change module of the decision and control software (under development) for an autonomous vehicle. This revealed incorrectness in the model, which could also be shown to exist in the actual software. Manual changes to the model did not result in absence of the incorrectness, and so in this paper we aim to patch the error by applying synthesis. The synthesized result is correct by construction, but it is not obvious what part of the functionality is disabled by the synthesis. Though different synthesis techniques were able to generate supervisors for the model, only when the supervisor was expressed as guard conditions on the events was it possible to interpret the effect of the synthesis. However, the supervisors put constraints on how the input data to the lane change module might change, so in the end the supervisors put behavioral requirements on the modules that generate the input to the lane change module.

Keywords: Extended finite-state machines, Synthesis, Supervisory control theory, Discrete event systems.

1. INTRODUCTION

The automotive industry is currently on a road towards autonomous vehicles that will maneuver themselves without human supervision or intervention, referred to as Level 4 and Level 5 automation according to the SAE standard (On-Road Automated Driving (Orad) Committee, 2016). These vehicles will operate in real traffic situations, together with other cars — both manually and autonomously driven — pedestrians, cyclists and all kinds of road users. Thus, their correct behavior is of utmost importance.

In order to deploy Level 4 automation and beyond on public roads, the automotive industry is required to follow the process recommended by ISO standard 26262 (ISO/TC 22/SC 32, 2012) to achieve functional safety. This in practice means that for certain safety requirements the probability of failure should be less than $10^{-9}$. Breaking down such requirements to a decision-making module, no more than 1 wrong decision should be made in $10^9$ hours of operation. Achieving such level of integrity and correctness makes any approach only based on simulation, field testing and statistical data inadequate and unrealistic. As noted by (Kalra and Paddock, 2016), 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).

Though system testing is still necessary, formal verification (Baier and Katoen, 2008) is a complementary tool, where a formal model of the system is subjected to mathematically proven methods to find counter-examples that show if and how the specified behavior is broken. The power of formal verification is that not only presence of counter-examples can be shown, but also absence of them and in that case the model has been exhaustively searched and is guaranteed to under no circumstances to break the specification. Assuming that the model is accurate in enough detail, this can then be said also about the modeled implementation.

One of the companies leading in development of software for autonomous driving is Zenuity. In a recent paper (Zita et al., 2017), we reported an application of formal verification to a small part of the software for autonomous driving being developed at Zenuity. The existing MATLAB (MathWorks, 2013) code was manually translated into Extended Finite State Machines (EFSM) (Cheng and...
Krishnakumar, 1993; Sköldstam et al., 2007), which were then loaded into the formal verification and synthesis tool Supremica (Åkesson et al., 2006; Malik et al., 2011; Åkesson, 2016). Some basic specifications were added, and existing verification techniques (Mohajerani et al., 2016) were used to check for counter-examples, the presence of which would signify non-fulfillment of the given specifications. As it turned out, such counter-examples were found.

Formal synthesis (Ramadge and Wonham, 1989; Mohajerani et al., 2014, 2017) can automatically remove all counter-examples related to a given plant with specification. Whereas formal verification points out a single counter-example, which the user can then contemplate, understand, and manually correct, synthesis automatically removes all counter-examples (using repeated verification and correction) in a structured but still “black-box” kind of way. Though the result will be correct by construction, it is not always clear what parts of the functionality that gets removed; for complex industrial systems the synthesis result might be highly complex and difficult to analyze. A designer might easily be fooled by the notion of proven correctness into believing that a successful synthesis means that the specifications are sound and that the synthesis results in a good system behavior. It is not obvious at all times how modeling and choice of synthesis method affect the possibility to effectively analyze the result, or the feasibility of the synthesis.

In this paper, we report a continuation of the work begun in Zita et al. (2017); Petersson and Zita (2016). Since the counter-example was not easily removed manually, we explore the feasibility of using formal synthesis to automatically remove all counter-examples related to the given specification. By doing this we hope to be able to patch the code, without having to model all specifications and implement the code anew. The system is complex (the system treated in this paper has more than $9.2 \cdot 10^5$ states and $1.3 \cdot 10^9$ transitions), so it is not trivial to find out if any useful behavior, such as in this case being able to turn both right and left, is removed. This is further accentuated by some synthesis methods that, to be able to cope with systems of huge complexity, employ abstractions to remove information redundant for the synthesis task, thus making the result next to impossible to relate back to the original system.

This paper reports the results of some synthesis experiments with the system modeled in Zita et al. (2017); Petersson and Zita (2016). Both the monolithic and the compositional abstraction-based synthesis of Supremica (Mohajerani et al., 2014, 2017) can indeed synthesize correct models, but the result is practically incomprehensible due to the size of the result in the former case, and due to the abstractions in the latter case. The BDD-based synthesis implemented in Supremica (Fei et al., 2014), though working on the monolithic model, can also synthesize a correct model, and since the result is returned as Boolean constraints on controllable events, the result is much more easily comprehended. However, the result puts constraints on how the input data to the lane change module might change, thus placing behavioral requirements on the modules that generate the input to the lane change module.

The paper is structured as follows. Section 2 collects the preliminary background of EFSMs, controllability, BDD-based synthesis, and compositional synthesis. Then, Section 3 gives an overview of the overall system together with the model of it. Section 4 then presents a specification that the systems should fulfill, but was in Zita et al. (2017); Petersson and Zita (2016) shown not to. The synthesis is then described in Section 5, which also discusses the result. The paper concludes with Section 6 where key findings are summed up and some future works are suggested.

2. PRELIMINARIES

The logical behavior considered in this paper can be modeled in terms of states and events; where states represent situations where certain properties hold, and events are associated to transitions between the states that affect changes of those properties. A typical formalism for this is finite-state machines (FSM) (Ramadge and Wonham, 1989; Cassandras and Lafortune, 2010). FSMs are a suitable formalism in this case, since continuous dynamics can be disregarded and floating point data is used mainly for comparisons.

Supremica implements several different synthesis algorithms for FSMs, among them ordinary monolithic synthesis, as well as synthesis based on binary decision diagrams (BDDs), and compositional abstraction-based synthesis. These all have different properties, some of which are investigated in the following sections.

2.1 Extended Finite-State Machines

Extended finite-state machines (EFSM) are similar to conventional finite-state machines, but augmented with updates associated to the transitions (Cheng and Krishnakumar, 1993; Sköldstam et al., 2007); formulas constructed from variables, integer constants, the Boolean literals true (T) and false (F), propositional logic connectives, and discrete arithmetic operators.

A variable $v$ is an entity associated with a bounded discrete domain $\text{dom}(v)$ and an initial value $v^0 \in \text{dom}(v)$. Let $V = \{v_0, \ldots, v_n\}$ be the set of variables with domain $\text{dom}(V) = \text{dom}(v_0) \times \cdots \times \text{dom}(v_n)$. An element of $\text{dom}(V)$ is called a valuation and is denoted by $\hat{v} = (\hat{v_0}, \ldots, \hat{v_n})$ with $\hat{v_i} \in \text{dom}(v_i)$, and the value associated to variable $v_i \in V$ is denoted $\hat{v}[v_i] = \hat{v_i}$. The initial valuation is $v^0 = (v^0_0, \ldots, v^0_n)$.

A second set of variables, called next-state variables, denoted by $V' = \{v' | v \in V\}$ with $\text{dom}(V') = \text{dom}(V)$, 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 $\Pi_V$.

For an update $p \in \Pi_V$, the terms $\text{vars}(p)$ and $\text{vars}'(p)$ 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 $\text{vars}(p) = \{x, y\}$ and $\text{vars}'(p) = \{x\}$. Here and in the following, the relation $\equiv$ denotes syntactic identity of updates to avoid ambiguity when an update contains the equality symbol $=$.
With slight abuse of notation, updates $p \in \Pi_V$ are also interpreted as predicates over their variables, and they evaluate to $T$ or $F$, i.e., $p : \text{dom}(V) \times \text{dom}(V') \rightarrow \{T, F\}$. For example, if $V = \{x\}$ with $\text{dom}(x) = \{0, 1\}$, 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 $p(0, 1) = T$ and $p(1, 1) = p(0, 0) = p(1, 0) = F$.

**Definition 1.** An extended finite-state machine (EFSM) is a tuple $E = (\Sigma, Q, \rightarrow, Q^0, Q^\omega)$, where $\Sigma$ is a set of events, the alphabet; $Q$ is a finite set of locations; $\rightarrow \subseteq Q \times \Sigma \times \Pi_Y \times Q$ is the conditional transition relation; $Q^0 \subseteq Q$ is the set of initial locations; and $Q^\omega \subseteq Q$ is the set of marked locations.

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

Given an EFSM $E = (\Sigma, Q, \rightarrow, Q^0, Q^\omega)$, its variable set is $\text{vars}(E) = \bigcup_{(q_0, \sigma, p, q_1) \in E} \text{vars}(p)$, and it contains all the variables that appear on some transitions of $E$.

**Definition 2.** Given two EFSMs $E_1 = (\Sigma_1, Q_1, \rightarrow_1, Q_1^0, Q_1^\omega)$ and $E_2 = (\Sigma_2, Q_2, \rightarrow_2, Q_2^0, Q_2^\omega)$, the synchronous composition of $E_1$ and $E_2$ is $E_1 || E_2 = (\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, \rightarrow, Q_1^0 \times Q_2^0, Q_1^\omega \times Q_2^\omega)$, where:

- $(x_1, x_2) \xrightarrow{\sigma_{1}, \sigma_{2}} (y_1, y_2)$ if $\sigma \in \Sigma_1 \cap \Sigma_2$, $x_1 \xrightarrow{\sigma_{1}} y_1$, and $x_2 \xrightarrow{\sigma_{2}} y_2$;
- $(x_1, x_2) \xrightarrow{\sigma_{1}, \sigma_{2}} (y_1, x_2)$ if $\sigma \in \Sigma_1 \setminus \Sigma_2$ and $x_1 \xrightarrow{\sigma_{1}} y_1$;
- $(x_1, x_2) \xrightarrow{\sigma_{1}, \sigma_{2}} (x_1, y_2)$ if $\sigma \in \Sigma_2 \setminus \Sigma_1$ and $x_2 \xrightarrow{\sigma_{2}} y_2$.

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, the EFSM can be flattened, which introduces states for the combinations of locations and variable values (Baier and Katoen, 2008).

**Definition 3.** Let $E = (\Sigma, Q, \rightarrow, Q^0, Q^\omega)$ be an EFSM with variable set $\text{vars}(E) = V$. The monolithic flattening of $E$ is $U(E) = (\Sigma, Q_U, \rightarrow_U, Q_U^0, Q_U^\omega)$ where:

- $Q_U = Q \times \text{dom}(V)$;
- $(x, \bar{v}) \xrightarrow{\sigma} (y, \bar{w})$ if $E$ contains a transition $x \xrightarrow{\sigma} y$ such that $p(\bar{v}, \bar{w}) = T$;
- $Q_U^0 = Q^0 \times \{v\}$;
- $Q_U^\omega = Q^\omega \times \text{dom}(V)$.

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

### 2.2 Controllability

A key property when it comes to synthesis is the notion of controllability. Synthesis removes states that are considered “bad” in the sense that they break the given specification, and transitions to bad states must also be removed to make the bad states unreachable in the model. However, some of these transitions are labeled by events that are considered uncontrollable in the sense that if the source-state of such a transition is reached there is no way to guarantee that the event labeling the transition will not occur taking the system to the bad state. A typical example is an event representing some input; if at some state this input may occur, the system must be ready to transit on that event, or not have that state at all reachable. This is expressed as the synthesis result must be controllable (Ramadge and Wonham, 1989; Cassandras and Laforet, 2010) with respect to the uncontrollable events. Controllability guarantees that the synthesized model can always remain within the specified behavior.

### 2.3 BDD-based Synthesis

A Binary Decision Diagram (BDDs) is a data structure that can efficiently store huge state-spaces and transition sets encoded as Boolean functions. The computational complexity of synthesis using BDDs does not depend on the number of states or transitions, but on the number of nodes in the BDD, as the computations are performed symbolically rather than explicitly; synthesis is performed on sets of states and transitions rather than single such elements. So even though the BDD-based synthesis works on a monolithic model, it can handle systems of considerable sizes. The approach presented in (Fei et al., 2014) uses partitioning techniques to further stretch the ability of the synthesis procedure.

The BDD-based synthesis works directly on the EFSM model, encoding the transition relation and the guards as Boolean expressions. The synthesis then results in a partitioning of the state-space into allowed and forbidden states, and then guarantees that only controllable events take the system from the allowed into the forbidden states. From this is then generated guards that represent the control actions of the supervisor. These guards are Boolean expressions over values of variables and state labels, that define when the particular event is enabled.

### 2.4 Compositional Synthesis

The compositional abstraction-based synthesis works on the flattened EFSM model by gradually abstracting system components and collecting partial supervisors at each step. Abstraction means merging states or removing transitions. Moreover, sometimes it can be determined that some states must eventually be removed by the synthesis. It is then possible to remove those states at an early stage and add the refined components as a partial supervisor to the supervisor set (Mohajerani et al., 2014, 2017). At the first step, the compositional synthesis abstracts each component and collects possible partial results. When no more abstraction is possible, some components are composed and the result is abstracted again. This procedure is continued until one single component is obtained. The final component has, due to the abstractions, less states or transitions compared to the original system. Monolithic synthesis is applied on this abstracted final component and the result is added to the supervisor set (Mohajerani et al., 2014, 2017).
Fig. 1. Lane change module system overview with signal flow. The Planner, lateral state manager, and path planner are depicted as boxes.

3. SYSTEM DESCRIPTION AND MODELING

This paper focuses on a part of the lane change module called the lateral state manager (LSM). The lane change module is implemented with the use of several classes, all with different responsibilities. The interactions between these classes, Planner, LSM, and the Path Planner, are depicted (see Fig. 1). The implementation of the lane change module is written in object oriented MATLAB-code. The lane change module is cyclically updated at a high frequency with the current status of the vehicle, surrounding traffic situation, and current reference signals.

Planner has the responsibility to decide in which lane to drive, and when. Based on the vehicle’s current position, it sends lane change requests to the LSM. These are three valued signals that can take the values NoRequest, ChangeLeft, and ChangeRight. The request reflects which lane is the desired lane, not that a lane change can or may be done right now.

The LSM receives the lane change request signal and issues commands to the Path Planner to safely change lanes, if Planner requests it. LSM has to keep track of where in the process of the lane change the car currently is, and thus it is implemented as a state machine.

Based on the commands from the Planner and the LSM, the Path Planner creates a path and required control signals to make a lane change in a safe and efficient manner. This is then executed by lower level controllers. Since the task of the Path Planner is to calculate a path for the current inputs there is no need to use data from previous updates. States are hence completely handled within LSM.

The implementation of LSM is done with a set of methods and variables, where the current state is one of the variables. The LSM code is called once every execution cycle, and it goes through three different stages during each call:

(i) All of the inputs are updated according to the function call arguments. This step is associated to the event update in the model.
(ii) A code snippet associated to the current state is executed. This code assigns output and persistent variables, and decides whether the system should transition into a new state.
(iii) If a transition occurs, then the third stage executes specific code connected to entering a certain state. This code assigns output and persistent variables.

A small example of this is shown in Fig. 2. The four locations to the left correspond to the high level LSM state called S0, and the four locations to the right correspond to the high level LSM state called S1. After an update call to the S1 state, the input variable request is first assigned a value. The internal code for the state is run during event $S1_{\text{update}}$. In the end, the request variable is checked to determine whether to stay in S1 or to transition into S0.

Fig. 2 is a minimal working example of the LSM and the properties that are relevant in this treatise. We will use it for illustrative purposes for the rest of the paper.

Modeling LSM’s 223 lines of MATLAB-code in Supremica was done manually, as described in Zita et al. (2017). This resulted in a single EFSM with 75 locations, 86 events, 123 transitions, and 17 variables (14 Boolean, 2 three-valued, and one 7-valued variable holding the current location). Using Supremica’s normalizing compiler, the flattening of this EFSM plant model has 370 864 states, 105 events, and 528 394 transitions.

These numbers are lower than what is quoted in Zita et al. (2017), because for verification the variables’ initial values could be left undetermined, whereas the synthesis algorithms implemented in Supremica can only handle deterministic systems, and so the initial values of the variables were set to false for the Booleans, and 0 for the others. These initial values match the default values of the variables in the source code. Moreover, as will be stated later, this simplification does not affect the result of the verification.

Although a systematic approach was used in the modeling of the MATLAB code, manual translation always has the chance of introducing modeling errors. However, since the conflicts found in the model were possible to reproduce by simulations of the code, there is at least some level of correspondence.

4. SPECIFICATION

Zita et al. (2017) specifies a property that the LSM shall always do a lane change to the same lane as is requested by the Planner. This was shown not to hold. A good manual fix was elusive, so synthesis is performed to try to patch the issue. The property states that the LSM internal variable direction and the incoming request from Planner may not differ during more than one update event; when a lane change is performed, it needs to accord with the currently active request. Fig. 3 shows $R$, the EFSM model of the lane change request specification.

If two consecutive updates occur where direction and request differs, then $R$ of Fig. 3 will transition from the initial location $s_0$ through location $s_1$ to location $s_2$. If the variables would be equal during one of the events, $R$ would transition back from $s_1$ to $s_0$, or do a self loop in $s_0$. From $s_2$ it is not possible to come back to the initial state, and so $R$ blocks in $s_2$. Verification showed that this blocking state is indeed reachable from the initial state in the global system of LSM|$R$. As described by Zita et al. (2017), a counter-example was given and it could be shown that this faulty behavior was also present in the actual code and could lead to collision.

Even with the initial values of the variables set to distinct values, the problem described by Zita et al. (2017) persists. The same problem can be shown to exist in the minimal example of Fig. 2.
do not allow the input signal request to take any other value than the current value of direction, if those two have already been unequal. For the event \(e_1\) in Fig. 2, this corresponds to having the additional guard

\[(request = \text{ChangeRight}) \lor (direction = \text{ChangeLeft})\]

So, if the Planner issues a request which is unequal to direction and differs from the previous request, then the guard will block the \(e_1\) event. This means that Planner may not issue a request to the left. However, the aim was to patch LSM, not to put new requirements on Planner. Since request is an input from Planner, events \(e_1, e_2,\) and \(e_3\) must always be enabled after an update. Thus, these three events should be uncontrollable for LSM.

By making \(e_1\) uncontrollable, the guards cannot disable its occurrence, but then they cannot prevent request and direction from being unequal either, so the synthesis fails in this case; the result is the null supervisor which signifies that no solution exists.

For the synthesis to become meaningful, the original model must be changed. The original model is a direct translation of the code, so every line of code is represented, but for synthesis what matters is how the inputs behave, how that affects the state, and how the outputs should be set based on the inputs and the state. What is immediately apparent when investigating the plant in Fig. 2 is that there is only one event, \(S_0\text{update},\) which is associated with an assignment of direction, and then it is only by setting direction to the same value as request. Hence, a supervisor has no possibility to change the value of direction during any transition between locations \(S_1\) and \(S_2\). Having identified this, it is possible to add the plant shown in Fig. 4 to the model. Now, a supervisor has access to a controllable event, output\_update, between \(e_4\) or \(e_5\) and update, where the supervisor has the possibility to choose the value of the variable direction. Synthesis of the plants in Fig. 2 and 4, and the specification in Fig. 3 produces a supervisor that assigns request to direction when Fig. 2 is in location \(S_1\) and if request and direction differed during the last update event.

The resulting supervisor makes sure to update direction such that it matches request sufficiently often. However, when analyzing the original model it becomes clear that this is not the behavior sought for. The values of direction and request should not differ during more than one update event, but if this is enforced only by letting direction mimic request we have only achieved a copy of the request variable. Instead, the intention is that if in location \(S_2\) in Fig. 2, direction differs from request, then event \(e_4\)
should be fired; otherwise $e_5$. Thus, by changing the guards for the $e_4$ and $e_5$ events, originating from location $S_2$, to request $\neq$ direction and request $=$ direction, respectively, we can verify that the plant in Fig. 2 complies with the specification in Fig. 3; no need at all for the plant in Fig. 4 or any synthesis. However, this way of correcting a model is not within the scope of synthesis, as synthesis can only remove behavior.

6. CONCLUSION

An EFSM model of a part of the lane change module for an autonomous vehicle with known incorrect and possibly dangerous behavior was examined and subjected to synthesis in an attempt to automatically adjust the model to fulfill the given specification. The standard monolithic supervisor synthesis, the compositional abstraction-based synthesis, and BDD-based synthesis were tried on the model and all three approaches managed to generate identical supervisors but only the BDD-based supervisor, which comes as a set of guards on the events, was useful for determining the behavior of the supervisor.

As it turned out, the supervisor synthesized from the original model put restrictions on how the input data might change, which basically means that the synthesis result places requirements on another module in the system. It seems that, in general, it is very difficult to patch legacy code with the technique attempted in this paper.

The results show that the possibility to effectively analyze the synthesized supervisor is crucial. Otherwise one cannot know whether the resulting constrained system is a plausible solution to the problem at hand. Although the synthesis did not produce usable supervisors, in this particular case the method highlighted an error in the code and a missing requirement.

Research is currently ongoing regarding how to incorporate these types of formal techniques into the daily engineering work flow. With existing legacy code, the main obstacles include how to get the model from the code, and finding the specifications, many of which are not written down explicitly, plus useful means to understand what behavior gets removed by synthesis. For these types of techniques to get industrial acceptance these obstacles must be overcome.

REFERENCES


