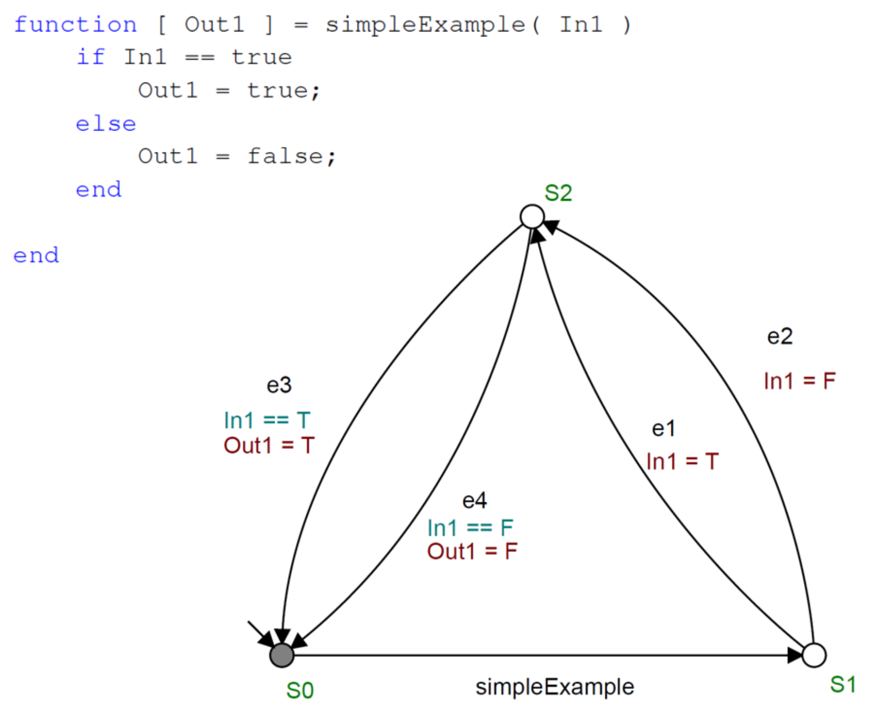
CHALMERS
UNIVERSITY OF TECHNOLOGY



```
function [ Out1 ] = simpleExample( In1 )
    if In1 == true
        Out1 = true;
    else
        Out1 = false;
    end

end
```

Logical Modelling and Formal Verification of Decision and Control Functions for Autonomous Vehicles

Master's thesis in Systems, Control and Mechatronics

PONTUS PETERSSON & ANTON ZITA

EX022/2016

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis EX022/2016

Logical Modelling and Formal Verification of Decision and Control Functions for Autonomous Vehicles

PONTUS PETERSSON & ANTON ZITA

Department of Signals and Systems
Division of Automation Control
Chalmers University of Technology
Gothenburg, Sweden 2016

Logical Modelling and Formal Verification of Decision and Control Functions for Autonomous Vehicles
PONTUS PETERSON & ANTON ZITA

Supervisor: Sahar Mohajerani, Volvo Car Corporation
Examiner: Martin Fabian, Department of Signals and Systems

Master's Thesis 2016:22
Department of Signals and Systems
Division of Automation Control
Automation research group
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Shows a simple function in MATLAB and how this is translated into a state machine using the same method as used in this thesis project.

Typeset in LaTeX
Gothenburg, Sweden 2016

Logical Modelling and Formal Verification of Decision and Control Functions for Autonomous Vehicles
PONTUS PETERSSON
ANTON ZITA
Department of Signals and Systems
Chalmers University of Technology

Abstract

As computer programs are getting more complex, traditional test-based verification techniques might not be enough to ensure correctness of safety critical systems. Formal verification is an alternative approach that can mathematically prove if specifications and requirements are fulfilled by the system. Formal verification is a quite well established tool in academia, but has yet to find its place in industry.

This thesis is a proof of concept where software for autonomous cars in the Volvo Car Corporation 'Drive Me'-project is logically modelled and formally verified. Parts of the module responsible for decision and control of the car behavior are, within the thesis work, modelled by Extended Finite State Machines and verified in Supremica, a tool developed at Chalmers University of Technology.

The result of the verification is that there exist situations where the system violates a desired behavior, and the presence of this could be shown in the actual software. Solutions to correct some of the undesired behaviors have been proposed and the new models have been verified again. Some of the solutions did indeed solve the problem, but some solutions gave rise to new issues, these have not been further investigated.

This thesis shows that formal verification indeed can contribute to the verification of software in real applications. Ways to automatically convert models to code or vice versa should be investigated more closely, but for safety critical parts, manual modeling of the system is still motivated.

Keywords: Discrete Event Systems, Formal verification, Logical modelling, Model checking, Supremica

Acknowledgements

First of all we would like to thank Volvo Car Corporation and Mohammad Ali for giving us the possibility to access and examine the software that this thesis is based on. Moreover we would like to thank our supervisors on Volvo Car Corporation, Sahar Mohajerani and Roozbeh Kianfar for their support during the project, as well as everybody on group 94415 for their curious questions about our work throughout the process.

Last but not least by any means, we would like to thank our supervisor and examiner Martin Fabian for his great support throughout the whole project. We wouldn't have managed so well without you!

Pontus Petersson & Anton Zita, Gothenburg, June 2016

Contents

List of Figures

1
Introduction

1.1 Background

The size and price of transistors, and thereby computational power, have for several decades been falling according to Moore's law. This has paved the way for more and more applications to not only use, but actually rely on computers for to be functioning. One good example of this can be found in the car industry, where computers are taking over functions that were either mechanical, or relied solely on the driver.

The computer hardware itself makes no use without software and, depending on the application, programs have different complexity. Some are very easy programs that are straight-forwardly coded. Others may be much more complex, i.e. where multiple subsystems are involved in pre-processing data that is used in decision making. Even a skilled programmer, well acquainted with the system, may introduce logical errors in the implementation of a highly complex software. Logical errors are of course always problematic, since they risk to ruin the intended function of the program, and it is extremely important to find and eliminate these errors, especially for systems that are safety critical for humans, machinery and environment. This calls for tools to support the programmers.

One way of trying to address the issue of eliminating logical errors is to use formal verification. This means to create a formal model of the system, and use mathematically proven methods to find logical 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 interesting behavior in enough detail.

1.1.1 Usage of formal verification

Formal verification has not yet found its place in industry, and it is still an active field of research. Two examples of the use of formal verification are presented here.

Gadara project

The Gadara project [2] presents a way to automatically analyze and correct source code from a C-program. The Gadara project has its focus in execution of multi-threaded programs, and to avoid blocking in these. The verification is done in the compiler step, and if issues are found the program is automatically corrected. The output executable is then guaranteed to be blocking-free, this without affecting any of the functionality of the program [3].

Verification of Caltech autonomous vehicle

In 2007 Caltech had a vehicle competing in the DARPA Urban Challange [4]. Some of the observations from this project are discussed in [5]. One interesting observation connected to the usage of formal verification concerns how the vehicle handled one of the tasks in the competition, to make a U-turn. The vehicle firmware had been tested both in simulations and in field tests, but at the time of the competition the vehicle had severe problems in making the U-turn. It was later shown with formal verification that the root of the problem was the collaboration of three different software modules that each worked as intended by itself, but when they where connected together the system exhibited unexpected behavior and the U-turn did not function correctly.

1.1.2 The 'Drive Me'-project

Volvo Car Corporation, VCC, have an ongoing project called 'Drive Me'. The intended outcome of the project is that 25 XC90's with autonomous driving capability are launched to VCC-customers in 2017. By "autonomous driving" is meant that the vehicle themselves, in real traffic on predefined roads in the Gothenburg area, make all maneuvers without a supervising driver.

To do this, the vehicles are equipped with a lot of sensors to receive data about its surroundings. This data is then processed on-board, and depending on the current situation the software takes different actions. Since the vehicles are in real traffic, software problems could potentially be disastrous, and great effort is therefore made to verify the correctness of the system.

1.2 Purpose

The purpose of this project is to use formal verification on parts of the existing software for the 'Drive Me' project to show conceptually that formal verification has a role to play in making the software of the autonomous driving XC90's logically correct.

1.3 Objective

The objective of the project is to show proof of concept. What is to be proved is that the use of formal verifications tools on existing code, in the Volvo Car Corporation 'Drive Me' project, could be used to find logical errors. This means that the existing code has to be translated to a language that can be handled by formal verification tools, and thereafter known verification techniques are to be used to find potential logical problems. The tool that is used in this thesis project is the software Supremica [6, 7, 8].

1.4 Scope and boundaries

The complexity of the whole autonomous driving system is high and every detail can therefore not be considered. One of the key elements in the project is to abstract parts of the code where logical errors have the possibility to occur, and to model this in a correct manner. The intention is not to model the whole system, but rather to find a methodology of how to do the translation from MATLAB to a model accurately and to formally verify correctness. It is beyond the scope of the project to do corrections of the verified software, but possible solutions might be discussed.

2
Theory

This chapter aims to explain and describe the most important basic theory that is used in this thesis project. The reader well acquainted with discrete event systems theory might find some parts of it too elementary, but the purpose is to give a short summary of what is needed to follow the reasoning in coming chapters. First, sections 2.1 and 2.2 give a short introduction of Discrete Event Systems and Supervisory Control Theory. The final section, 2.3, introduces Formal Verification, why it is motivated and what it can be used for.

2.1 Discrete Event Systems

The term Discrete Event Systems, DES, refers to systems which can naturally be described with *states* and *events*. A DES is a discrete-state and event-driven system, meaning that its state evolution depends entirely on the occurrence of asynchronous discrete events [9].

An example of a system that could be viewed as event-driven would be a switch changing from being off to on, which is referred to as an *event*, and this makes the system do a *transition* from the *state* 'Idle' to 'Working'. This example is shown in Figure 2.1. An important property of DES is that the event is assumed to happen instantaneously, it is not a continuous change.

2.1.1 Finite state machine

There are different modeling formalism to model DES, for example formal languages and Petri nets [9]. However, in this thesis Finite State Machine, (FSMs), are used, as they are intuitive to work with. Moreover, the software used for verification, Supremica, works well with FSMs. Characteristics of FSMs are given here.

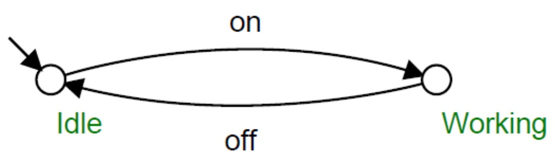


Figure 2.1: An example of a simple discrete event system.

States

As the example with the switch tried to describe, states are used to describe the system status, and they may represent both physical or abstract properties.

It is sufficient to have states which only define where the system is, these can be called non-marked states, but to give the whole system description more meaning some additional properties are defined for the system: the initial state, marked state(s) and forbidden state(s). The initial state is the state where the system starts. Marked states are states that are for some reason desired to be reached. They may for example represent a place where the program is finished. It is possible for a state to be both initial *and* marked. This would then represent that it is desired for the system to come back to where it started. Lastly, a forbidden state is a state that is for some reason not desired to be reached. It could for example be a state where something has gone wrong.

Transitions and Events

A transition is used to describe the move from one state to another, and is said to *fire* when the transition from the source state to the target state takes place. There can be multiple transitions in and out from a state, and which one that is fired depends on the *event* and the *guard* that is associated with it.

An event labels a transition, i.e. the event makes it observable that the transition has been fired. To continue with the example in Figure 2.1, this would be that the event puts a name (*on*) on what happens when the switch goes from open to closed.

2.1.2 Extended finite state machine

A more compact state machine representation than the FSM would be to use Extended Finite State Machine, EFSMs. The EFSM introduces the use of discrete, bounded variables and these can be used in *guards* and *actions*, see description of these below. The use of variables gives as well the possibility to represent states in a slightly different way than in a FSM. In an EFSM is a state represented with a *location* together with variable values, i.e. defining more variables does implicitly define more states no matter how many locations there are. Consequently, in an EFSM there are as many states as there are combinations of locations *and* variable values. The mapping from EFSM to FSM is therefore simple, for each location is all combinations of variables taken to make states, the variables is only a convenient way of packing that information. A benefit of introducing variables is that this makes the model closer to how computer code with statements and assignments work. Furthermore, this more compact graphical representation is sometimes preferred since it is easier to overview.

Guards

Guards are used to determine which transitions that can be fired at a given state. A guard must be true in order for the transition to be allowed to be fired. Guard expressions are logical expressions such as $AND, OR, <, >$ etc, and typically correspond to what would be used in *if-statements* in most coding languages.

Actions

An action is triggered when its transition is fired. An action writes a value to a variable, and this corresponds well to what an assignment would be in most coding languages.

2.2 Supervisory control theory

Independent of if FSM or EFSM are used as the modelling language, DES models are usually regarded as either *plants* or *specifications*. Plants describe the possible behaviour of the system, and can be seen as the event generator, and specifications describe the desired behaviour of the system. Supervisory control theory, SCT, is a framework that is used when supervisors, which are a type of control function, are *synthesized* from plants and specifications. These supervisors are such that the specifications are always fulfilled [10]. Since formal verification is one important part of the SCT, a brief review of the whole framework is given here.

2.2.1 Plants and specifications

Plants are models of the system of interest and should describe the entire possible uncontrolled behavior. The plant can have *controllable* and *uncontrollable* events. This naming is descriptive; controllable events represent system occurrences that can be controlled, such as when a switch is put from off to on. An uncontrollable event would be if for example the power gets cut off when the switch is on, i.e. an event that is not possible to control within the system.

Specifications are models of the desired behaviour of the system, and could be considered to be the algorithms that the system should follow in order to work in a safe manner. The idea of marking and forbidding states introduced in section 2.1.1 becomes more meaningful here: the marked states in plants and specifications are the states that it is for some reason desired to reach, and the forbidden states are desired not to be reached.

2.2.2 Synchronisation

When two, or more, models are available it is possible to combine them into what is commonly referred to as the *synchronous composition* or simply *synchronisation*. The synchronous composition combines the models into one *monolithic* model.

Events that only appear in one of the state machines is considered to be a *local* event. This means that transitions with this event label can happen whenever the state machine it belongs to is in a state where this transition is possible. If events appear in more than one state machine, they are considered to be *shared*. That means that all state machines have to be in a state where this event could be taken, otherwise the event is disabled. The benefit of this will be more obvious when the supervisor synthesis is introduced.

If all events are local, the state space will grow exponentially with the number of states in each of the models that take part in the synchronous composition, which will rather quick lead to what is commonly referred to as the *state space explosion*, more about this in Section 2.3.2.

2.2.3 Blocking and non-blocking

When the synchronisation has been performed, the new component might be more restricted in its behaviour than what was possible for each of the systems themselves. This is because of the fact that to fire a transition labelled with a shared event all components that are part of the synchronisation have to be able to do this event. And, since the whole purpose of this framework is to come up with supervisors that make sure that the system is only doing what we want it to, this makes sense, i.e. the supervisors can prohibit the system from doing unwanted behaviour.

If the result of the synchronisation is a model that has a state from where no marked state can be reached, this is referred to as a blocking system, and this is in general not desired. All models that are part of the synchronisation have to be in a marked state to make the synchronised state marked, so if a system is blocking it is not possible to fulfill the specification.

The desired outcome is naturally the opposite of blocking, non-blocking. That means that the synchronous composition can always reach some state that is marked in both the specification(s) and the plant(s).

2.2.4 Controllability

Transitions that are labelled by uncontrollable events cannot be influenced by the supervisor. The SCT framework aims for making a supervisor that can make sure the system does what is desired from it. Hence, a supervisor may not depend on influencing any uncontrollable event when controlling the system, the supervisor must be *controllable*. This means that whatever the supervisor allows the controlled system to do, it must always allow the possible uncontrollable events to occur. This needs to be guaranteed when *synthesizing* the supervisor.

2.2.5 Synthesis

The purpose of using the SCT framework is now reaching its final step, to synthesise a supervisor that makes sure that the plant fulfills the desired behaviour. To do that, it is first to be proved whether the plant models satisfy the specifications using formal verification. To be more clear: the verification checks that the system cannot, by following the specifications, be blocking or uncontrollable. Formal verification is more elaborately discussed in Section 2.3.

If the specifications are fulfilled, the synthesis is finished and the achieved supervisor is the same as the synchronous composition of the models.

If specifications are violated by the system, the supervisor is found by iteratively pruning the synchronous composition until the specifications are fulfilled. This iterative way is important, because the SCT is based on the assumption that one want to maintain as much functionality as possible, i.e. make the supervisor *least restrictive*.

2.2.6 Summary: Supervisory control theory

In this section have an overview of what the SCT is based on, and how the synthesis of the supervisor is performed. As already stated, the SCT is more extensive then what is needed in this thesis project. In this thesis project only the use of formal verification is needed, this to do non-blocking verification. Still, since plants and specifications are constructed in the same manner as described by the SCT, and the verification is actually part of the SCT algorithms, it is motivated to present the whole framework, even though it is not done profoundly.

2.3 Formal verification

Many times one can claim that the goal when making a system is to make it such that it will do what its is supposed to do, and nothing more than this. Whether these properties are fulfilled or not can be checked in several ways, and the most obvious alternative is to *test* it. Since testing is the most well known procedure this has by far the greatest usage, and in some applications testing might be enough to ensure that the outcome is as intended. However, in many cases testing can never ensure the correctness of the system. This section introduces *formal verification* that mathematically proves the correctness of the system. The term 'system' often refers to computer programs, and even though the tool itself might be applied to check logical correctness of any system, it will further on be assumed that computer programs constitute the system of interest to verify.

2.3.1 State of the art

Formal verification is a relatively new area closely connected to software development. Just a couple of decades ago computers were several magnitudes less powerful than what is common today, and this restricted the size and complexity of computer programs. Understanding and testing code could then be enough to verify correctness. Increased computing power have cleared the way for bigger and more complex programs, and this has meant that it is often not possible to fully understand where the program might go wrong, and what consequences that might have for the further execution. Furthermore, the increased size and complexity leads to the case that it is not possible to make exhaustive testing due to the high, or even extreme, workload this way of verification would lead to. To test all different combinations that a program can execute, 100% test coverage, using automated testing is impossible already for rather small programs due to computational power limitations, it would simply take too long time [11].

With this in mind, what testing commonly offers is therefore to expose the real system to a limited number of situations and evaluate if the behaviour is correct or not. That means that there is some sort of *specification of what the desired system behavior is.* Depending on the application this might be enough, but especially for safety critical systems this might not be the case.

This is where formal verification comes in. With the help of a model of the system, formal verification can guarantee that a specification is either fulfilled or

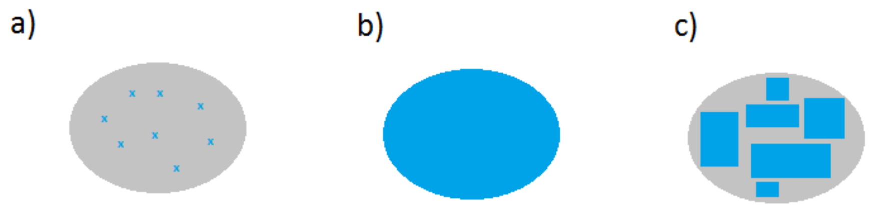


Figure 2.2: Comparison between testing and formal verification. (a) Testing. (b) Ideal Formal Verification. (c) Real world formal verification.

not, under the assumption that the model is correct. To be more precise: in contrary to testing, *formal verification can show absence of errors*. Depending on the la till approach, specifications can either try to address *wanted* or *unwanted* behaviour, this makes no difference for the method itself. As mentioned already, some sort of specification of the system behaviour is needed in testing as well, so the use of formal verification does not really require anything extra, one still *have to know what the system is intended to do*, and in many cases the exact same type of specifications are needed.

The difference between testing and formal verification are illustrated in Figure 2.2. As can be seen in (a), testing gives the possibility to have individual test samples from the whole state-space. Each sample gives information about that specific situation, and therefore the question of when to stop testing is of great importance, i.e. when is the coverage sufficient?

The ideal case would of course be to make sure that all logic is tested for all possible scenarios of the program, which theoretically is possible to do with ideal formal verification as illustrated in (b). Because of the need for specifications to check the logical consistency this is often hard to reach, as illustrated in (c). Even if the model cannot reflect all aspects of the program, formal verification can still verify large parts of the programs state-space.

2.3.2 State-space

As said in Section 2.3.1, one of the big problems when working with testing is that in many cases the time to test the program behaviour with all combinations of input signal values could be several years or more, even for not very complex programs [11].

The purpose of using formal verification rather than testing is that it makes it possible to guarantee that the whole domain is covered in an efficient way. To be able to do this, formal verification requires that each and every possible *state* of the program is represented and checked for consistency. The main underlying problem, that all combinations of input signal values could be enormous, will though still remain, but will show up in another shape, commonly referred to as *state-space explosion*.

This might need a little explanation to be fully understood. Let us say that a program, for some reason, reads the values of three sensors measuring the temperature of liquid water. For the sake of simplicity we assume that the sensors will only have integer resolution, and since it is predefined that the water will be liquid we only care about values between $0 - 100°C$. If the program for some reason has to be tested for its output depending on the input values, this will lead to $100 \cdot 100 \cdot 100 = 1,000,000$ possibilities. Add one decimal to the resolution for all the sensors, and this has to be multiplied by another 1000 possibilities, making one billion combinations in total, i.e. the state-space have in some meaning exploded. In addition it should be noted that this is for a combinatorial system, where the input combinations directly correspond to the internal states. In sequential systems, where the internal states hold more information than simply the combinations of inputs, this problem is significantly magnified. Assume for instance that the program was to keep track of the trends of the sensors, and behave differently depending on the current trend. Then extra information about the trends will be needed, which would require even more states.

To handle this, smart algorithms has been developed with the purpose to defeat the state-space explosion. Since this project uses *Supremica* as its tool for making the formal verification, some of the algorithms used are described in Section 2.3.6.

2.3.3 Modelling and abstraction

In order to do formal verification, a model of the system to be verified is needed. The mathematical model can then be analysed and verified with a tool that uses formal methods. Supremica can handle both FSM and EFSM models, and therefore the program or system has to be modelled as one of these types.

Behaviours

In general terms modelling is the same thing as making an abstraction of the real system, and that the purpose of doing the model is to keep, and analyze, the behaviors of interest. This applies here as well as in physics. Now, for logical modelling the ideal case is when a system A can be translated to a state machine model A' without loosing or adding any behaviours. This model can then be used to verify the true system. This is illustrated in Figure 2.3 (a). If A' satisfies a specification then also A satisfy it.

Another approach would be to as in Figure 2.3 (b), where A' is modelled as an over-abstraction, which means that A' cover not only all behaviours of A, but even more behaviours. This type of models can be obtained when applying abstraction methods such as *generalization* [12] or *state merging* [13]. This will give the model strictly more behaviours than the original system has, but still keep all the behaviours of the original system. This model can also be used to verify the true system, and if A' satisfies a specification then also A satisfies the specification. However, if A' violates the specification one can say nothing about A. This since it cannot be distinguished if the behaviour which violates the specification only is in A' or in both A and A'.

Yet another alternative is found in Figure 2.3 (c) where an under-abstraction of A is shown. That means that A' is obtained from A by restriction. This will give the model strictly less behaviours than the original system has. If A' violates

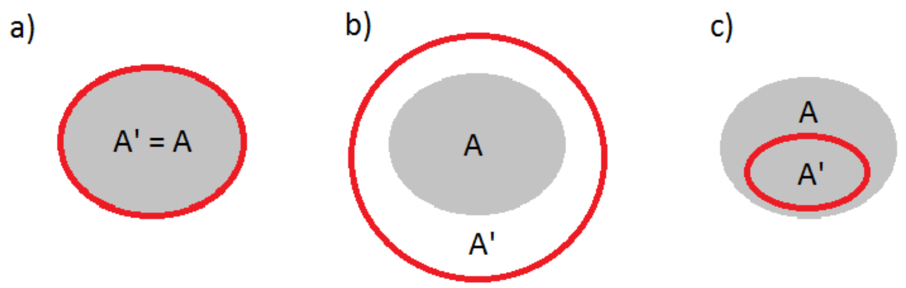


Figure 2.3: Relation between the true system A (grey) and the model abstraction A' (red). (a) Modelled without loosing or adding any behaviours. (b) Modelled as an over-abstraction. (c) Modelled as an under-abstraction

a specification then A will violate it as well. But if A' satisfies a specification one can say nothing about A; thus, in this case the model cannot be used to verify the correctness of the true system.

2.3.4 Possible logical errors

In this thesis project, there are three types of logical errors that will be looked for. The first two are *deadlocks* and *livelocks*, which are similar in the way that once they occur the program is in a state from where a marked state cannot be reached. These types of errors are in the literature commonly referred to as *blocking* properties [9]. These errors can be found in the model by just verifying the model without any need of specifications. The third type of logical error that will be looked for is *illegal behaviors* [9]. This is done by describing the intended behaviours or properties of the system as specifications. These specifications are then synchronised with the model into a new synchronised model. If a specification is violated by the model this will result in a deadlock or livelock in the synchronised model. So the third type of error will be included by looking for the first two types.

Deadlock

Deadlocks are probably one of the most well known logical errors. When a deadlock occurs, execution cannot continue. This happens when there are no outgoing transitions from a state. However, reaching a deadlock is not necessarily a bad thing, and most often the termination of a program is modelled as reaching a deadlock [14, p.143]. In Figure 2.4 (a) two deadlock states are shown. Since the left state is marked, the deadlock here is allowed behaviour, while the right state is undesired behaviour since no marked state can be reached from here.

One possible situation where a deadlock can occur is if two users try to use the same resources at the same time, when multiple use of this resource is not permitted, and each user has to ask for access before use of the resource, as in the following example: resource A is used by user 1 and resource B is used by user 2 and both users wait for the other to finish the use of its taken resource to be able to continue. Since the other resource is already used by the other user, a deadlock has occurred and no further continuation is possible. This would be an unwanted logical error.

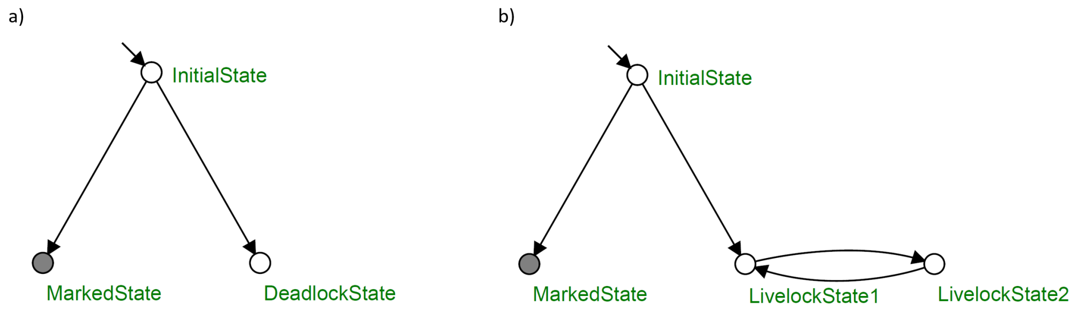


Figure 2.4: Two examples showing the difference between deadlock and livelock. (a) Shows a deadlock state from where a marked state (gray) cannot be reached. The marked state is also a deadlock state but is wanted behaviour. (b) Shows a livelock.

Livelock

Livelock is also a program behaviour where the program will not be able to finish, but works in a quite different way from deadlock. If a program contains a possible livelock, this means that once it has reached this it will never be able to reach a marked state. In contrast to deadlock, execution is possible in a livelock situation. However, the execution will eventually lead back to a state it has already visited, and thereby go on in a never ending series of execution [13]. The difference between a deadlock and a livelock is illustrated in Figure 2.4.

2.3.5 Synchronisation results

The term synchronization refers to combining two, or more, components of the system into a single component. Synchronisation makes no difference in what the model is representing; it may be both plant models or specifications, it makes no difference for how the synchronisation is done or the result.

When synchronisation has been performed, it is possible to examine the result and what properties the single component has, for example if there are deadlock states. For multi-threaded and concurrent systems it could be of great interest to synchronise the plant models only, since there is an increased risk of having deadlock states in these systems, especially if priorities for the different tasks are implemented. It may be hard to find these types of errors without some sort of verification tool, because the logic could be perfectly fine in each part of the program and the problems arise when the parts interact with each other. Compare to the case with the Caltech autonomous car in the DARPA challenge, described in Section 1.1.1

Another way of entering deadlock states is to make specifications that define that if a certain sequence of events occurs, this will lead the specification state machine to end up in a deadlock. If the plant model allows this specification to be fulfilled, the synchronised component will as well possess this deadlock state.

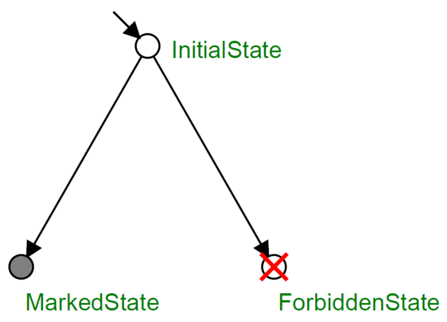


Figure 2.5: An example of a Supremica representation of a FSM with an initial state, a marked state and a forbidden state.

2.3.6 Supremica

Supremica [8] is a software developed at Chalmers University of Technology, that can be used for formal verification. In Supremica, models are built in a graphical *editor* as FSMs or as EFSMs. These models can be simulated and visualised in the *simulator* environment and by using the *analyser* feature the models can be verified using a wide range of verification algorithms [7]. When the verification tool is used, Supremica gives either a message that the specification is fulfilled, or a *counterexample* when the specification is violated. Supremica uses efficient algorithms to limit the state-space of the analyzed model, while still making sure that the behaviours are caught, and this allows large and complex systems to be verified. Supremica implements FSM and EFSM in the way these are described in sections 2.1.1 and 2.1.2. A description of how this is implemented can be found in this section.

States and state names

In Supremica, the symbol for a state is a circle. A marked state is grey-coloured and a forbidden state is over-crossed by a red cross. The initial state has a small arrow pointing to it. How these properties are shown in Supremica is shown Figure 2.5. All states must be uniquely named, but the names themselves are only a help for the user to keep track of what each state represents in the modelled system. It might as well be beneficial to use different state names when synchronisation is performed to make the interpretation of, for example, a counterexample more straightforward.

Transitions and events

Transitions between states are represented by arcs in Supremica. The arcs are directed, so to be able to go freely between two states two transition arcs are needed.

As said, events label transitions, i.e. the event makes a transition observable. The event-labels are used by Supremica to determine if events are shared between different state machines. The event-label also holds information that is valuable for the user since it eases the navigation through the model. In the case of a blocking system, a counterexample of which chain of events that led up to the blocking is given, and it is therefore beneficial to have well chosen event-labels.

Variables

Supremica implements variables of the types integers and enumerations. A variable has to be given a range of its possible values, and initial value(s). Variables are used in guards and actions of EFSMs.

Guards and actions

Guards and actions are implemented in Supremica as described in Section 2.1.2, and are thus connected with the transitions. Guards are marked in blue and actions are marked in red in Supremica.

Mathematical expressions can be used in guards and actions when using integer variables. Addition, subtraction, multiplication and comparison operators are examples of the expressions allowed.

2.3.7 Supremica Algorithms

In this section, some of Supremica's algorithms for synthesis and verification are described. As mentioned before, the *state-space explosion problem* is the main limitation of how big systems that can be verified using formal verification. When using an explicit representation of all states, as in the monolithic approach, the state-space grows exponentially and thus it quickly grows too big to handle in terms of time and memory, already for rather small systems [15]. Still, for small systems it is possible use a monolithic approach and Supremica has support for this. But in order to verify and synthesize larger systems, Supremica has other algorithms to represent the system in more efficient ways, while the result of the verification still can be maintained. Some of the implemented and used approaches to represent the system in a compact way are described here.

Compositional

One approach is to make use of the *compositional* algorithms. This approach is proper to use if the system is modelled in a *modular* way, meaning that the system consists of several plants and supervisors [6]. The compositional method simplifies and abstracts each module before synchronisation in a way such that only the shared events and the ones needed for verification are represented, and the local parts are abstracted away. By doing this for each module the state-space and number of transitions can be kept low in the synchronised model. The final verification result will still be the same as it would be with a monolithic approach, but the system is now represented in a much more compact form. How the compositional algorithms works are described in detail in¨[16].

BDD

The BDD algorithm make use of binary decision diagrams, *BDDs*, which is a way to symbolically represent digital functions as tree structures. BDDs were first introduced by [17] and was also found suitable to be processed by a computer by the same. The power of the BDD representation lies in the fact that it can be reduced according to certain rules often yielding a very efficient data representation [15]. Under the right conditions, BDDs can reach an exponential compression of the state-space [18]. An example of such a compression could be if a result of a branch always ends up with the same result, this branch can then be replaced with just this single result.

An FSM is represented and defined by its transitions, with the necessary information about between which states and what event. This set of transitions are then reformulated and binary-encoded, and by this the whole system can be represented as a BDD, with all its features. Then, in this BDD-domain, the SCT framework can be applied and be used to do formal verification. By using this algorithm, the limitations in terms of system size due to the state-space problem can be pushed and by this be able to verify larger systems than would have been possible with a monolithic approach.

3
System Description

In this thesis, parts of the lane change algorithm for the Volvo Car Corporation 'Drive Me' project has been verified with the use of formal verification. To be more specific, a part of the lane change algorithm called *Lateral State Manager* is verified, this since it fits well into modelling as a Discrete Event System. The Lateral State Manager is part of the class *Planner*, which will be described in some detail in this section. The implementation of the lane change algorithm is written in object oriented MATLAB-code and simulations are made in the MATLAB/Simulink environment.

The main reason that the choice fell on verifying the lane change algorithm was that it was the most developed function at the start of this project. However, it is important to keep in mind that the project is to do proof of concept, and that any function with logic could have been chosen.

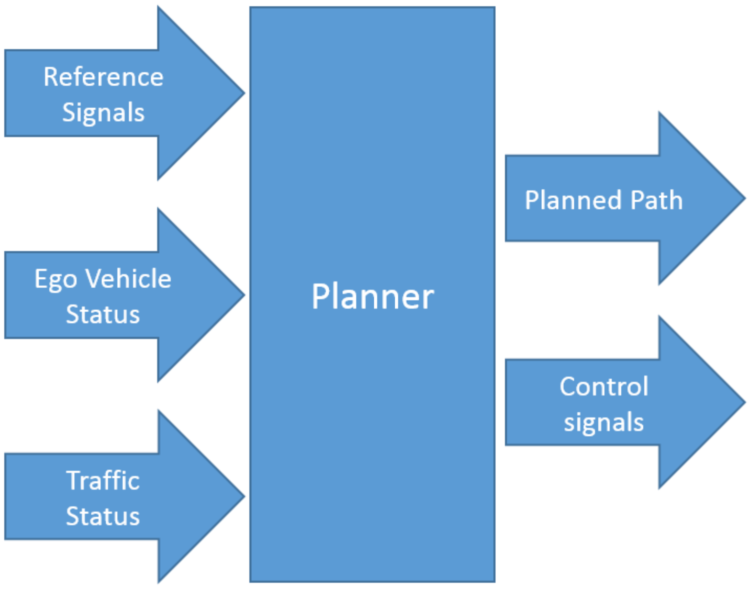


Figure 3.1: A simple illustration of the inputs and outputs of the class Planner, which is a part of the lane change algorithm.

3.1 Planner

The lane change algorithm is implemented with the use of several classes, all with different responsibilities. In this project, the class *Planner* is considered. Planner has the responsibility to decide and control how the lane change should be done. The Planner is updated with the current status of the vehicle, surrounding traffic situation and current reference signals at a high frequency. The reference signals hold for example the current lane change request. With this information, the Planner should return a path and required control signals to make a lane change in a safe and efficient way. A simple illustration of the inputs and outputs to the Planner is shown in Figure 3.1. Since the task of this class is to calculate a path for the current inputs there is, with some exceptions, no need to use data from the last update iteration.

3.2 Lateral state manager

Most of the execution in the Planner only relies on current input data, and does not use preprocessed data from memory. This so, since the Planner should always take the decisions of what to do for the current situation. But, there are parts that have memory and one such part is the Lateral State Manager, which is a state machine. This state machine keeps track of where in the process of the lane change the car is, remembered from the last update iteration. The state machine is illustrated Figure 3.2.

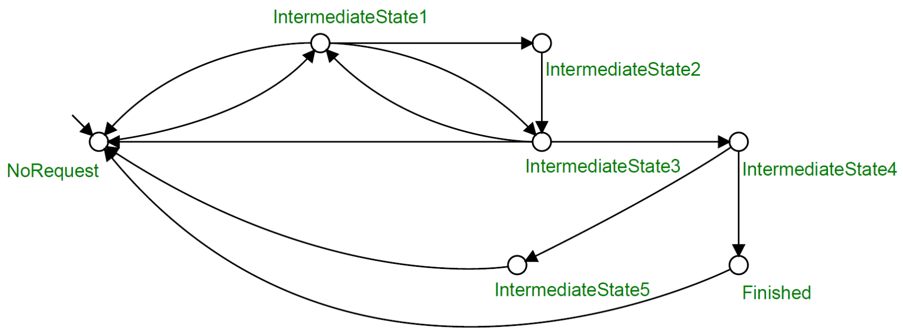


Figure 3.2: A simple illustration of the Lateral State Manager and its seven states. This state machine keeps track of where in the process of the lane change the car is.

The state machine consists of seven states, where two of them are 'NoRequest' and 'Finished'. The other five states will be called 'IntermediateState' due to anonymization. Each time the Planner is updated the resulting outputs depend on what state the state machine is in. Depending on the inputs, the state will remain the same or be changed to another according to Figure 3.2. Note that at most one transition can be fired each time the Planner is updated.

When no lane change is requested the state should be 'NoRequest', this is also the initial state when the Planner is instantiated. Once a request comes, the state is changed to 'IntermediateState1', and from there to either 'IntermediateState2',

’IntermediateState3’ or back to ’NoRequest’ depending on the situation. Finally, when the lane change is done the state will be ’Finished’ and then, in next iteration, make a transition to ’NoRequest’.

3.2.1 Code implementation

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

The Lateral State Manager consists of three different types of methods. The first type has only one method, which is called ’updateState’. It is called from the Planner every time the Planner is updated. The purpose of the ’updateState’ method is to call the current state’s ’during’-method, which is the second type of method. The ’during’ methods, which are tailored for each state, are where the decisions are taken and different parts of the code are executed depending on the inputs. Before the ’during’ method terminates, it will either change the state to another or keep the current state. The latter alternative means that the same ’during’ method will be called in the next update. If the state is changed by the ’during’ method, this will also call the new state’s ’enter’ method, which is the third type of method, before ’updateState’ is finished. In contrast to the ’during’ methods which are executed repeatedly while the state stays the same, the ’enter’ methods are executed only once when the transition into the state occurs. This is illustrated as a flow chart in Figure 3.3.

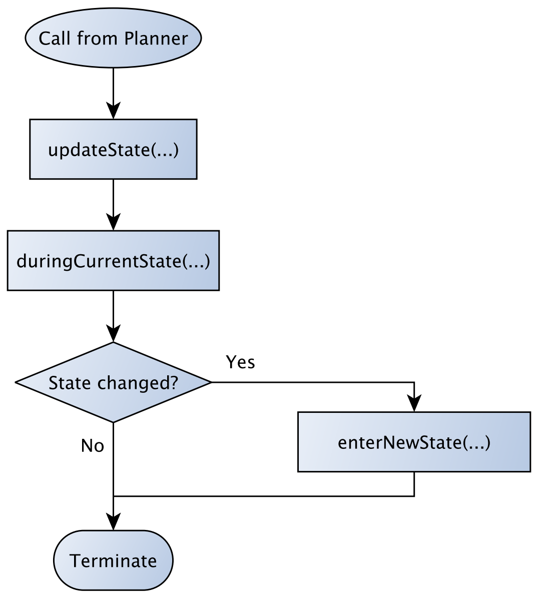


Figure 3.3: Flow chart showing the execution sequence of methods used in Lateral State Manager.

4
Methods

To do formal verification it is fundamental to have a *model* of the system to be verified, and *specifications* of what the verification should check for. Since the modelling and the verification in this thesis are made with the software tool Supremica, this is reflected in the working approach, but many of the methodologies will work with other software and software tools as well.

When it comes to the verification, this is done easily in Supremica, and examples of how are shown in this chapter. If the system is proven correct then the verification is completed. But, if the system is blocking, Supremica gives a counterexample that shows what sequence of events that violates the specification. The model can then be corrected, and a new attempt to verify the model can be done; this process is then iterated until all issues have been found.

4.1 Modelling

As described in Chapter 3, a state machine implemented in MATLAB-code is to be verified. As mentioned before, the used approach is to make the model as simple as possible, while still capturing the behaviour. Since the system of interest is already a state machine it is well suited to be modelled in Supremica. Nevertheless, the steps to make a useful model from the code are described in this section. As an introduction an example of how MATLAB-code can be converted to a Supremica state machine is shown.

4.1.1 Modelling MATLAB code in Supremica

Let us consider the MATLAB function *simpleExample* in Figure 4.1 and the Supremica model this is translated to in Figure 4.2. The MATLAB-code takes the input parameter *In1* and depending on the value of *In1* it will return either *true* or *false*. A quick look at the Supremica state machine shows that the state machine consists of three states and five transitions, with guards and actions.

The initial state is 'S0', which relates to the MATLAB-code in the sense that the function has not been called yet. First when the function simpleExample is called, represented by the event *simpleExample*, a transition to state 'S1' is taken. The first statement in the MATLAB-function is an if-statement that checks the value of the input parameter *In1*. It is here important to note that this implies that *In1* is a Boolean, that can be either true or false, since the type affects the modelling. Since it is a Boolean, the variable *In1* can be assigned one of its possible values value using

```
function [ Out1 ] = simpleExample( In1 )
    if In1 == true
        Out1 = true;
    else
        Out1 = false;
    end

end
```

Figure 4.1: MATLAB-code for a function namned simpleExample

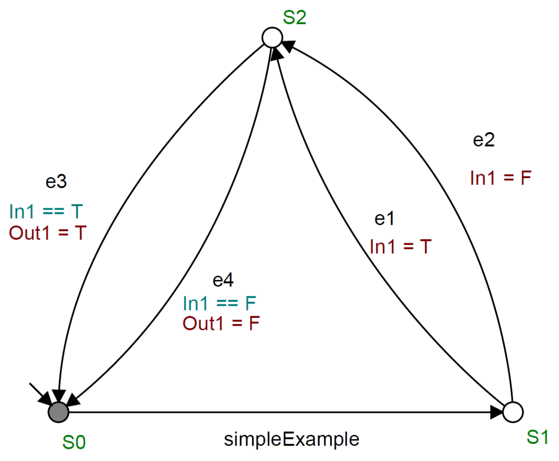


Figure 4.2: Supremica model of the MATLAB-code for simpleExample

only two transitions, *e1* and *e2*. These two transitions do not have guards, because from the function perspective both of these values are as likely to appear. Now, in 'S2' *In1* does have a value, and now the if-statement and else-statement can be implemented with just two transitions back to 'S0'. The guards on the transitions match the statements exactly, and depending on the value of *In1*, *Out1* will either be true or false when the state machine is back in its initial 'S0' state.

The work with making the model will be further discussed in the coming sections, what this example shows is just how the logical behaviour of the code can be represented as an extended finite state machine in Supremica.

4.1.2 System definition

To be able to make a model it is necessary to define what should be part of the model, and what should not. This thesis considers only the Lateral State Manager, as it is implemented in MATLAB. This means that input parameters are considered to be able to take all possible values allowed by its type. More precisely, if there is logic in other parts of the Planner that somehow makes the input signals to the Lateral State Manager take specific values, in such a way that two or more variables are coupled, this is not represented in the model.

However, the initial values of the variables in the Lateral State Manager are considered to be known. This, since those values are well defined in the constructor. When the Lateral State Manager is initialised, its variables are assigned values, among them, the variable *State* is given the value *NoRequest*. This value is used to define the initial state of the EFSM, since Supremica requires a well defined initial state.

This defined system will from here on be called S and is illustrated in Figure 4.3.

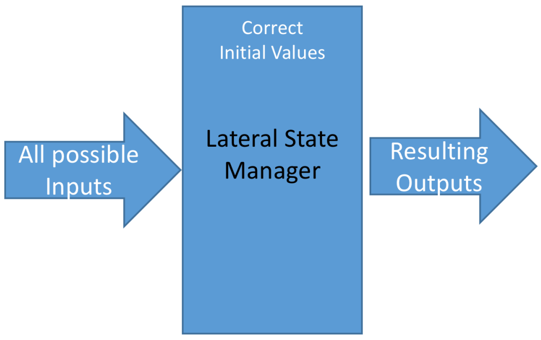


Figure 4.3: Illustration of the defined System.

4.1.3 System to State machine

It is now clear what the system of interest to model is, and the next step is to start the work of representing it as an EFSM in Supremica. As shown in the introductory example, logical tests using Booleans are easily implemented in Supremica. But code can consist of other, more complex, parts that cannot directly be translated into EFSM. In this section, some techniques of how to abstract such code will be discussed.

The parts that do not fit into the DES-modelling, but still interacts with the model must of course be represented in some way to catch the system behaviour. One such example would be mathematical expressions leading to, possibly, continuous results. But, if the value from the operation is used to make comparisons, a variable could be introduced in the EFSM to represent the result of the comparison, true or false. Thereby, the logic can still be verified. An example of this is the function *abstractionExample* in Figure 4.4, where a division of two variables, X and Y is compared with the value of a third variable, Z, and the comparison determines the further execution. The same logical behaviour can be represented with a variable XYlessZ, that can be either true or false. In Figure 4.5 is shown example of how this can modelled in Supremica. This abstraction of the mathematical expression gives the same models for abstractionExample and the simpleExample, discussed in Section 4.1.1.

```
function [ Out1 ] = abstractionExample(X, Y, Z)
    if X/Y < Z
        Out1 = true;
    else
        Out1 = false;
    end
end
```

Figure 4.4: MATLAB-code for a function namned abstractionExample

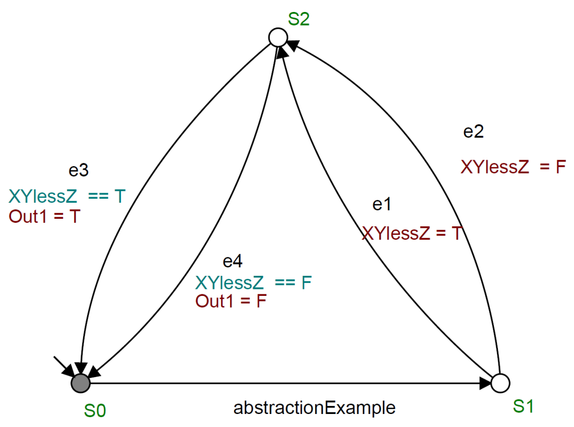


Figure 4.5: Supremica model of the MATLAB-code for abstractionExample

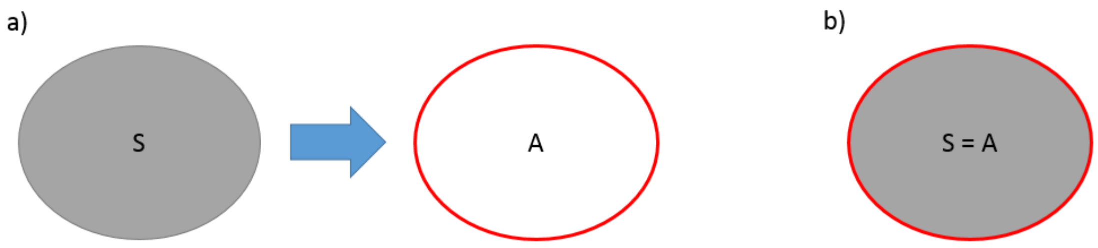


Figure 4.6: (a) The defined system S, is translated into A, a EFSM Model. (b) All the behaviours of S are also in A, no less, no more. The defined system's domain of behaviours are here illustrated with a gray ellipse. The model's behaviours are illustrated with a ellipse but with a red outline.

4.1.4 A becomes A'

In Section 4.1.3, the process of how the original system was abstracted to EFSM *without altering the behaviour of the system* was described. In this section, over-abstraction that *enlarges* the domain of the system to be verified is discussed. This is done in order to check whether the system is robust against usage not expected when the system was designed. The used technique to do this over-abstraction is generalization. This means that instead of modelling A, a less restricted version A' is modelled. The generalized model can still be used to verify the system, since it contains all behaviours from A, as was explained in Section 2.3.3. The behaviours of A and A' will relate to each other as illustrated in Figure 4.7. One of the obvious backsides of doing this generalization is that it might lead to false positives, that will be discussed in Section 4.3.1.

The generalization technique used on Lateral State Manager was to allow more initial states than the constructor actually implemented. The initial values of the variables were allowed to take all possible combinations of values. How the model A' differs from A is illustrated in Figure 4.8.

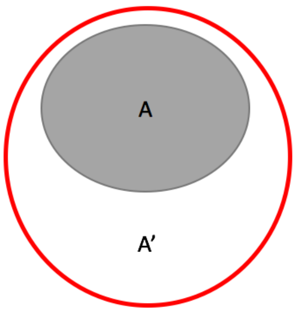


Figure 4.7: Generalization. In order to check if the system is robust, a generalization of the system was done.

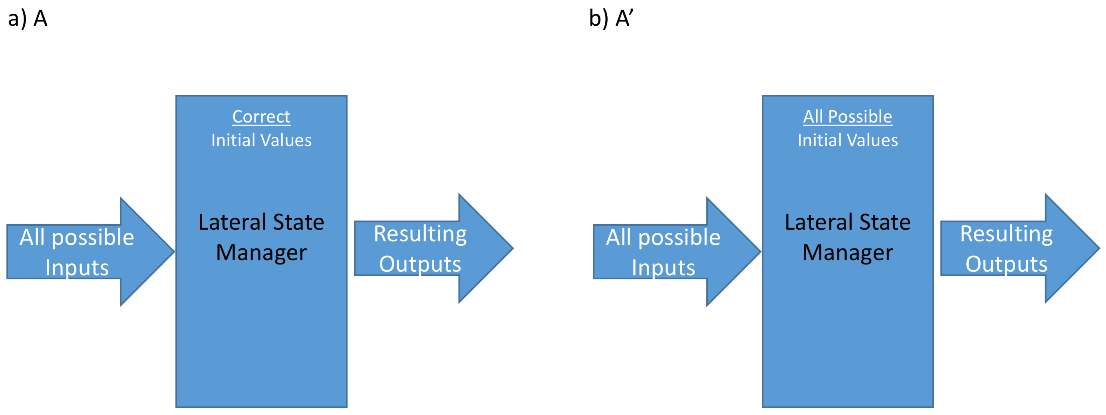


Figure 4.8: Generalization by allowing all combinations of initial values. (a) Model A exactly describe the behaviours of the system, without loosing or adding any. (b) The generalized model A' has all behaviours from A, but also additional behaviours due to generalization and over-abstraction.

Generalization using all combinations of inputs

As mentioned in Section 4.1.2, the modelling of the Lateral State Manager was delimited in such a way that no coupled input signals are considered. What this means is discussed more closely here.

In many cases a system handles inputs and, depending on the inputs and the systems current state, the system shall give a result as output. If the system has multiple inputs there might be some combinations of these variables that are not possible in reality. One could argue that in order to test the system properly it should only be tested with only the possible combinations of inputs. But in order to have correct inputs, one would need a perfect model of the surrounding systems' behaviours. Somewhere there has to be a delimitation of what to be included in the model, and a solution to this is to allow all combinations of inputs to the model. The price paid is that the verification might find false positives.

4.1.5 Modelling Techniques

To make the model intuitive to understand for someone that has access to the underlying MATLAB-code, the focus is in this thesis put on to get a model as close as possible to the MATLAB-code. The example in the introduction of this chapter gave a taste of how the mapping between the MATLAB-code and the Supremica model is done. In this section, some embroidery of these hands-on methods to translate and MATLAB code into state machines is described.

Implementation of MATLAB-functions in Supremica

In the example in figures 4.1 and 4.2 it can be seen that one way to keep track of if a function has started executing is to have one transition, without guards or actions, that has the same event label as the function name. This implementation makes it clear that a function has started its execution, and also that a new function call can be done first when the state machine is in its initial state.

This way of modelling requires that great care is taken so that all code can be uniquely connected to transitions in the model, i.e. that there is no possibility to traverse the model without passing the transitions labelled with function names in the same order as the real program invokes the respective functions.

Elapsed time Since the Lateral State Machine in MATLAB is updated with a fixed frequency, and the function updateState is called each and every time, this event could be used to keep track of how many time steps that has been taken. The elapsed time is simply the number of times updateState has occurred multiplied by the time period.

Variable values and update of input parameters

In a Supremica EFSM, the variables can get their values in two ways; either the value is initialised, and as said Supremica actually allows multiple initial values, but at least one is needed, or the value is updated by actions on transitions. The example in figures 4.1 and 4.2 is therefore revisited to emphasize that a system that uses the value of an unknown input signal in its execution requires the use of variable assignments on actions to make the model accurate.

To be able to mimic the behaviour of variables getting assigned values that cannot be known, the first thing to do is to introduce a Supremica variable to represent the values that the input signal can take. The next step is to make sure that when the variable is read, it does have the possibility to have any of its possible values. This is done via actions on the transitions named $e1$ and $e2$ in Figure 4.2. This is needed since the variable is used to represent an input signal, and this value cannot be calculated from the function itself. The value will just be read when the function is called, and it could in theory have any of its possible values at the time of the function call. Since the transitions have no guards, the model can assign any of the possible values by taking one of the transitions, which makes the model accurate.

This reasoning could be generalized to all types of input data that are unknown for the function. One thing to keep in mind is that the shown way of assigning a value to *In1*, by using transitions, is of limited use if the number of possible input values of *In1* is high, but is useful for Booleans.

4.2 Specifications

When the model has been constructed, the next step is to look for specifications that describe the intended behaviour of the system. These specifications can be made from the requirements of the system, or from someone who has designed the system and know what the intended behaviour is. How the requirement is translated to a specification, which is also a state machine, is described in this section.

4.2.1 Requirements for software

The Lateral State Manager was, at the time of this thesis, implemented as MATLAB-code with a few comments and sparse other documentation of the intended behaviour. Therefore, a lot of effort was put into finding out the intended behaviour of the Lateral State Manager module and the Planner subsystem. This was done by studying unit-tests, running simulations and by talking to the developers.

There are two different kinds of requirements that have been translated into specifications, *Module requirements* and *Implementation correctness*, these will be described here in some detail.

Module requirements

Module requirements are the requirements that would be hard to understand for someone who does not have any insight of the whole system;these try to answer questions like: what outputs are correct given some input? In order to find answers to questions like that, it is important to have a good system understanding, or that someone who has a good understanding has written requirements.

In this thesis, requirements regarding traffic safety, traffic laws and common sense in traffic were looked into, and some examples follow. One example is that the lane change should be done to same lane as the one requested. Another example is that if no lane change is requested, then no lane change should be done. A third example is that if no lane change is requested the turning indicators should be turned off. These kind of intuitive requirements were translated into specifications that connected to the corresponding variables.

Implementation correctness

Another approach is to look at the implementation of code, since there are implementation issues that may result in logical errors. In contrast to the Module Requirements, these kind of issues do not depend on any detailed system understanding. For example, if a timer that keeps track of elapsed time is used, this timer should be incremented exactly once every time the updateState function is executed. This is something that cannot be found by any compiler or static syntactical analyser, since that kind of software tools do not know what the intended behaviour is.

But on the other hand, there is no need of knowing anything about the code outside of the module.

Neither syntactic nor static verification, where the latter in short looks for division by zero, overflows, data conversion, are done in this thesis. This so, since syntactic problems are assumed to be found in the MATLAB compilation, and since Supremica is not made for static verification.

4.2.2 Specification examples

Examples of how specifications are used in Supremica are given in this section.

A first example is to verify that variables that should have the same value after a time step actually have that. A specification for this can be seen in Figure 4.9. If the event label *newIteration* is assumed to show that a function is called repeatedly, as described in Section 4.1.5, the transition between 'S0' and 'S1' can be used as a handle to see that the assignment of values to variables works as intended. If the assignment does not work, and the variables could take a time step without having the same value, this will mean that the specification deadlocks. Thus, if the specification is synchronised with the model and the model allows $Var1 \neq Var2$, the resulting state machine will also deadlock, which the verification tool in Supremica will find. More about this in Section 4.3.

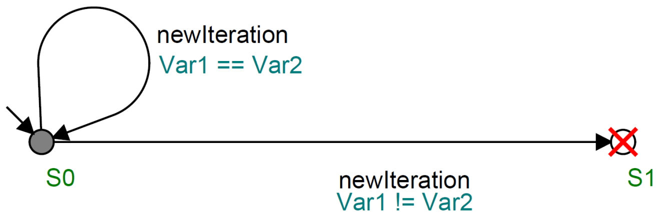


Figure 4.9: An example of a specification that makes sure that two variables always have the same value when a new iteration occurs.

Another example of a specification is to verify that timers are updated once per time step. A specification for this can be seen in Figure 4.10. It is once again assumed that *newIteration* is used to show that a function is repeatedly. If the timer is updated once and a time step is taken via *newIteration*, before next update, everything is fine. But, if two timer updates occur the specification deadlocks, which the verification tool will find.

4.3 Verification

As mentioned before, the purpose of making plant models and specifications is to be able to verify logical correctness. The verification itself is done with Supremica's built in tool for verification. The model and the specification are synchronised together to a new state machine. If the specification is violated anywhere, this will give raise to a deadlock in the synchronised model, these deadlocks will then be found in the verification step.

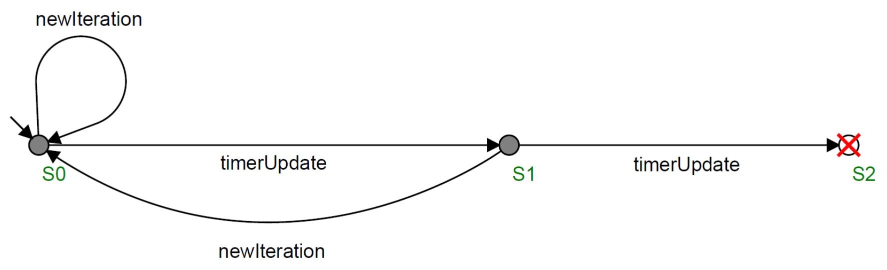


Figure 4.10: An example of a specification that makes sure that a timer update can happen only once each iteration.

One of the remaining obstacles to handle is to identify if an indicated error is present in the real system, or if it is introduced in the modelling. The latter is referred to as a false positive.

4.3.1 Positives and false positives

When using the verification tool *Conflict Check* in Supremica, the result is presented in a message box; either no blocking state was found which means that, as long as the modelling is correct, the system is correct or the system is blocking and a deadlock is found. Then Supremica will give a counterexample where the specification is violated, and give a trace that leads to the blocking state. Only the first found counterexample is given, so if there are multiple places where specifications are violated these will not be shown. This could partly be handled by only synchronising one specification at a time with the plant model. When the system has been verified, and possibly corrected, so that all specifications individually have been proven to be fulfilled, all specifications could be checked simultaneously with the plant model to check that all specifications are fulfilled at the same time.

False Positives

Receiving the message that the system is non-blocking ends the verification of that specification. But, a blocking situation might need a little extra examination to determine where it happens, and if this is an actual behaviour of the system or if it is a *false positive*. As described earlier it might be both more convenient, and of great interest, to make a model of the system which captures a greater domain, A', than the real system, A, does. This means for example that all possible input signals are accepted, i.e. any coupled input is neglected. If this is the case, the verification is even more valuable, because if the absence of logical errors is shown with unexpected input the system is tolerant to faults. The backside of this way of modelling is that if a blocking state is found in A' it cannot be guaranteed that this is in the real system, and if not it would be classified as a false positive. A false positive is illustrated in Figure 4.11 (a). Whether it is a false positive or not has to be determined, and there are strategies for this.

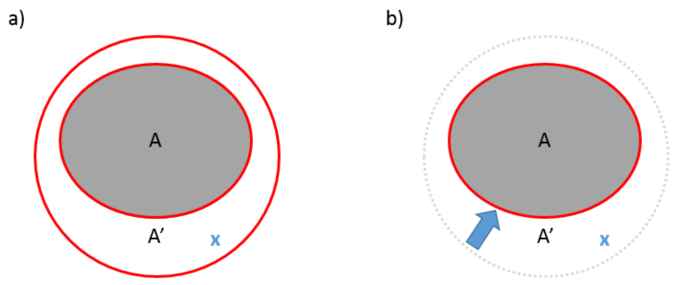


Figure 4.11: If a generalized model A' is used instead of the system model A, this might give raise to false positives. (a) False positives are issues that can not be found in A, but only in A', a false positive is marked as an x. (b) In the case of a positive in verification of A', the model has to be changed to a model closer to A.

What one probably first should try to do is to see if the blocking state is reached where it is obvious that it cannot be reached in the real system, i.e. the system is not robust in this sense but might be working under the correct initialisation. If this can be shown to be the case the model can be updated according to this and verified again. This is illustrated in Figure 4.11 (b).

If it is hard to determine whether the positive can occur, maybe because any possible covariance between variables are unknown, the next alternative is to try to show the behaviour in the real system, because then one has to assume that the positive really is there. If it can not be shown, then one has to check that the model is consistent with the system.

4.4 Summary: Methods

This chapter has described methods for modelling MATLAB-code in Supremica. Many of the ideas are generally useful and could be applied when modelling source code written in other languages than MATLAB.

It is well worth noting that even if the steps of modelling, making the specifications and verifying the system are three different tasks that depend on each other sequentially, it is not necessarily that the work is done in such sequence. It could be that the all three parts are developed continuously in the verification process.

5
Results

When the Lateral State Manager's 223 lines of MATLAB-code are modelled in Supremica as an EFSM, as described in Chapter 4, it consists of 75 locations. If this would have been implemented as an FSM, it would have 601168 states. The difference comes from the fact that each combination of seventeen variables has to be represented as a state in the FSM.

However, the main subject of this chapter is to present the specifications that was used to verify the Lateral State Manager together with the results from the verification. In total, seven specifications are implemented, and four out of these turned out to be violated by the model. Three of the violated specifications was further analysed in Supremica.

5.1 Timers

In the Lateral State Manager there are three Boolean variables, lets call them 'timer1', 'timer2' and 'timer3'. Each of these variables are holding timers (the variables are not of simple type, but instances of a class) that are incremented each time the Boolean value is updated with the same value that is already stored, i.e. a time step has been taken and the value is intact. The timers are reset when the value is changed.

To check that only one incrementation per time step is performed specifications for this was produced for each of the variables. The verification showed that two of the timers, timer1 and timer2, where updated exactly once each time step, but that timer3 could in certain situations be updated twice.

Since the violated specification is a timer, which it could be reasonable to assume should hold the correct value of the elapsed time, another specification was produced to check whether the double update of timer3 could happen multiple times. This verification showed that this was possible as well. This means that the most extreme case, that the timer is having a value that is double the elapsed time, is possible. What the consequence would be for the lane change algorithm is not further investigated.

With help of the counterexample in Supremica it could be shown that the root of the problem was that the variable was updated not only in its during method, but also in its enter method. Since the model was built to follow the source code as closely as possible it was a simple task to draw the conclusion that it would be sufficient to take away one line of code in the source code to solve the problem. This line was updating timer3 in the enter method, which turned out to be faulty in certain conditions. This was then corrected in the Supremica model, and the verification was performed once more. This showed that the problem was solved and the code can easily be corrected in the same way.

5.2 Lane Change Request

In MATLAB the Lateral State Manager receives a request from other parts of the program if the vehicle should stay in the current lane, or change to either left or right. From a system requirements point of view it does make sense to make sure that when a lane change is performed, it actually follows the current request. Therefore a specification to check this property was constructed.

In practice this means to check if the value of an internal variable, lets call it 'direction', and the incoming request could be different from each other for two time steps. Difference for one time step would probably be sufficient to check for this, but the specification was made like this to make sure that, if it was violated, it was not a false positive somehow originating from model abstractions. This extra margin originated from that a positive was received that had its root in that the initial values of the Supremica variables was more tolerant than the original system. Therefore the model was adjusted to come closer to the real system, but the specification was kept like this for additional reliability. The result of the verification showed that these values could be different for at least two consecutive time steps.

Since the Lateral State Manager is only a part of the lane change algorithm it was not obvious what this discrepancy between the variable values could possibly lead to. Therefore this case was further examined with the use of simulations, in an environment used by the developers to run test scenarios. The result from the simulations showed that when a request was changed, some parts of the program still followed the first request and other parts followed the current request. This lead to that the vehicle changed lane according to its first request, but was checking for traffic in the opposite lane from the one it was changing to.

Since this was a quite severe issue some further investigations of possible solutions were done. With guidance of the counterexample trace from Supremica it could be concluded that the problem could possibly be solved by adding an extra transition from state 'IntermediateState2' to state 'NoRequest'. This solution was implemented in the model and verified once again. The result of the verification showed that the problem had been moved to another 'IntermediateState'. Since that indicated that this problem was not just a slip in one part, but possibly a situation that was overlooked, no further investigations were made about finding solutions.

5.3 Turning indicators

From a system perspective it is reasonable to make sure that when no lane change is requested, the turning indicators should be turned off. A specification to check that this was fulfilled was produced, and the result of the verification showed that the turning indicators could be on even if there were no request to change lane. The counter example showed that this situation could appear if the state changes from state ’IntermediateState3’ to state ’NoRequest’. However, the turning indicators are turned off in the state ’NoRequest’ so the indicators are only turned on one time step too many, and this is an example of something that testing would probably not find. In this case it might be harmless, but in other situations this might not be the case.

An interesting note about this issue was that, once it was confirmed, solutions to solve the problem were tried out. It turned out that the most obvious solution did work for the turning indicators, but that new issues about the timer updates where then introduced. Further search for solutions was then abandoned.

5.4 No request

From a system perspective it is reasonable to make sure that if there is no lane change request, the Lateral State Manager should be in the ’NoRequest’ state. A specification to check this was therefore constructed. The result showed that the Lateral State Manager could be in ’IntermediateState2’ when the current request was ’None’. Since Supremica only gives one counterexample, and no solution to this problem was introduced, no other states were shown to have the same possible behaviour. However, examination of the MATLAB-code led to the conclusion that this would probably also be possible in the ’IntermediateState4’, ’IntermediateS-tate5’ and Finished states.

This was tested in simulations but was not able to be shown. The reason for this might be that there is some un-modelled covariance between variables used in the decision making.

5.5 Summary: Results

The resulting model, constructed based on what is presented in the methods chapter, has in this chapter shown to be able to not only find issues in the model. All except one of these have been found existing in the real system as well. One solution was implemented in the model and found working, two other attempts were done, but showed that issues may be moved, or lead to other issues when adding fixes. This shows the benefit of using formal verification, since this could possibly have been missed if a fix was implemented directly in the code.

6
Conclusions and future work

This thesis is a proof of concept that shows that formal verification can indeed contribute in the verification of software in real development projects, since the modelled state machine was proved to have some issues. This chapter discusses what this could lead to next.

6.1 Conclusion

Modelling the EFSM in Supremica was a relatively small task compared to finding and making specifications to use in the verification. One of the reasons for this was that there was not much documentation about what the Lateral State Manager was supposed to handle, so the main source for finding specifications was studying the MATLAB-code and thereby trying to find the intended behaviour. If more well defined specifications had been available, that would probably have shortened the phase of defining what behaviours to look for. Still, it can be considered as promising that the software, even without given specifications, could be verified with the result of finding issues that had not been found in the development process. Since the modelling could be assumed not to be a very tedious task it could therefore be motivated to model safety critical parts of software, with well defined desired behaviour, manually.

Another thing to take home from this project is that the verified model was relaxed in some of the initialisation, and that the verification is therefore checking the robustness in the system. Only in one of the specifications were problems regarding this more tolerant initialisation observed and the model altered accordingly. That this was a viable route shows that robustness is indeed reasonable to include in the verifications process.

6.1.1 The nature of the specifications

It is well worth noting that the implemented specifications show that there are different types of behaviour to look for. The update of the timers is a real implementation check, and the other specifications check for module level correctness. The way of finding the issues was also a bit different. The timer variables where checked like a routine check, if timers are implemented one can assume that they should be showing the elapsed time.

The other specifications where checking for module level requirements, and the most obvious example of this is the issue about the turning indicators. It seemed to be reasonable to make sure that the vehicle was not using turning indicators if there was no lane change request. This issue would probably had been hard to find with ordinary testing, and one can argue that it would not affect very much. But that is a somewhat improper discussion in this case, the thesis is a proof of concept, and this kind of issues could have been found where it could possibly be more problematic.

6.1.2 Additional remarks about modelling

This thesis has focused on a small part of the lane change algorithm, which has paid off reasonably well. But it should be added that it is sometimes not sufficient to look for that each module is doing what it is supposed to do, but also what can happen if multiple modules are connected and the output from one effects what happens downstream, as was shown in the Darpa project described in Section 1.1.1. This example shows that even if the intention was good with implementing different modules for path-planning and surveying that the path planner worked as desired, problems may pop up when the whole system is connected together. Since formal verification allows multiple modules to be synchronised together, at least in Supremica, this type of problems could possibly have been found, but that needs a broader approach then what is shown in this thesis project.

6.2 Future work

So, formal verification has shown to be possible to use to verify correctness of software that is not prepared specially for it. This section now discusses how to continue the work of implementing the usage of formal verification in the development process.

6.2.1 How to work efficiently with formal verification

In this thesis, software is verified from already existing code. Another solution would be to work the other way around; start with modelling and verifying the correctness, and when this is done start the actual implementation. Already the top level requirements for a system could possibly be represented with flow-charts or state machines, and therefore be verified to be logically correct. The benefit of focusing on the logical behaviours from the start is that these anyway have to be sorted out to get a well working system, and that when everything except the logic is taken away there are less possibilities for flawed ideas to be disguised under implementation details.

One can argue that it is hard to fully know from the start what functionality a system needs to serve its main purpose, and that it is easier to start directly with implementation and see where obstacles occur. But this doesn't necessarily stand in contrast to using formal verification. In fact, one could argue that no meaningful implementation can be done without some idea about what functionality it is intended to add. In the same manner, code implementation does not necessarily have to solve everything from the start, the model could be continuously updated as the system evolves.

Furthermore, with a model of the system available it is possible to check that if something is to be altered or added, this will actually add the intended functionality without ruining something else. The price paid is of course that some extra time have to be spent on the modelling, but it will probably lessen the burden for debugging since the logic is already verified.

It would therefore be of great interest to make further studies of how to implement formal verification as a tool that not only helps in the development process, but could also be used to verify that the finalized system is correct.

6.2.2 Automating the process

The need of a model to be able to use formal verification is one of the big disadvantages of the concept, and it is therefore tempting to look for ways to automate some of the work. It is beyond the scope of this project to discuss this in detail, but some remarks can be made.

In research there are ongoing projects which aim to verify, and even update code to make it work correctly, as the the Gadara project described in Section 1.1.1. This is of course one way of automating the procedure. This could be very suitable to look for general issues in multithreaded programs, such as the handling of mutual exclusive execution to make sure that there is no risk of deadlock. However, for other types of issues it will still be up to the user to define specific behaviours to verify.

Another solution would be to work the other way around; start with modelling and verifying the correctness, and when this is done adequately, auto generate code from the model. Specifications would still be needed to check the system, but the process of defining what behaviour one desires have to be done anyway before the implementation work can start.

What alternative that is the best is hard to say with only the results from this project, but further examination of both alternatives could be interesting topics to look into.

Bibliography

[1] E. Seligman, T. Schubert, and M. V. A. K. Kumar, *Formal Verification.* Elsevier Science, 1 ed., 2015.

[2] Y. Wang, H. Liao, S. Reveliotis, T. Kelly, S. Mahlke, and S. Lafortune, "Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, pp. 4971–4976, Dec 2009.

[3] H. Liao, Y. Wang, H. K. Cho, J. Stanley, T. Kelly, S. Lafortune, S. Mahlke, and S. Reveliotis, "Concurrency bugs in multithreaded software: modeling and analysis using petri nets," *Discrete Event Dynamic Systems*, vol. 23, no. 2, pp. 157–195, 2013.

[4] DARPA, "Darpa urban challange." http://archive.darpa.mil/grandchallenge/. Online, accessed 2016-06-03.

[5] T. Wongpiromsarn, *Formal Methods for Design and Verification of Embedded Control Systems: Application to an Autonomous Vehicle.* PhD thesis, California Institute of Technology, 2010.

[6] 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*, pp. 384–385, 2006.

[7] 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*, pp. 7000–7005, 2011.

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

[9] C. G. Cassandras, S. Lafortune, S. O. service), and S. (e-book collection), *Introduction to Discrete Event Systems.* Springer US, 2008;2007;.

[10] 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.

[11] A. Dasso, A. Funes, B. (e-book collection), K. (e-book collection), and I. Books24x7, *Verification, validation and testing in software engineering.* Hershey, PA: Idea Group Pub, 2007;2006;2013;.

[12] G. J. Holzmann, "Software model checking, lecture notes, nato summer school," 2000.

[13] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie, *Systems and Software Verification.* Springer Berlin Heidelberg, 2001.

[14] R. Alur, *Principles of Cyber-Physical Systems.* The MIT Press, 2015.

[15] M. Byrod, B. Lennartson, A. Vahidi, and K. Akesson, "Efficient reachability analysis on modular discrete-event systems using binary decision diagrams," in *Discrete Event Systems, 2006 8th International Workshop on*, pp. 288–293, July 2006.

[16] S. Mohajerani, *On compositional approaches for discrete event systems verification and synthesis.* PhD thesis, Chalmers University of Technology, 2015.

[17] Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516, 1978.

[18] R. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.