A Preliminary Report on Tool Support and Methodology for Feature Interaction Detection

Alma L. Juarez-Dominguez, Nancy A. Day

School of Computer Science
University of Waterloo, Canada
{aljuarez,nday}@cs.uwaterloo.ca

Richard T. Fanson

Critical Systems Labs Inc.
Vancouver, Canada
richard.fanson@cslabs.com

Technical Report CS-2007-44

December 18, 2007

Abstract

In this report, we describe our effort to create in Matlab's Stateflow a set of non-proprietary advanced automotive feature design models, and to translate these design models into models that can be input to the model checker SMV. We are interested in verifying the absence of feature interactions in the integration of the automotive features, as well as the lack of errors in the design. In the automotive domain, a feature is a bundle of system functionality recognized by the driver and providing advanced functionality to the vehicle, for instance, Cruise Control (CC). Each feature is normally implemented in software and has a degree of control over the mechanical components that operate the dynamics of the vehicle. Examples of these mechanical components are brakes, throttle and steering. We have created our set of non-proprietary feature design models to assess different techniques and tools that analyze the integration of the features and their correctness. The translated models will allow us to verify properties of the automotive features at the same level of description as the design, so we can ensure that the findings of our analysis are applicable to the design of the automotive components.

1 Introduction

Automotive embedded systems consist of software, as well as electrical and mechanical processes monitored by the software. In the automotive domain, a feature is a bundle of system functionality recognized by the driver; it is normally implemented in software, and provides advanced functionality to the vehicle. For instance, an automotive feature is Cruise Control (CC). These features have a degree of control over the mechanical components that operate the dynamics of the vehicle. Examples of these mechanical components are brakes, throttle and steering.

Ideally, features should be able to be integrated into or removed from an existing embedded system without problems. However, when features that have been developed and tested in isolation are integrated into a system, the features' combination can cause undesired or unexpected system behaviour. These unexpected and undesired interactions form the essence of what is called the feature interaction problem. Feature interactions in automotive systems arise from the activation of two or more features sending requests to the mechanical processes that cause contradictory physical forces, possibly at distinct times. An example of a feature interaction in an automotive embedded system is having simultaneous requests to apply the brakes and the throttle. The feature interaction problem becomes more prevalent with the increased complexity in systems. Given that features often evolve as a product line, feature interaction is an ongoing problem.

Our goals in this work are to create a set of non-proprietary feature design models in Matlab's Stateflow [1], and to translate the design models of automotive features in Stateflow into an input representation to be used in the future for feature interaction analysis in the model checker SMV [13]. We call our non-proprietary feature model set of automotive features the "University of Waterloo Feature Model Set" (UWFMS). The UWFMS is novel in the sense that there is not a publicly available set of models that we could have used. Some of the non-proprietary feature design models are derived from the publicly available textual descriptions provided in the TRW website [2]. Another feature was devised by us, so that we would have a larger number of feature design models to work with. We chose to model our UWFMS in Stateflow/Simulink because these tools are extensively used for designing embedded components in the automotive and avionics industry. After studying the semantics of Stateflow, we decided to translate the design models into models that can be used to perform model checking in SMV since this tool is suitable for describing the semantics of Stateflow and of the composition of features. The translated SMV models will allow us to verify properties of the automotive features at the same level of description as the design, so that the findings of our analysis can be applied to the design of the final automotive components. Some of the properties that we are interested in verifying are the absence of feature interactions in the integration of the automotive features and the lack of errors in their design. Most of the previous work on the formal analysis of Stateflow models assumed that Stateflow's semantics and that of Statecharts were the same, but this is not the case, so our work follows closely the semantics of Stateflow. To study feature interaction, we must create an integrated model of multiple features. This requirement makes our work different from previous work on translation and analysis of Stateflow models since in those efforts the purpose of property verification only considered one model at a time.

In Section 2, we provide a brief description of the Stateflow notation that we used when modelling automotive features. Section 3 explains how the features' functionality is modelled in Stateflow to create our non-proprietary feature design models. In Section 4, we give a brief description of the SMV notation that our translated models use. We explain the translation process from Stateflow to SMV models in Section 5, along with issues regarding the use of Stateflow, and the challenges we overcame during the translation process. Finally, we briefly overview related work in Section 6 and conclude in Section 7.

2 A Brief Description of Stateflow

In this section, we describe the subset of the Matlab's Stateflow notation that we used when modelling automotive features. Matlab is the foundation for Simulink and all other MathWorks products. Simulink is software for modelling, simulating, and analyzing dynamic systems, and Stateflow is an interactive graphical design tool that works with Simulink to model and simulate event-driven systems. Our description is based on the Stateflow documentation [1], and the Dabney and Harman book [8].

The syntax of Stateflow is similar to that of Statecharts [11]. For the purposes of this brief overview, we assume the reader is familiar with the Statecharts family of notations. Some of the differences from Statecharts are that the Stateflow action language has been extended to reference Matlab functions, and that Stateflow does not perform true concurrency for AND-states as Statecharts does. In Stateflow, AND-states execution is sequential: each AND-state reacts to the same input, but only one AND-state executes at a time.

A Stateflow design model consists of a set of *states* connected by arcs called *transitions*. Each state has a name and can be decomposed, creating a hierarchical state machine. The Stateflow documentation defines two kinds of decomposition for a state, which are: (1) 'exclusive' or 'OR-states' (indicated by solid borders) and (2) 'parallel' or AND-states' (indicated by dashed borders). In Stateflow, at each level of the hierarchy, only one kind of decomposition can be chosen, *i.e.,* all states at the same hierarchy level must be either OR-states or AND-states. Unlike Statecharts, Stateflow AND-states are not truly concurrent since Stateflow actually runs in a single thread during simulation. Each AND-state is executed sequentially following its respective execution order. The execution order is based on the geometric position of the AND-states, where priority is assigned from top to bottom and then from left to right, according to the rules:

- The higher an AND-state's vertical position in the diagram, the higher its priority for execution.
- Among AND-states with the same vertical position, the left-most state receives highest priority.

The lower the number, the higher the priority. This order determines when each AND-state performs the actions it executes, only one state at a time for a given set of inputs, but the same set of inputs is used for all AND-states in the same level of the hierarchy.

A STATEFLOW model can have data input/output ports and event input/output ports. Both data and events can be defined as local to the STATEFLOW model or external, *i.e.,* communicated from the SIMULINK parent model through ports. Each transition's label follows the syntax:

event[condition]/transition_action

Each part of the label is optional. The event specifies an event that causes the transition to be taken, provided the condition, if included, is true; the condition is a boolean expression that, when true, allows a transition to be taken; the transition_action is executed after the condition, if included, is true. Each transition has also a priority of execution, determined by the hierarchy level of the transition's destination state, the type of information in its label (*e.g.,* events have priority over conditions) and the geometric position of the transition source. The lower the number, the higher the priority. If a state has no output transitions, the model deadlocks. A history junction represents historical decision points in the STATEFLOW diagram, indicating that historical state activity information is used to determine the next state to become active.

When modelling automotive features, we do not use the following STATEFLOW syntax: condition_action's in transitions, actions within states, connective junctions, graphical functions, MATLAB functions, the In(state_name) condition function (which is evaluated as true when the state specified as the argument is active), temporal conditions (use of temporal logic within STATEFLOW), and any notation that could allow event broadcasting. These syntactic elements were not needed when modelling any of our automotive features, and we decided not to use them. If they are desired while designing a feature model, in most cases, an equivalent design can be created without these syntactic elements.

3 University of Waterloo Feature Model Set (UWFMS)

In this section, we describe the functionality of each of the automotive feature design models in our non-proprietary set and show how it is modelled in STATEFLOW. Some of these feature design models are based on TRW Automotive features' textual descriptions as provided at their website [2]. These features are regarded by TRW as 'Active Safety Systems', under the heading of 'Driver Assist Systems'. The quotes and figures in this section are taken directly from the TRW website. Another feature was devised by us to have a larger set of feature design models to work with. We call the whole set of non-proprietary feature design models the "University of Waterloo Feature Model Set" (UWFMS).

We provide the final STATEFLOW design model per feature, as well as a brief description of the design decisions that we made when modelling our UWFMS in STATEFLOW. The inputs, outputs and local variables used in each design model are also explained in a table for each type of variables. These tables are placed close to the design model to help the reader to understand the figures.

3.1 Adaptive Cruise Control (ACC)

"TRW's Adaptive Cruise Control (ACC) technology improves upon standard cruise control by automatically adjusting the vehicle speed and distance to that of a target vehicle. ACC uses a long range radar sensor to detect a target vehicle up to 200 meters in front and automatically adjusts the ACC vehicle speed and gap accordingly."

Figure 1 shows an example of the feature's execution from the TRW website.

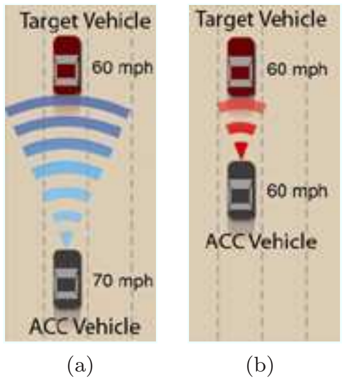


Figure 1: Adaptive Cruise Control Functionality: (a) The ACC vehicle approaches the Target vehicle at 70 mph (ACC's vehicle set speed); (b) Due to the proximity, ACC starts coasting by adjusting the ACC vehicle's speed to 60 mph, matching the Target vehicle's speed.

There are two kinds of buttons that can be used when modelling features:

- A *Push button*, when clicked, causes an action. Given that the triggering of an action is instantaneous, push buttons are normally modelled by events (*e.g.,* Cancel). However, some push buttons can be held. These are modelled as two events, one to capture when it is initially pressed (*i.e.,* SetAccelIn), and another to model the instant at which it is no longer pressed (*i.e.,* SetAccelOut).
- A *Toggle button*, when clicked, alternates between two states, which are set and unset. Given that the state remains, toggle buttons are normally modelled by data such as a Boolean (*e.g.,* CC_Enabled).

Figure 2 presents ACC's functionality modelled in Stateflow, which we will explain in the following paragraphs. Table 1 shows the input variables, Table 2 the output variables and Table 3 the local variables used in the ACC's Stateflow design model.

The details of our design in Stateflow are as follows: ACC should automatically accelerate and decelerate the ACC vehicle based on a target speed specified by the driver and a constant distance separation with a Target vehicle of 50 meters. ACC consists of two AND-states: the LOGIC_CONTROL state, which controls the vehicle logic for the throttle, and the SPEED_SETTING state, which keeps track of the target speed. The default of the LOGIC_CONTROL AND-state is the DISABLED state, and ACC should start in the DISABLED state when the car is turned on. When the driver turns ACC on by pressing the CC_Enabled button, ACC enters the ENABLED state, and sets the target speed to 0 (which remains 0 until the feature is engaged). The default of the ENABLED superstate is the DISENGAGED state. Pressing the Set/Accel button when the speed is greater than 40 KPH makes ACC to enter the ENGAGED state. The default of the ENGAGED superstate is the COASTING state. ACC must be in the COASTING state when either the current speed exceeds the target speed, or if the vehicle ahead is less than 50 meters away. Coast means that the ACC vehicle decreases its speed by requesting no throttle, but not by braking. ACC moves to the ACCELERATING state when the current speed is less then the target speed and the Target vehicle is farther than 50 meters from the ACC vehicle. The throttle output is proportional to the difference between the target speed and the current speed. Depressing the brake or pressing the Cancel button shall cause ACC to enter the OVERRIDE state. This state remains active until either the Set/Accel or Resume/Coast buttons is pressed, which makes ACC to enter the ENGAGED state. An Error event at any time when ACC is on will cause a transition to the FAIL state, and it will not be able to recover from this condition until the ACC vehicle is restarted.

The default of the SPEED_SETTING AND-state is the HOLD_SPEED state. SPEED_SETTING only acts when ACC is engaged. Pressing the Set/Accel button makes ACC to enter the INC_SPEED state and locks in the current speed value as the target speed. If the Set/Accel button is held while the target speed is less than 200 KPH, the target speed is incremented for every clock signal that occurs. Release of the Set/Accel button sends ACC back to the HOLD_SPEED state. Pressing the Resume/Coast button makes ACC to enter the DEC_SPEED state and locks in the current speed value as the target speed. If the Resume/Coast button is held while the target speed is greater than 0 KPH, the target speed is decremented for every clock signal that occurs. Release of the Resume/Coast button sends ACC back to the HOLD_SPEED state.

Type	**Name**	**Meaning**
event	Clock	A clock signal for the STATEFLOW model
event	SetAccelIn	Signal indicating depression of Set/Accel button
event	SetAccelOut	Signal indicating release of Set/Accel button
event	ResumeCoastIn	Signal indicating depression of Resume/Coast button
event	ResumeCoastOut	Signal indicating release of Resume/Coast button
event	Cancel	Signal indicating depression of Cancel button
event	Error	A signal indicating when an error has occurred
data	CC_Enabled	Driver controlled main power to enable/disable the feature [Boolean]
data	FollowDist	Number to describe distance to the vehicle ahead (Value between 0 and infinity, from a processing sensor unit) [Meters in integers]
data	BrakePedal	Physical brake pedal input (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 200) [KPH in integers]

Table 1: Input Variables Used in Adaptive Cruise Control (ACC)

Type	**Name**	**Meaning**
data	Throttle	Value proportional to difference between the target speed and current speed [Percentage in integers]
data	TargetSpeed	Target cruising speed of the feature (note this could be a local variable and is only an output so we can see its value in the display) [KPH in integers]

Table 2: Output Variables Used in Adaptive Cruise Control (ACC)

Type	**Name**	**Meaning**
data	CC_Engaged	Value that indicates when ACC is engaged [Boolean]

Table 3: Local Variables Used in Adaptive Cruise Control (ACC)

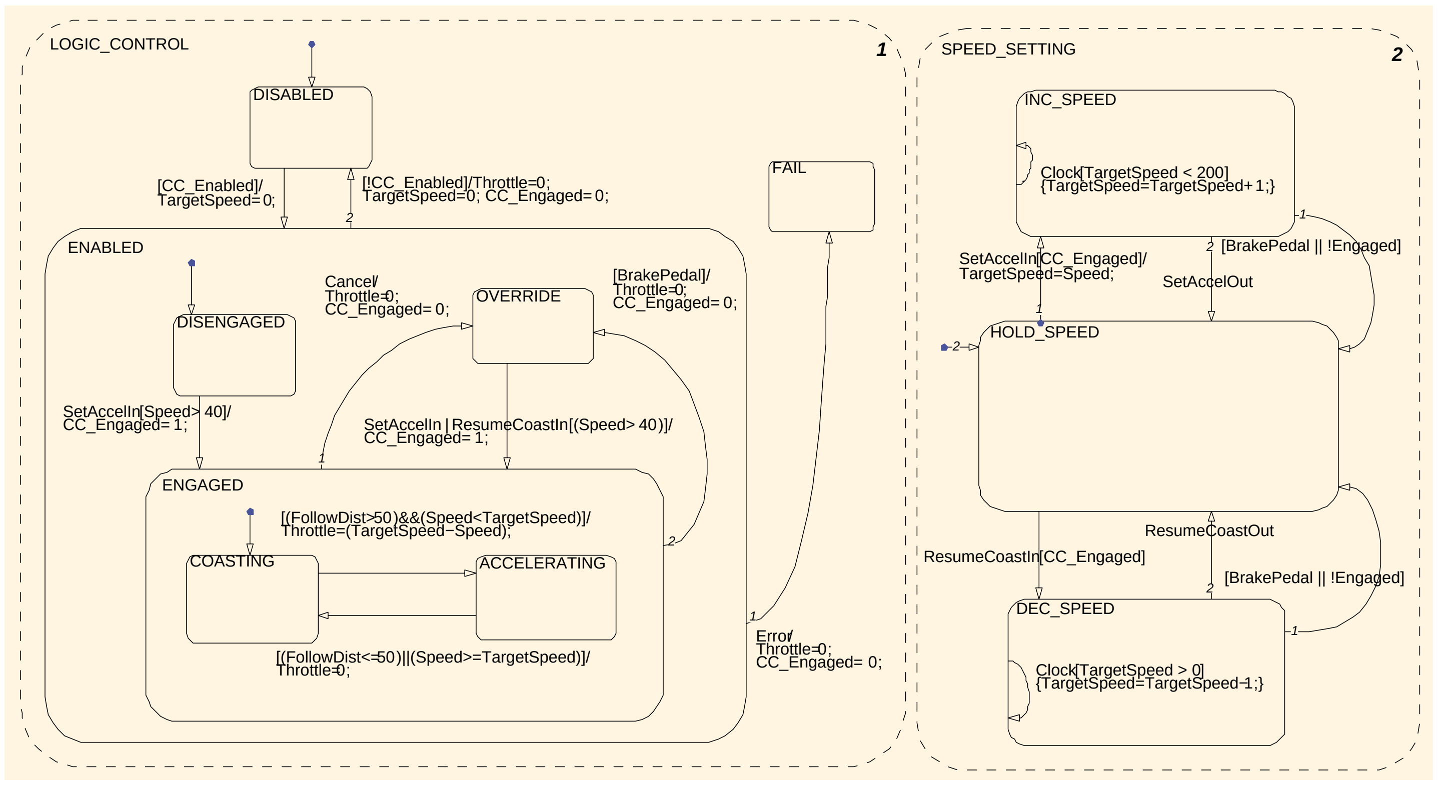
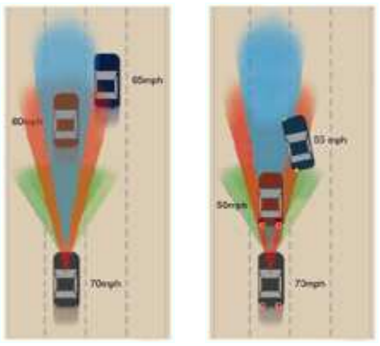


Figure 2: Adaptive Cruise Control STATEFLOW design model

3.2 Collision Warning (CW)

> "TRW's Collision Warning (CW) System can assist drivers by helping to prevent or mitigate accidents. Combining long and short range radars with a video camera, TRW's Collision Warning monitors the road ahead (including part of the side-fronts)."

Figure 3 shows an example of the feature's execution from the TRW website.



(a) (b)

Figure 3: Collision Warning Functionality: (a) CW uses radars and sensors to monitor the presence and distance of vehicles ahead of the CW vehicle; (b) When CW senses a collision threat with a vehicle in front, it alerts the driver (if a mild threat) and applies brakes (if an imminent threat).

Figure 4 presents CW's functionality modelled in Stateflow, which we will explain in the following paragraphs. Table 4 describes the input variables and Table 5 describes the output variables used in the CW's Stateflow design model.

The details of our design in Stateflow are as follows: The CW feature will begin in the DISABLED state. When the driver turns CW on by pressing the CW_Enabled button, CW enters the ENABLED state. The default of the ENABLED superstate is the DISENGAGED state. Once the CW vehicle's speed exceeds 25 KPH, the feature will enter the ENGAGED state. When engaged, CW will be able to warn the driver and/or take action if a possible threat is encountered. It is assumed that some pre-processing of sensor inputs will compute the sensor data and convert it into a threat input for CW. The threat input data could be either: None=0, Mild=1, Potential=2, Imminent=3. If a threat input is received while in either the ENGAGED, WARN, AVOID, or MITIGATE states, one of the following transitions will occur depending on the threat:

- In the case of a mild threat, a mild threat warning will be output (*i.e.,* Warning = 1) with no braking intervention. CW enters the WARN state.
- In the case of a potential threat, a potential threat warning will be output (*i.e.,* Warning = 2) and soft braking will occur to slow the vehicle and avoid the potential collision (*i.e.,* Brake = 1). CW enters the AVOID state.
- In the case of an imminent threat, an imminent threat warning will be output (*i.e.,* Warning = 3) and full force braking will occur to mitigate the collision as much as possible (*i.e.,* Brake = 2). CW enters the MITIGATE state.

Output for Braking is simply a 0 (indicating no output braking), a 1 (indicating a soft-brake) or a 2 (indicating a hard-brake). If the vehicle is brought to a halt when in either the AVOID or MITIGATE states, CW will go to the HALT state and the vehicle will be held at halt until there is no threat and the driver presses on the brake pedal beyond a threshold (for our model it is set to 10 percent of depression) to resume control of the vehicle. Then, CW will enter the ENABLED but DISENGAGED state. We assume that the pre-processing threat assessment will account for vehicle speeds so that what once was a threat will not remain a threat at speeds lower than 25 KPH or at a stop. Otherwise, the feature could be stuck at halt or other undesirable outcomes. A driver acceleration pedal input exceeding 75 percent of depression will cause CW to enter the OVERRIDE state from any state in the ENABLED superstate. An Error event at any time when CW is on will cause a transition to the FAIL state, and it will not be able to recover from this condition until the vehicle is restarted.

Type	**Name**	**Meaning**
event	Clock	A clock signal for the STATEFLOW model
event	Error	A signal indicating when an error has occurred
data	CW_Enabled	Driver controlled main power to enable/disable the feature [Boolean]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 200) [KPH in integers]

data	Threat	Input from a pre-processing threat assessment block that converts sensor inputs into a threat:
		Threat = 0: No Threat
		Threat = 1: Mild Threat
		Threat = 2: Potential Collision
		Threat = 3: Imminent Collision
		[Threat value in integers]

Table 4: Input Variables Used in Collision Warning (CW)

Type	Name	Meaning

data	CW_HVI	Value to indicate the message being displayed to the driver through a human-vehicle-interface (HVI):
		CW_HVI = 0: CW Disabled
		CW_HVI = 1: CW Enabled
		CW_HVI = 2: CW Engaged
		CW_HVI = 3: CW Error
		CW_HVI = 4: CW Override
		[Display value in integers]

data	CW_Warning	Audible and/or visual warning to indicate the presence of threats, their severity and the intervention of CW:
		Warning = 0: No Threat
		Warning = 1: Mild Threat
		Warning = 2: Potential Collision
		Warning = 3: Imminent Collision
		Warning = 4: Vehicle Held Stopped
		[Warning value in integers]

data	Brake	Output indicating if the feature is physically intervening by applying brake force:
		Brake = 0: No Braking
		Brake = 1: Soft-Braking
		Brake = 2: Hard-Braking
		[Brake degree in integers]

Table 5: Output Variables Used in Collision Warning (CW)

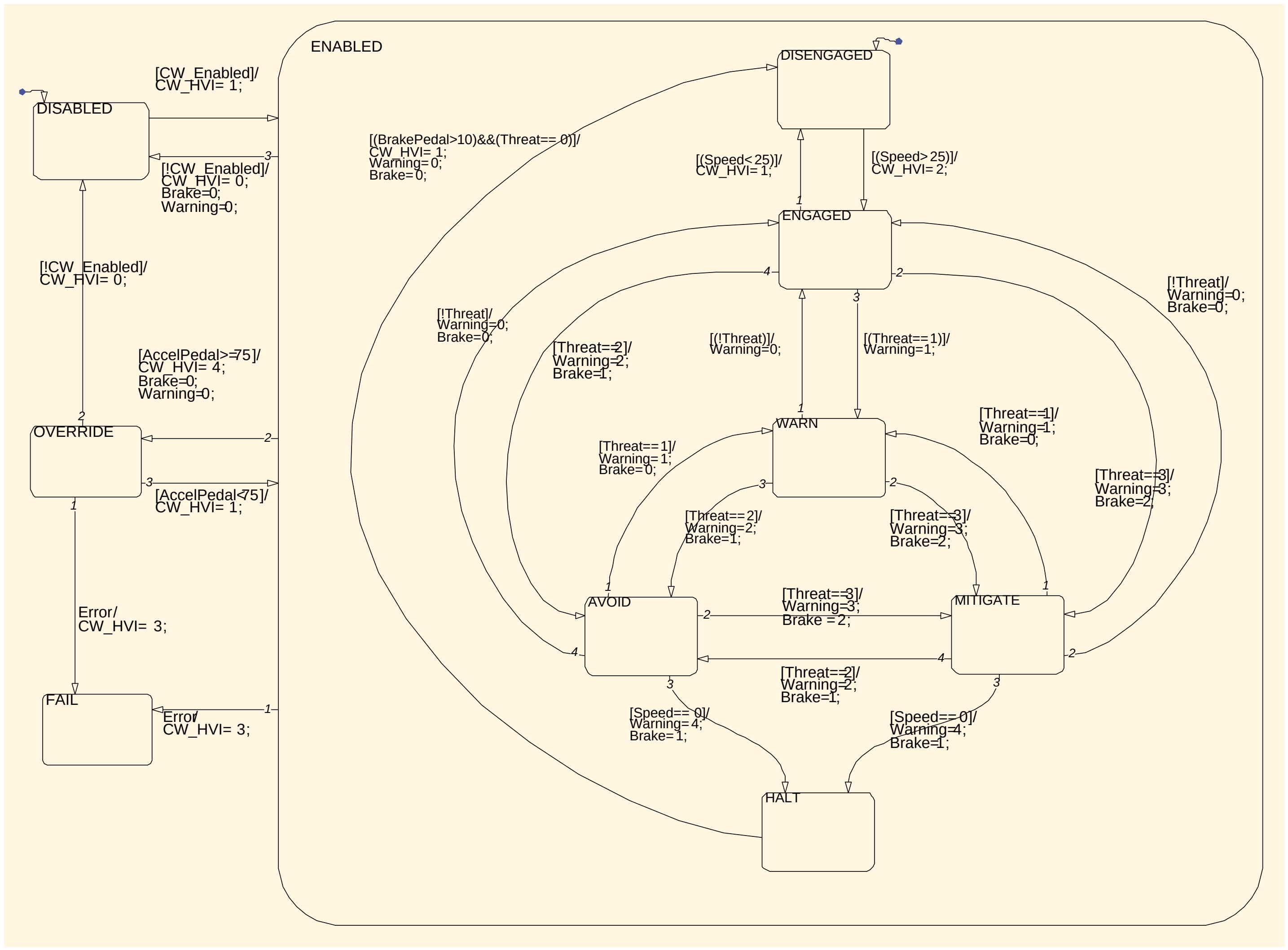


Figure 4: Collision Warning STATEFLOW design model

3.3 Park Assist (PA)

"The TRW Park Assist (PA) system combines electrically powered steering with environmental sensing to aid drivers during parallel parking maneuvers. The system uses short range radar sensors to evaluate the length of the parking slot."

Figure 5 shows an example of the feature's execution from the TRW website.

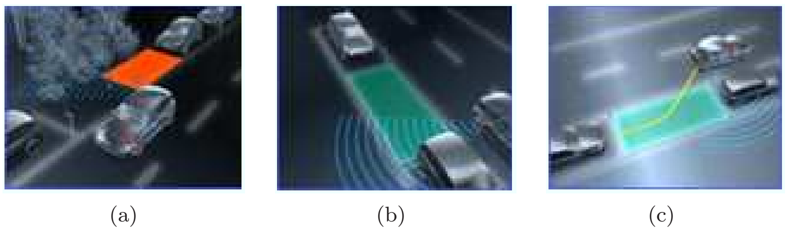


Figure 5: Park Assist Functionality: (a) PA monitors for an empty space where the PA vehicle can fit; (b) If a large enough space is found and the driver accepts the space, PA starts the parking maneuver; (c) PA automatically adjusts the steering, throttle and braking during the parking maneuver.

Figure 6 presents PA's functionality modelled in Stateflow, which we will explain in the following paragraphs. Table 6 describes the input variables and Table 7 describes the output variables used in the PA's Stateflow design model.

The details of our design in Stateflow are as follows: Unlike TRW's PA, in our design, we included the ability for PA to operate the gear, throttle and braking system while the parking maneuver is completed. PA starts in the DISABLED state when the car is turned on. When the driver turns PA on by pressing the PA_Enabled button, PA enters the ENABLED state. The default of the ENABLED superstate is the IDLE state. The enabled PA will wait until the speed of the vehicle is less than 10 KPH to move to the SEARCHING state and start monitoring the sizes of the adjacent parking spaces. We assume that the sensor data is pre-processed by an external module and that PA simply obtains a SpaceFound boolean input indicating when a large enough space has been found. When PA receives the SpaceFound boolean with value true, it will enter the PROMPTING state to prompt the driver (through a human-vehicle interface – HVI) to stop the car and accept or decline the space. If the space is declined or if the driver fails to stop, the external module will make the SpaceFound boolean false, and PA will return to the SEARCHING state to monitor for another parking space. If the driver stopped the car and the space is accepted, PA enters the ENGAGED superstate, where PA will take over and begin the parallel park maneuver. The default of the ENGAGED superstate is the SWIVEL_OUT state. We assume that the progress of the maneuver is monitored by another external component that signals PA with a Next event through a HVI when to transition to the next step of the parking maneuver. The maneuver is broken up into 5 steps, each of which correspond to a state: (1) At state SWIVEL_OUT, when receiving a Next event, PA will first reverse and turn into the parking space. PA will enter the SWIVEL_IN state. (2) At state SWIVEL_IN, when receiving a Next event, PA will continue to reverse but turn the wheels the other way to swivel the front end into the parking space. PA will move to the STOP1 state. (3) At state STOP1, when receiving a Next event, PA will stop and straighten the wheels. PA will go to the CENTER state. (4) At state CENTER, when receiving a Next event, PA will pull forward into the middle of the parking space. PA will enter the STOP2 state. (5) At state STOP2, when speed is 0, PA will finally stop since the maneuver is complete. PA will move to the DISABLED state. Braking by the driver or the detection of a threat during the maneuver will send PA into the OVERRIDE state, which can be resumed from when the threat or braking ceases. Some pre-processing of sensor inputs will compute the sensor data and convert it into a threat input for PA. Steering or acceleration by the driver during the maneuver will send PA to the ABORT state. We designed PA so the feature cannot resume from ABORT since the vehicle is likely off its trajectory path and cannot complete the parking maneuver. An Error event at any time when PA is on will cause a transition to the FAIL mode that cannot be left until the vehicle is shut off and restarted.

Type	**Name**	**Meaning**
event	Next	An signal from a diagnostic external component to indicate when to transition between the different phases of the parking maneuver
event	Clock	A clock signal for the STATEFLOW model
event	Error	A signal indicating when an error has occurred
data	PA_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	SpaceFound	Value to indicate whether or not the sensors have found an appropriately sized space [Boolean]
data	Accepted	Driver's input through a HVI to indicate (when prompted) that the driver accepts the space found, so PA can begin the maneuver [Boolean]
data	Declined	Driver's input through a HVI to indicate (when prompted) that the driver does not accept the space found, so PA looks for another space [Boolean]
data	SteerIn	Driver controlled physical steering wheel input as a steering wheel angle (values within the range (-360,360), with 0 being centred) [Angle in integers]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 200) [KPH in integers]
data	Threat	Input from a pre-processing threat assessment block that converts sensor inputs into a threat; For PA, it only indicates presence of obstruction [Boolean]

Table 6: Input Variables Used in Park Assist (PA)

Type	Name	Meaning
data	Throttle	Request for throttle control of the vehicle as a percentage of maximum throttle (For PA, only one constant value of throttle percentage is output, which corresponds to a reasonable acceleration for parallel parking) [Percentage in integers]
data	Brake	An output request for braking force by the feature as a percentage of maximum braking ability (For PA, only one value of braking is output, which corresponds to soft-braking) [Percentage in integers]
data	SteerOut	Output request for steering control of the vehicle (the (value -1 indicates that the vehicle shall turn the wheels to the right, 0 indicates that the wheels shall be centred, and 1 indicates that wheels shall turn to the left. Some external component will process these values and manipulate the wheels accordingly) [Steer request in integers]
data	PA_HVI	An output to represent the following information that would be given to the driver: PA_HVI = 0: PA disabled PA_HVI = 1: PA enabled, but idle waiting for speed range for searching PA_HVI = 2: PA searching PA_HVI = 3: PA prompting the driver, asking to stop the vehicle and accept or decline the space just found PA_HVI = 4: PA engaged and will display to the driver that it is currently executing the parking maneuver PA_HVI = 5: PA has completed the parking maneuver PA_HVI = 6: PA has had to abort and cannot resume, and the driver will have to try again PA_HVI = 7: PA is being overridden and will continue when reason for override ceases PA_HVI = 8: PA has encountered an error and will not be available again until the vehicle is restarted [Display value in integers]

data	PRNDL	The output representing a change of gear request by the feature with the following values assumed:
		PRNDL = 0: Park
		PRNDL = 1: Reverse
		PRNDL = 2: Neutral
		PRNDL = 3: Drive
		PRNDL = 4: Low
		[Gear request in integers]

Table 7: Output Variables Used in Park Assist (PA)

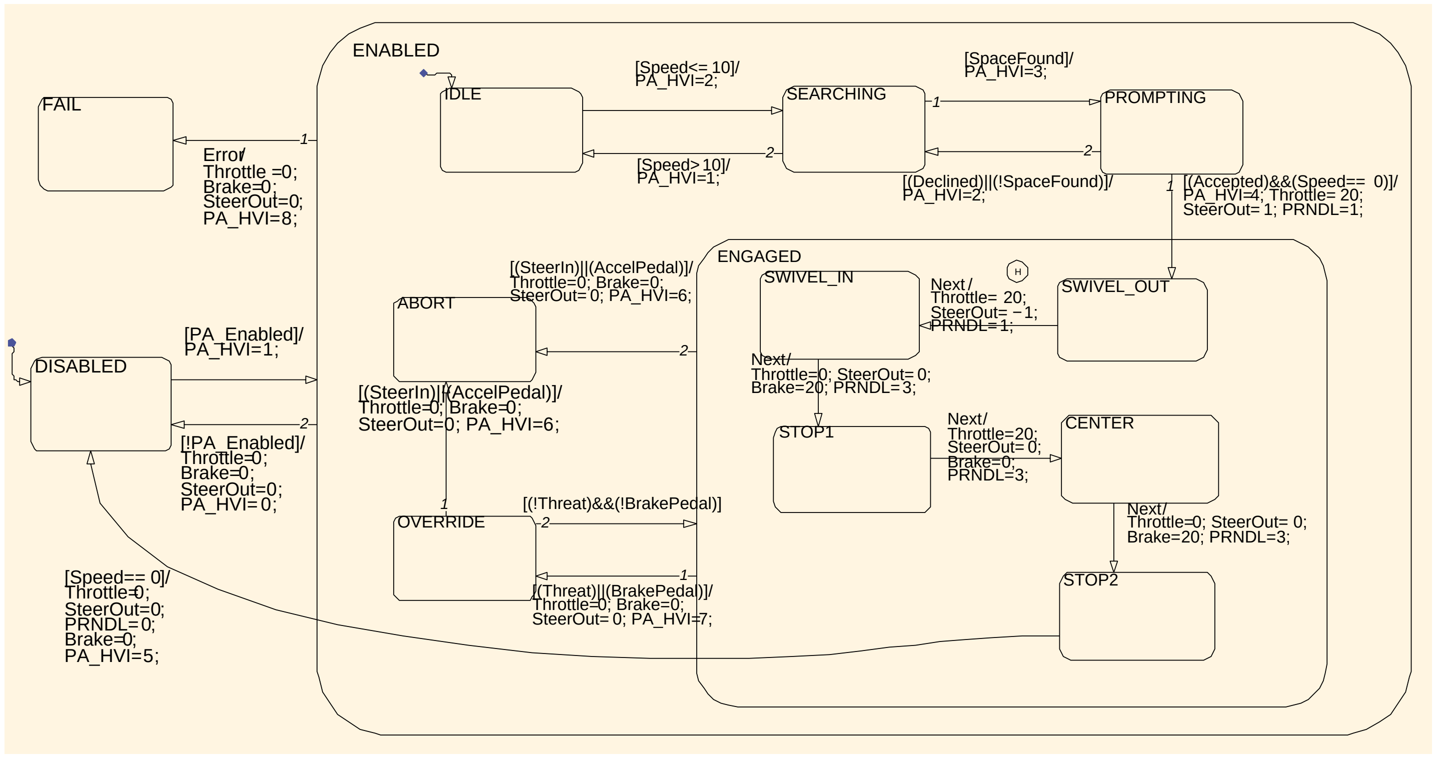


Figure 6: Park Assist STATEFLOW design model

3.4 Lane Guide (LG)

"TRW's Lane Guide Departure (LG) System supports the driver and assists in preventing unintentional lane departures. Utilizing a forward-looking video camera that continuously monitors the vehicle's lane, the system can determine whether or not a driver is unintentionally drifting from their lane or the road."

Figure 7 shows an example of the feature's execution from the TRW website.

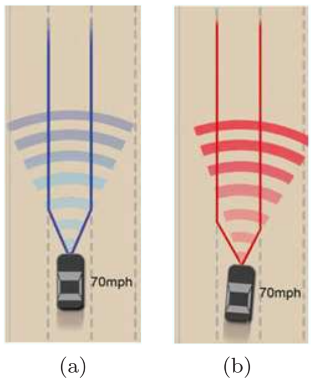


Figure 7: Lane Guide Functionality: (a) As the LG vehicle cruises in a lane, LG monitors the lane markings and the vehicle's position; (b) If vehicle drifts from its lane, LG first can warn the driver, and even provide steering corrective input.

Figure 8 presents LG's functionality modelled in Stateflow, which we will explain in the following paragraphs. Table 8 describes the input variables and Table 9 describes the output variables used in the LG's Stateflow design model.

The details of our design in Stateflow are as follows: LG should start in the DISABLED state when the car is turned on. When the driver turns LG on by pressing the LG_Enabled button, LG enters the ENABLED state. The default of the ENABLED superstate is the DISENGAGED state. We assume that the sensor data is pre-processed by an external module and that LG simply obtains a LaneDrift value indicating the amount of vehicle drifting. If the vehicle is adequately centered in the lane, LaneDrift is equal to 0. If the LaneDrift input reaches the threshold of -10, the vehicle is drifting to the left. If the LaneDrift input reaches the threshold of +10, the vehicle is drifting to the right. LG can be used in two modes: "Warn" and "Assist". Both modes are contained within the state ENGAGED. If LG_Mode = 0, this means that LG is set to "Warn", and the feature will output LG_Warning = 1 to indicate that the vehicle drifts too far to the left or to the right, but no other action other than the warning is taken. While LG_Mode = 0, if the vehicle is drifting left (*i.e.,* LaneDrift < -10), LG will enter the WARN_LEFT state, whereas if the vehicle is drifting right (*i.e.,* LaneDrift > 10), LG will enter the WARN_RIGHT state. If LG_Mode = 1, this means that LG is set to "Assist", and the feature will output LG_Warning = 0 (*i.e.,* no warning indication). While LG_Mode = 1, if the vehicle is drifting left (*i.e.,* LaneDrift < -10), LG will enter the ASSIST_LEFT state, and output SteerOut = -1; if the vehicle is drifting right (*i.e.,* LaneDrift > 10), LG will enter the ASSIST_RIGHT state and output SteerOut = 1. These SteerOut output values will indicate some external component to determine and request the steering required to center the car. If the mode is set to "Assist", the warning will not activate when the vehicle drifts because PA will act to center the vehicle in its lane and a warning during the execution of PA's centering would be annoying to the driver. The turn signal indication, brake pedal depression, and steering wheel input (over a threshold of 10 and -10 degrees of rotation) will all send LG from any state in ENGAGED to the OVERRIDE state. An Error event at any time when LG is on will cause a transition to the FAIL state, which will remain the active state until the vehicle is restarted.

Type	Name	Meaning
event	Clock	A clock signal for the STATEFLOW model
event	Error	A signal indicating when an error has occurred
data	LG_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	LG_Mode	Mode selected by the driver, indicating that LG works in either “Warn” mode (LGmode = 0) or “Assist” mode (LGmode = 1) [Mode value in integers]
data	LaneDrift	Value to indicate if the vehicle is centered or not in the lane (input from an external sensor processing component, where a 0 indicates centered, -10 indicates drifting to the left, and 10 indicates drifting to the right) [Drifting value in integers]
data	SteerIn	Driver controlled physical steering wheel input as a steering wheel angle (For LG, values are within the range (-10,10), with 0 for centered) [Angle in integers]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	TurnSignal	Value indicating if the turn signal is on or not [Boolean]

Table 8: Input Variables Used in Lane Guide (LG)

Type	Name	Meaning
data	LG_Warning	Value that indicates if the LG is providing a warning [Boolean]
data	SteerOut	An output request for steering control of the vehicle (For PA, the value -1 indicates that the vehicle shall turn the wheels to the right, 0 indicates that the wheels shall be centered, and 1 indicates that wheels shall turn to the left. Some external component will process these values and manipulate the wheels accordingly) [Steer request in integers]

Table 9: Output Variables Used in Lane Guide (LG)

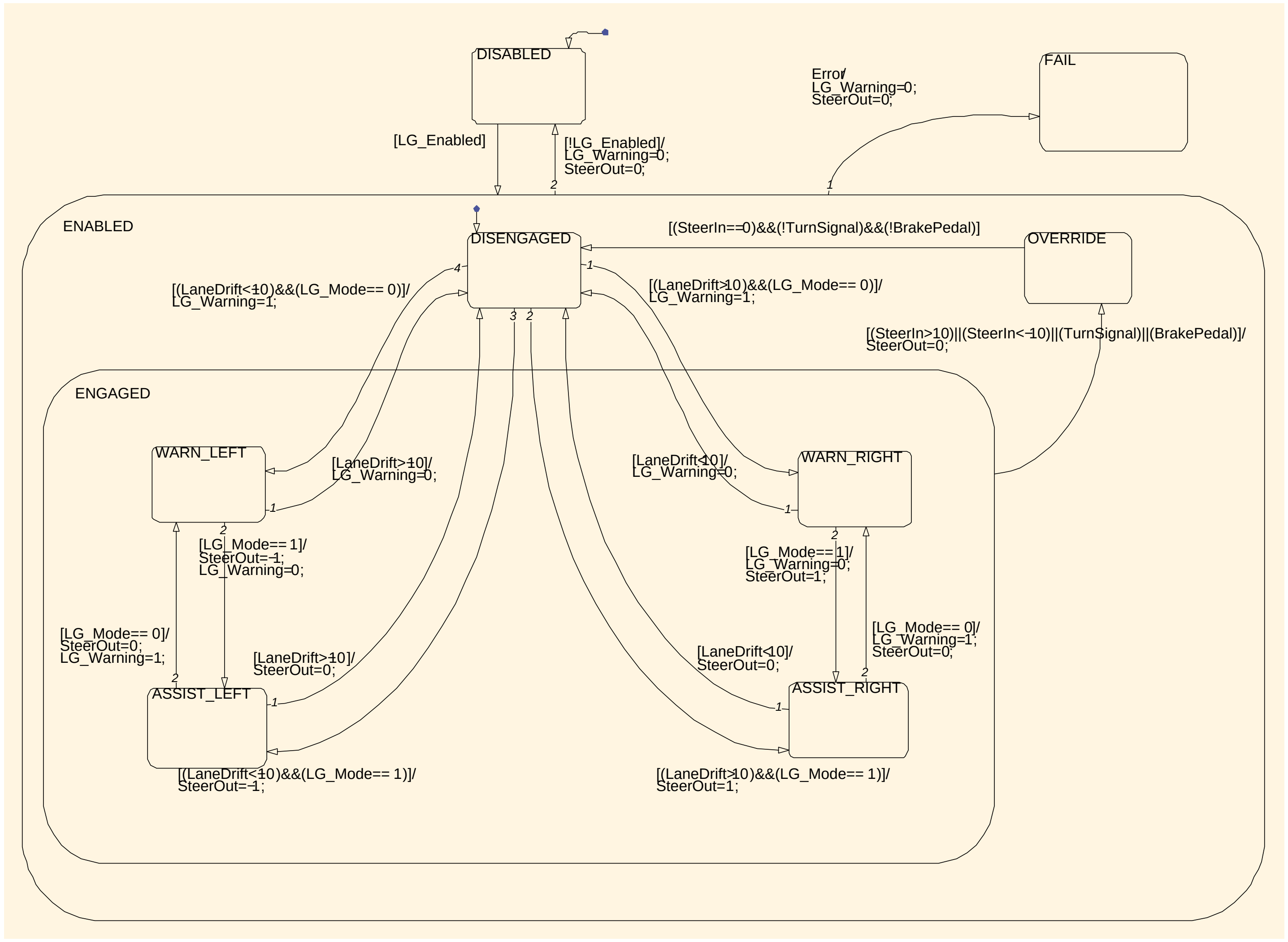


Figure 8: Lane Guide STATEFLOW design model

3.5 Emergency Vehicle Avoidance (EVA)

The Emergency Vehicle Avoidance (EVA) feature assists drivers by pulling over the vehicle in situations when an emergency vehicle, which makes use of a siren, needs the road to be cleared. Combining long and short range radars with a sound detector, and the use of a GPS device, this feature determines when and where the vehicle needs to pull over. When a siren is detected the vehicle should slow and pull to the right-side of the road, and stop if the vehicle is in a safe location. If at an unsafe location (*e.g.,* in the middle of an intersection), the vehicle will coast until it is safe to continue the stop procedure.

Figure 9 presents EVA's functionality modelled in Stateflow, which we will explain in the following paragraphs. Table 10 describes the input variables and Table 11 describes the output variables used in the EVA's Stateflow design model.

The details of our design in Stateflow are as follows: EVA starts in the DISABLED state when the car is turned on. When the driver turns EVA on by pressing the EVA_Enabled button, EVA enters the ENABLED state. At the ENABLED state, the feature will remain idle until a siren from an emergency vehicle is detected, as indicated by the Siren boolean input. When EVA receives the Siren boolean with value true, it will enter the AVOID state, which is like an engaged state where the feature can acquire control of the vehicle. The default of the AVOID superstate is the SLOW state. At the SLOW state, EVA will apply a safe and gradual braking, as implied by the action Brake = 20 in the default transition, which we assumed to be a safe and gradual constant value for braking. EVA will slow gradually and monitor the right to see if pulling over is feasible. We assume that the determination of safe location is monitored by an external component that signals EVA with a WayClear event when the location is safe, or with a DontStop event when the location is unsafe. If at the SLOW state EVA gets as an input a WayClear value true, it is safe to pullover farther to the right, and EVA will enter the PULLOVER state. At the PULLOVER state, EVA will continue to brake at the same gradual rate, but it will also request steering control by outputting SteerOut = -1, which indicates that the vehicle's wheels need to move to the right. The SteerOut output value will indicate some external component to determine and request the steering required to pull over the vehicle. In either of the SLOW or PULLOVER states, if EVA receives the DontStop boolean input value as true, EVA will enter the COAST state, which will turn off any steering or braking control to apply a slight throttle request for the vehicle to keep moving until a safe location is found. We assume that a constant of Throttle = 10 is adequate to ensure that the vehicle does not come to a stop in an unsafe location (*e.g.,* the middle of an intersection). Any brake pedal depression or steering input of non-zero, and any acceleration pedal input greater than a 30% depression from the driver will send EVA to the OVERRIDE state from any state in the AVOID superstate. An Error event at any time when EVA is on will cause a transition to the FAIL state, which will remain the active state until the vehicle is restarted.

Type	**Name**	**Meaning**
event	Clock	A clock signal for the Stateflow model
event	Error	A signal indicating when an error has occurred
data	EVA_Enabled	Driver selected main power to enable/disable the feature [Boolean]
data	Siren	Input from an external component that uses GPS information and microphones to determine if an emergency vehicle is nearby and its whereabouts, so EVA can act by moving the EVA vehicle out of the emergency vehicle's path [Boolean]
data	WayClear	Input from an external component that uses GPS and radar information to indicate if pulling the EVA vehicle over into the far right lane is possible [Boolean]
data	DontGo	Input from an external component that uses GPS information to determine if the current braking action would cause the EVA vehicle to stop in an unsafe location [Boolean]
data	SteerIn	Driver controlled physical steering wheel input as a steering wheel angle (values within the range (-360,360), with 0 being centered) [Angle in integers]
data	BrakePedal	Value indicating the physical amount of depression of the brake pedal by the driver (0 for not depressed and any positive value when depressed) [Percentage in integers]
data	AccelPedal	Value of physical pedal input represented as a percentage of maximum depression [Percentage in integers]
data	Speed	Current speed of the vehicle (value within the range of 0 to 200) [KPH in integers]

Table 10: Input Variables Used in Emergency Vehical Avoidance (EVA)

Type	**Name**	**Meaning**
data	Throttle	Request for throttle control of the vehicle as a percentage of maximum throttle (For EVA, only one constant value of throttle percentage is output, which corresponds to a reasonable acceleration for pulling over) [Percentage in integers]
data	Brake	An output request for braking force by the feature as a percentage of maximum braking ability (For PA, only one value of braking is output, which corresponds to soft-braking) [Percentage in integers]
data	SteerOut	An output request for steering control of the vehicle (For PA, the value -1 indicates that the vehicle shall turn the wheels to the right, 0 indicates that the wheels shall be centred, and 1 indicates that wheels shall turn to the left. Some external component will process these values and manipulate the wheels accordingly) [Steer request in integers]
data	EVA_HVI	An output to represent the following information that would be given to the driver: EVA_HVI = 0: EVA disabled EVA_HVI = 1: EVA enabled, but idle waiting for the an emergency vehicle to be detected (indicated by the siren boolean) EVA_HVI = 2: EVA engaged and executing evasive action to slow and get as far out of the way as possible EVA_HVI = 3: EVA is being overridden and will continue until reason for override ceases EVA_HVI = 4: EVA has encountered an error and will not be available again until the vehicle is restarted [Display value in integers]

Table 11: Output Variables Used in Emergency Vehical Avoidance (EVA)

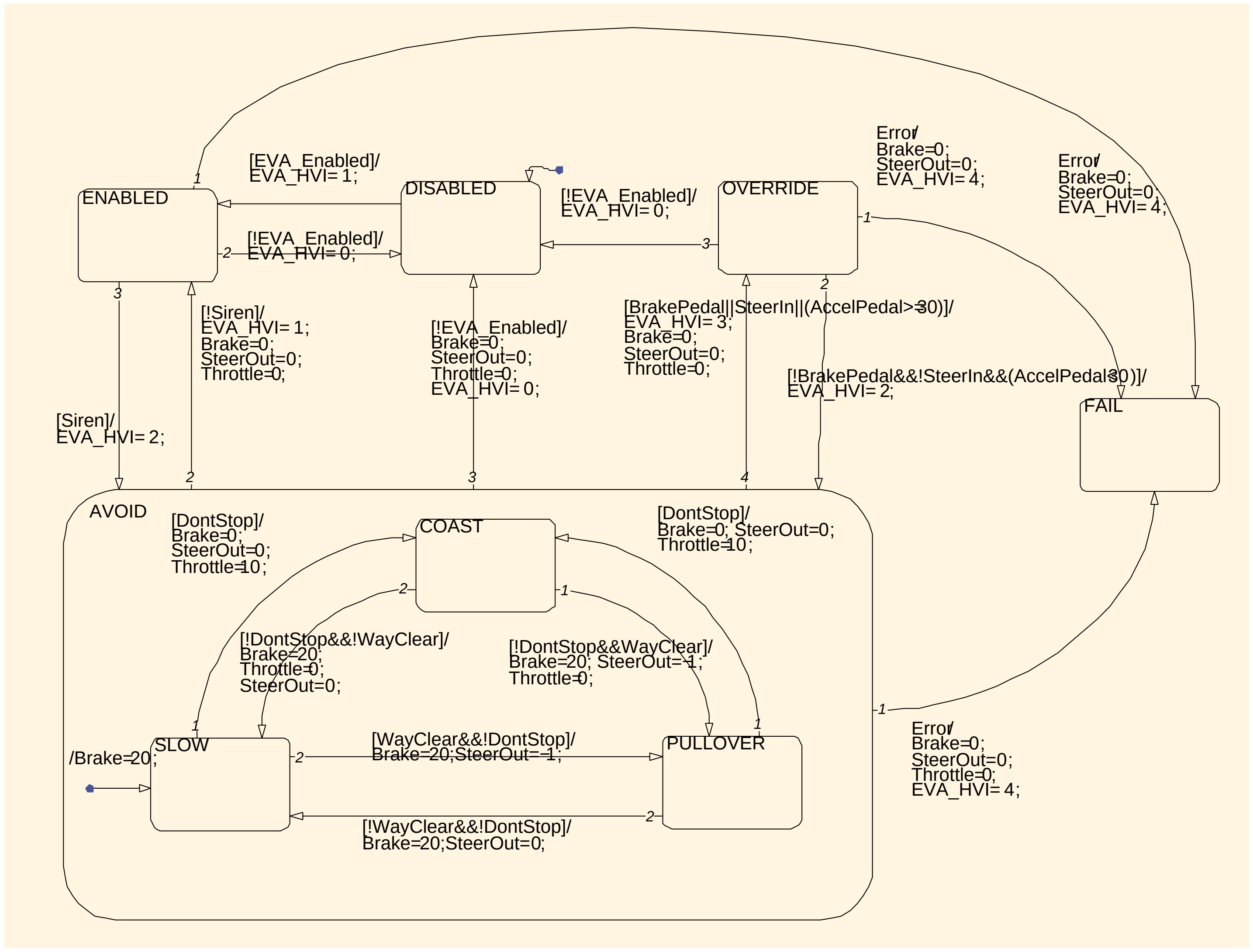


Figure 9: Emergency Vehicle Avoidance STATEFLOW design model

4 A Brief Description of Smv

In this section, we describe the SMV notation that is used in our translated models. The description is based on McMillan [13]. The general format of an SMV module model is shown in Figure 10.

An SMV model consists of a set of modules and a main module. An SMV module is composed of declarations, assignments, and optionally, assertions. Each model can also have formal parameters, which are often declared as inputs (assigned outside the module) or outputs (assigned inside the module). The *Declaration Section* contains the input, output and local type declarations, with the input and output declarations occurring before any local declarations and assignments. In the *Assignment Section*, a set of assignments of the form 'name := value;' are declared, indicating how the variables change value. Different operators can be used in the *Assignment Section*, and for our translator we used operators such as boolean ("and", "or", "not"), conditional ("if-then-else", "case", "switch"), arithmetic ("+", "-", "*", "/"), and comparison ("=", "<", ">", ">=", "<="). Special operators for describing recurrences are "init" and "next". Given that the set of values that a variable, say x, takes can be seen as a sequence of values, init(x) specifies the initial value of x (*i.e.,* the first value of x) and next(x) denotes the next value of x (*i.e.,* the (i+1)st value of x). The next value of x is defined using operators and constants from the range of values that x can take given its declaration. In the *Assertion Section* we write the properties that we want to check. An assertion is a condition that must hold true in every possible execution of the program, and in SMV each assertion is written in linear temporal logic (LTL). A macro uses the `DEFINE` statement to give a concise and meaningful name to a constant or a conditional directive.

```
MODULE Module_name (inputs, outputs)
{
        /* *** Declaration Section *** */
        INPUT input_name : [boolean | enumerated | range];
        ...
        OUTPUT output_name : [boolean | enumerated | range];
        ...
        local_name : [boolean | enumerated | range];
        ...

        /* *** Assignment Section *** */
        name := value;
        init(name2) := value;
        next(name2) := NEWvalue;
        ...

        /* *** Assertion Section *** */
        property_name: assert temporal_logic_formula;
}
```

Figure 10: General Format of SMV Models

5 Translating Stateflow to Smv: mdl2smv

In this section, we describe our tool for translating Stateflow to SMV and the design decisions we made in the creation of our translator. We also provide the syntactic rules that are used in our approach for modelling automotive features.

5.1 Motivation

For automotive embedded systems, the models that describe the functionality of the features are often created using Stateflow. Stateflow models can be included in a Simulink model by placing them in Stateflow blocks, so they can be simulated. Simulink/Stateflow stores the information about a model in an ASCII file with a .mdl extension. We can extract the necessary information from the .mdl file and translate it to the modelling notation that the tool to verify properties of the system requires. In our work, SMV is the analysis tool chosen as a target because its notation can reflect precisely the semantics of the design models in Stateflow, and that of the composition of features. Thus, we built a direct translator to SMV. We plan to perform the analysis at the level of the design because having the right level of description will allow us to apply the findings of our analysis to the design of the automotive components. Some of the properties that we are interested in verifying are the absence of feature interactions in the integration of the automotive features and the lack of errors in the design. Thus, we must refer to more than one feature during our analysis. This requirement makes our work different from previous work on translation and analysis of Stateflow models since in those efforts the purpose of property verification only considered one model at a time. Also, most of the previous work assumed that Stateflows semantics and that of Statecharts were the same, but this is not the case, so our work follows closely the semantics of Stateflow.

Our tool could potentially have been implemented as an addition, or plug-in, to the Simulink/Stateflow interface instead of as stand-alone program. The primary benefit of a plug-in implementation would be ease-of-use, since a feature could be converted to the model checker's syntax by clicking a button. However, this method would involve additional development for its integration with Simulink/Stateflow. Therefore, we opted to develop a stand-alone tool, which is more portable because a Matlab license is not be necessary. Also, there is no need for additional interface development since we simply make use of an executable program, which takes one text file as input and gives another text file as output. The translator is written in C, having as input an ASCII .mdl file and as output an ASCII .smv file.

5.2 Parsing

Parsing is the process of structuring a 'linear representation' (*e.g.,* a sentence, a computer program, a piece of music, etc.) in accordance with a given formal grammar [9]. To do so, lexical analyzers (such as Lex or Flex) and parser generators (such as YACC or Bison) are utilized to help capture the input and create an abstract syntax tree from which the output is generated.

Our lexical analyzer (abbreviated as lexer) was written in Flex, the GNU version of Lex. The token definitions describe how the input file is broken up into a sequence of tokens. Using the structure of the .mdl file, the token definitions capture the important objects, as well as their attributes and values found in the Stateflow portion of the .mdl file and ignore the rest. The Stateflow objects captured are states, transitions, junctions [1], events, and data. These objects generally include as attributes: ID, label, node relationship (for tree structure) and type-specific information such as the source state of a transition, or the data-type. In Figure 11, we show the main tokens defined using regular expressions.

```
Stateflow           return STATEFLOW;
"chart {"           return CHART;
state               return STATE;
junction            return JUNCTION;
transition          return TRANSITION;
event               return EVENT;
data                return DATA;
id                  return ID;
labelString         return LABELSTRING;
treeNode            return TREENODE;
linkNode            return LINKNODE;
firstTransition     return FIRSTTRANS;
"type {"            /* ignore this 'type' */
```

```
type                return TYPE;
executionOrder      return EXECORDER;
src                 return SOURCE;
dst                 return DESTINATION;
name                return NAME;
scope               return SCOPE;
trigger             return TRIGGER;
dataType            return DATATYPE;
[A-Z_]*_STATE       return STATE_TYPE;
[A-Z_]*_(DATA|EVENT) return DATAOREVENT;
[A-Z_]*_JUNCTION    return JUNCTION_TYPE;
[0-9]+              return NUMBER;
\"[^"]*             return STRING;
```

Figure 11: Main Tokens in the Lexical Analyzer

[1]In our translator, we currently parse junction information, but we do not do anything with it.

The lexical analyzer can be directly synchronized with a parser generator such as YACC, or the GNU equivalent called Bison. The parser generator takes the tokens defined in the lexer and breaks down the input in a recursive fashion to create an abstract syntax tree or parse tree. Our translator uses C functions called in the parser generator to create a tree for the states, showing the internal organization of states in Stateflow, and a series of linked lists for each of the Stateflow objects, which contain all the necessary information about the model. In Figures 12 and 13, we show the grammar definitions without the function calls that create the parse tree.

```
features:
        | features feature
        ;

feature:
        STATEFLOW chart_ent state_list junc_list
                  trans_list event_list data_list
        ;

chart_ent:
        CHART chart_dtls
        ;

chart_dtls:
        chart_attr
        | chart_dtls chart_attr
        ;

chart_attr:
        ID NUMBER
        | NAME STRING
        | TREENODE NUMBER NUMBER NUMBER NUMBER
        | FIRSTTRANS NUMBER
        | EXECORDER NUMBER
        ;


state_list:
        | state_ent
        | state_list state_ent
        ;

state_ent:
        STATE state_dtls
        ;
```

```
state_dtls:
        state_attr
        | state_dtls state_attr
        ;

state_attr:
        ID NUMBER
        | LABELSTRING STRING
        | STRING
        | TREENODE NUMBER NUMBER NUMBER NUMBER
        | FIRSTTRANS NUMBER
        | TYPE STATE_TYPE
        | EXECORDER NUMBER
        ;

junc_list:
        | junc_ent
        | junc_list junc_ent
        ;

junc_ent:
        JUNCTION junc_dtls
        ;

junc_dtls:
        junc_attr
        | junc_dtls junc_attr
        ;

junc_attr:
        ID NUMBER
        | LABELSTRING STRING
        | TYPE JUNCTION_TYPE
        | LINKNODE NUMBER NUMBER NUMBER
        ;
```

Figure 12: Grammar in the Parser Generator (Part 1)

5.3 Syntax Rules in Stateflow

In our translator, we assume that Stateflow design models adhere to the following syntax rules:

- Designer must specify the type of data used for all input data, output data, local data, etc. To do so, in the Stateflow Model Explorer, click data variable. Then, set “Data type mode:” to “Built-in” and set “Data type:” to appropriate type (double, boolean, etc.).
- Do not put quotes in the string of any label.
- Do not include anything other than numbers, letters, or an underscore in any names.
- Do not put any sort of spaces in data names in Stateflow.

```
trans_list:
        | trans_ent
        | trans_list trans_ent
        ;

trans_ent:
        TRANSITION trans_dtls
        ;

trans_dtls:
        trans_attr
        | trans_dtls trans_attr
        ;

trans_attr:
        ID NUMBER
        | LABELSTRING STRING
        | STRING
        | SOURCE ID NUMBER
        | SOURCE
        | DESTINATION ID NUMBER
        | EXECORDER NUMBER
        | LINKNODE NUMBER NUMBER NUMBER
        ;

event_list:
        | event_ent
        | event_list event_ent
        ;

event_ent:
        EVENT event_dtls
        ;

```

```
event_dtls:
        event_attr
        | event_dtls event_attr
        ;

event_attr:
        ID NUMBER
        | NAME STRING
        | SCOPE DATAOREVENT
        | TRIGGER DATAOREVENT
        | LINKNODE NUMBER NUMBER NUMBER
        ;

data_list:
        | data_ent
        | data_list data_ent
        ;

data_ent:
        DATA data_dtls
        ;

data_dtls:
        data_attr
        | data_dtls data_attr
        ;

data_attr:
        ID NUMBER
        | NAME STRING
        | SCOPE DATAOREVENT
        | DATATYPE STRING
        | LINKNODE NUMBER NUMBER NUMBER
        ;
```

Figure 13: Grammar in the Parser Generator (Part 2)

- Ensure that actions written in transitions have the form " x = y ; ". Some functions such as graphical functions and MATLAB functions were not needed while modelling our automotive features. Therefore, they are not currently supported in our translator.
- Ensure that no two STATEFLOW models have the same name for their SIMULINK block name (*i.e.*, "Chart", which is the default), and provide a meaningful name.
- When using our translator, if a parsing error occur, you may need to ensure that all label strings are on a single line. (Parsing errors were occurring with older MATLAB versions due to newline characters such as \n and \r).

5.4 Translation functions in C

Our code is based on the series of nodes stored in memory as linked lists and a tree. We defined a generic node structure to hold any necessary information required for all STATEFLOW entries. This simplifies our translation process by having generic functions for creating and traversing, instead of having to write different functions for creating and traversing each kind of STATEFLOW object. For each STATEFLOW entry found in the .MDL file, a node is created and stored in one of five linked lists, depending on its kind, having one link list for each of the kinds of STATEFLOW entries: states, junctions, transitions, events and data. We also created a tree structure from information found in the .MDL file to have easy access to the same internal representation that STATEFLOW uses to organize states. Table 12 shows the parameters stored in a node and which STATEFLOW entry requires them.

Parameter	Type	S	J	T	E	D
id	Integer	x	x	x	x	x
labelString	String	x		x	x	x
parentId	Integer	x				
childId	Integer	x				
leftSibId	Integer	x				
rightSibId	Integer	x				
firstTrans	Integer	x				
type	String	x	x			
execOrder	Integer	x		x		
src	Integer			x		
dst	Integer			x		
scope	String				x	x
triggerData	String				x	x
next	Pointer	x	x	x	x	x
parent	Pointer	x				
child	Pointer	x				
leftSib	Pointer	x				
rightSib	Pointer	x				

Table 12: Parameters used in the Generic Node Structure; For Stateflow Entries, we use S for State, J for Junction, T for Transition, E for Event and D for Data

Once all the Stateflow entries are stored in memory, we create an SMV file that maintains the semantics of Stateflow. Each Stateflow design model is a separate module for SMV, *i.e.,* we create one SMV module per feature design model in Stateflow. The process of creating the SMV module is described in detail in the rest of this section.

To start, the data and event linked lists are traversed to print in the module parameters all the inputs and outputs used. The data and event lists are then traversed again to declare the module's variables (inputs, outputs and local) and their possible values as enumerations, ranges, or Booleans. SMV requires its datatypes to be finite, while Stateflow allows its types to be used without value declaration. Thus, SMV needs the values that the data can take specified in the *Declaration Section.* To account for this issue, our translator prompts the user when reading non-Boolean data and requests to enter the values the data may take (range or enumeration). An example of the result of this part of the translation process is shown in the file excerpt in Figure 14.

```
MODULE CWfeature (CWfeatureevents,CW_HVI,CW_Enabled,Speed,Warning,Threat,Brake,BrakePedal,AccelPedal)
{
        INPUT CWfeatureevents : {no_event,Clock,Error};
        INPUT CW_Enabled : boolean;
        INPUT Speed : 0..100;
        INPUT Threat : 0..4;
        INPUT BrakePedal : 0..100;
        INPUT AccelPedal : 0..100;
        OUTPUT CW_HVI : 0..4;
        OUTPUT Warning : 0..1;
        OUTPUT Brake : 0..1;
```

Figure 14: SMV File Excerpt Showing Inputs and Outputs

After the variable declarations are produced, the state list is traversed to declare the states that are part of the module. We declare a variable per level of state hierarchy, having enumerated all possible states at the state hierarchy level (for OR- and AND-states). Also, we make use of the default transition information stored in the .MDL file to initialize, using the init operator, the first active state at each level. If AND-states are encountered, there is no default transition and the state with execution order of 1 is used for initialization, so this state will be the first active state and coincide with the STATEFLOW execution. An example of the result of this part of the translation process is shown in the file excerpt in Figure 15.

```
        stateCWfeature : {ENABLED,DISABLED,OVERRIDE,FAILED};
        stateENABLED : {DISENGAGED,ENGAGED,WARNING,MITIGATE,AVOID,HALT};

        init(stateCWfeature) := DISABLED;
        init(stateENABLED) := DISENGAGED;
```

Figure 15: SMV File Excerpt Showing State Declaration and Initialization

The next part of the translation process is defining the state transitions. An example of the result of this part of the translation process and the corresponding part of the STATEFLOW model being translated, which only has OR-states, is shown in the excerpts in Figure 16. This is modelled in SMV using switch statements. There is a main switch statement to account for the state machine as a whole, and an additional nested switch statement for every level of hierarchy in the STATEFLOW model. Thus, a simple state diagram of one level of hierarchy corresponds to one main switch statement. Within the switch statement is a case for each state at the corresponding level of the hierarchy. Then, within the case statement, there is a series of if-then-else statements, one per each event and/or transition condition. The order of the if-then-else statements corresponds to the transitions' priority of execution. If we are creating transitions within a nested switch statement (*i.e.,* at a hierarchy level other than one), outgoing transitions from the superstate must be listed before inner transitions, following their priority of execution. Then, internal transitions are listed; first the transition with highest priority, and consecutively creating the rest in decreasing priority. The highest transition priority is 1. This method will create an SMV model that matches the STATEFLOW semantics since SMV tests and executes statements in the order they are present. Within each if statement, there is an assignment for every output variable that corresponds to the actions shown in the transition of the STATEFLOW design model. For any output whose value is not explicitly defined in the transition of the STATEFLOW model, we assign it the value it currently holds, *e.g.,* Warning in the Figure 16. Also within the if statement, there is an assignment for the new state as a result of the transition, but such assignments depend on the kind of state we are translating: (1) For OR-states, the state assignment corresponds to the state name that is specified by the destination (dst) attribute of the transition; (2) For AND-states, the state assignment correspond to the state name that is next in execution order. All the variables that were initialized with the init operator must be updated using an assignment with the next operator.

In the remaining of the section, we explain some differences in the translation process when AND-states are present. STATEFLOW runs in a single thread during simulation, so AND-states are executed sequentially by following their respective execution order, but all AND-states in the same hierarchy use the same input (or set of inputs) when checking the transitions that can be taken. To make the explanation more concrete, consider the following example, which is illustrated in Figure 17. In Figure 17, the model **C** has two children that are AND-states in the same hierarchy level, called **A** and **B**, with order of execution 1 and 2 respectively. If at the i-th step of execution, the model gets as input the event e_1, it would check in two successive steps:

1. at step i-th, STATEFLOW checks if any transition can be taken on event e_1 in state **A**, which makes substate $\mathbf{A}_2$ active.

2. at step ($i+1$)-th (*i.e.,* a subsequent clock cycle), STATEFLOW checks if a transition can be taken on the same input event e_1 in state **B**, which makes substate $\mathbf{B}_2$ active.

```
switch(stateCWfeature)
{
          DISABLED :
                    if( ( CW_Enabled ) )
                    {
                              next(Warning) := Warning;
                              next(Brake) := Brake;
                              next(CW_HVI) := 1;
                              next(stateCWfeature) := ENABLED;
                    }
                    ENABLED :
                    if( ( ~CW_Enabled ) )
                    {
                              next(CW_HVI) := 0;
                              next(Brake) := 0;
                              next(Warning) := 0;
                              next(stateCWfeature) := DISABLED;
                    }
                    else if( ( CWfeatureevents = (Error) ) )
                    {
```

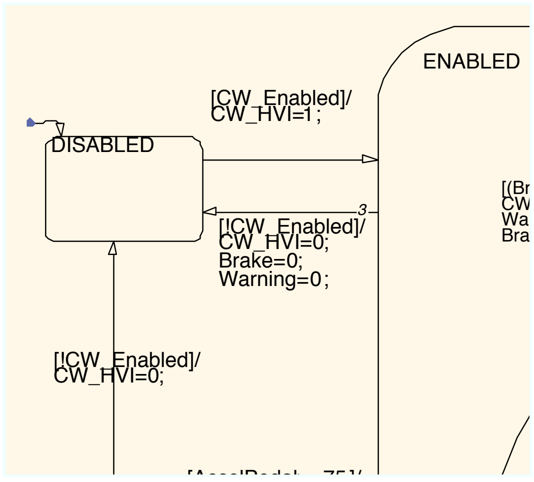


Figure 16: SMV File Excerpt Showing State Transition and CW Stateflow Design Model Excerpt

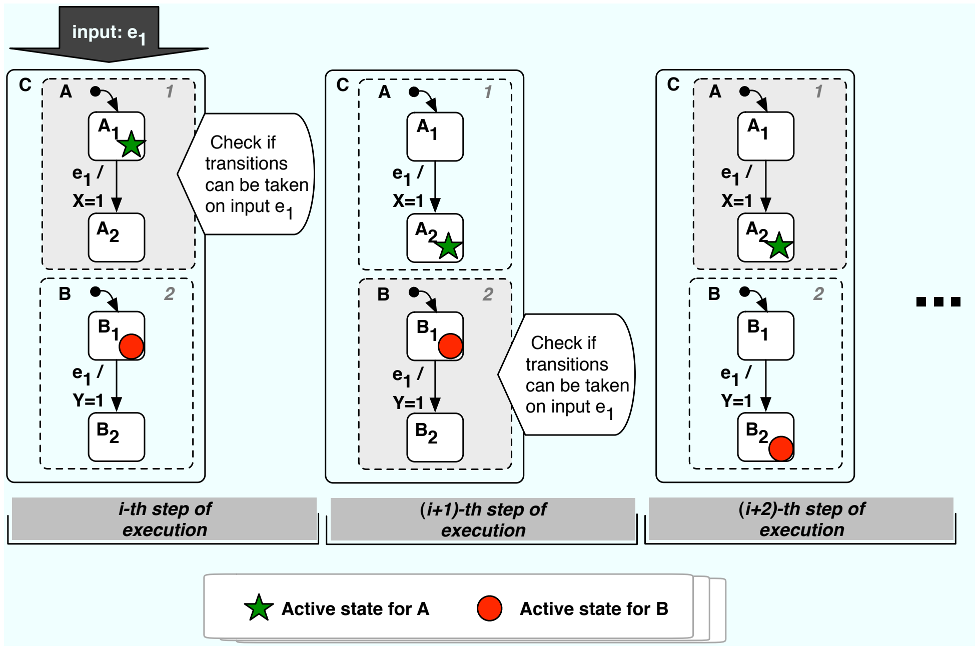


Figure 17: Illustration of Sequential Execution for AND-states in Stateflow

The semantics of Stateflow for AND-states must be preserved in the translated SMV models. The translated model that corresponds to Figure 17 is shown in Figure 18. Because AND-states must react to the same inputs, we cannot admit new inputs until all AND-states at the same level of hierarchy have executed. To ensure that a model uses the same input values to check if a transition can be taken in all AND-states at the same hierarchy level, we introduce the concept of `Stable`. As explained in Section 2, AND-states at the same hierarchy level have assigned execution order so during simulation they carry out their actions sequentially.

The system is stable when it is in the AND-state with execution order 1, *i.e.,* the first of a set of AND states in the same level of the hierarchy. When the system is stable, new inputs are admitted to the system. We translate this behaviour using an SMV macro called `next_Stable`. `next_Stable` is true when the system is moving back to the AND-state with execution order 1 (*e.g.,* `next(stateC) := A`). If `next_Stable` is true, the inputs can change non-deterministically (*e.g.,* `next(Cevents) := no_event,e1,e2,e3`). If `next_Stable` is not true, the inputs keep the value they had in the previous step (*e.g.,* `next(Cevents) := Cevents`). In this way, the input values are held while the order of execution is other than 1, *i.e.,* for all AND-states at the same level of the hierarchy. The input values are updated in the MAIN module, and given as formal parameters to the SMV models that we are analyzing.

```
MODULE C (Cevents,X,Y)
{
        INPUT Cevents : {no_event,e1,e2,e3};
        OUTPUT X : 0..10;
        OUTPUT Y : 0..10;

        stateC : {A,B};
        stateA : {A2,A1};
        stateB : {B2,B1};

        init(stateC) := A;
        init(stateA) := A1;
        init(stateB) := B1;

        init(Cevents) := no_event;

        switch(stateC)
        {
        A : switch(stateA)
           {
            A1 : if( ( Cevents = (e1) ) )
               {
                    next(Y) := Y;
                    next(X) := 1;
                    next(stateA) := A2;
               }
               else
               {
                    next(X) := X;
                    next(Y) := Y;
                    next(stateA) := A1;
               }
           }
           next(stateC) := B;
        B : switch(stateB)
           {
            B1 : if( ( Cevents = (e1) ) )
               {
                    next(X) := X;
                    next(Y) := 1;
                    next(stateB) := B2;
               }
               else
               {
                    next(X) := X;
                    next(Y) := Y;
                    next(stateB) := B1;
               }
           }
           next(stateC) := A;
     }
}

MODULE main()
{
 init(X) := 0;
 init(Y) := 0;

 DEFINE Stable := (stateC = A);
 DEFINE next_Stable := (next(stateC) = A);

 if (next_Stable)
         next(Cevents) := {no_event,e1,e2,e3};
 else
         next(Cevents) := Cevents;

 C (Cevents,X,Y);
}
```

Figure 18: SMV File Showing the Translated Model from Figure 17

Although there is still room for improvements to our translator mdl2smv, so that some advanced Stateflow syntax could be captured, the process described above is capable of translating the major details of Stateflow state machines and is sufficient to translate our UWFMS created in Stateflow. The syntax that we do not yet support is: any actions within states, connective junctions, graphical functions, Matlab functions, and temporal conditions. However, they are not used in our created UWFMS.

5.5 Combining Features

SMV was chosen as the analysis tool because it preserves the semantics of Stateflow individual models and also because it allows us to represent the semantics of combining multiple features.

In the automotive domain, when several features are integrated in a vehicle, they work concurrently, *i.e.,* they all receive the same input (or set of inputs) simultaneously. Likewise, each feature reacts to the inputs independently, not communicating with each other directly, but indirectly by making requests to the mechanical processes that change the environment in which the features work. Even though all features receive their inputs synchronously, each feature must preserve its individual Stateflow semantics, such as sequentiality of execution when a feature contains AND-states at certain hierarchy level. Whenever a feature model with several AND-states is combined with other feature models with fewer or no AND-states, the latter feature models must “idle” while the former feature model (*i.e.,* the one with more AND-states) finishes its sequential execution (*i.e.,* checking for transitions in all its AND-states with the same input). When a feature model remains “idle”, it must retain the outputs it generated in its most immediate previous step of execution. The following example explains the semantics of the combination of features while preserving the semantics of individual features, and it is illustrated in Figure 19.

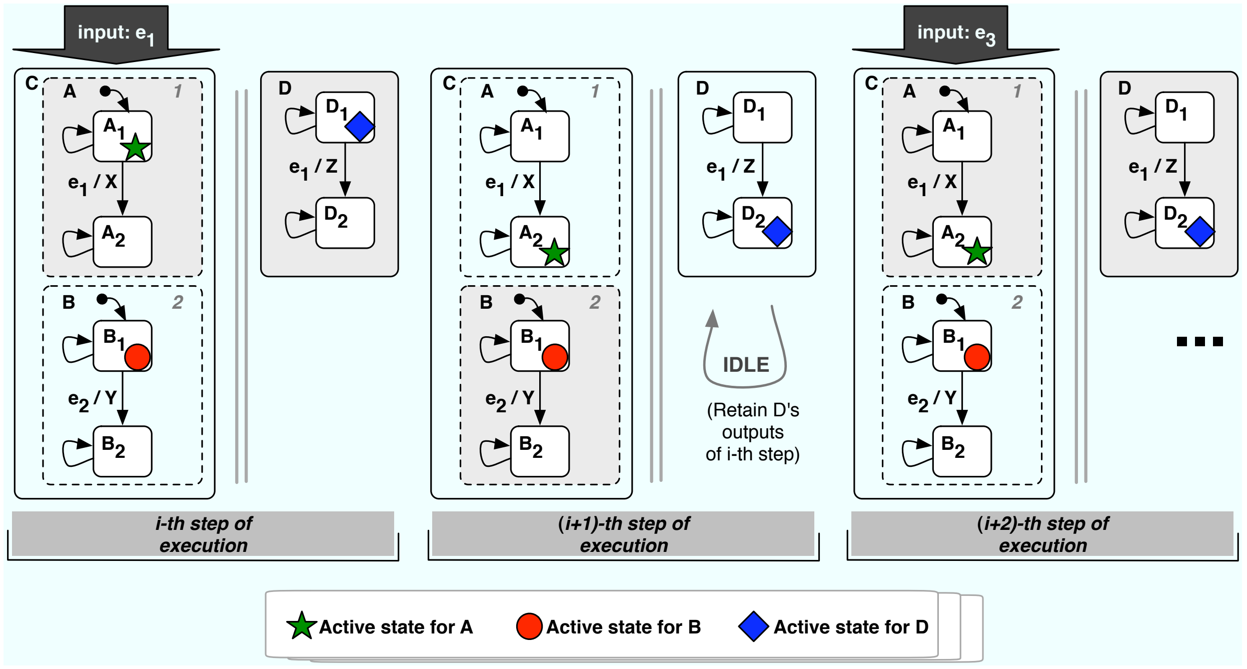


Figure 19: Illustration of Parallel Execution when Combining Multiple Features in SMV

In Figure 19, two features are modelled in Stateflow, named **C** and **D**. Feature model **A** has two children AND-states, **A** and **B** with execution order 1 and 2 respectively. When **C** and **D** are combined, they synchronize when the inputs are received for all features. Inputs are received when all the features are in their `Stable` step. If at step i input e_1 is received, **C** will take two steps to process the input since it has to check for transitions that can be taken in its two children AND-states, while **D** will only take one step to process the same input. At step $i+1$, both **A** and **D** take a transition on the input e_1, and now it should be checked if **B** can take a transition while **D** idles. At step $i+2$, both feature models can process another input, say e_3, because **C** already finished its sequential execution.

We are currently implementing a feature composition operator in SMV to capture these semantics.

6 Related Work

In this section, we overview other efforts to translate Stateflow design models into an input representation for a tool that allows property verification. The translation tool descriptions are in no particular order. The main differences with our work are: (1) Most of the previous work assumed that the Stateflow semantics and that of Statecharts were the same, but this is not the case, so our work follows closely the semantics of Stateflow during the translation process while previous efforts mostly follow Statecharts semantics; (2) To study feature interaction, we must create a composed model of multiple features since in the automotive domain, when several features are integrated in a vehicle, they work concurrently, *i.e.,* they all receive the same inputs simultaneously. This requirement makes our work different from previous work on translation and analysis of Stateflow models since in those efforts the purpose of property verification only considered one model at a time, so they are not likely to work with our integration of feature models methodology.

Scaife *et al.* developed a tool that translates from a Stateflow model into an input to Lustre [10], and the tool is called ss2lus [16]. Lustre is a synchronous language, and variables in the language are called flows (*i.e.,* infinite stream of values). A flow's value is its instantaneous value in a particular reaction, and at a particular instant in time, outputs only depend on current or previous inputs. The translator preserves Stateflow semantics, and supports the translation of most of the Stateflow objects, including event broadcasting provided that the broadcasting is bounded by a reasonable small value. To encode states into Lustre, each state is represented as a boolean variable and some code is added to update that variable according to the incoming and outgoing transitions, as well as the actions associated to the state. The update of the state's variable should use dependencies to force order in the action's evaluation. Some of the actions that can be associated with states are entry actions, exit actions, and actions in transitions. To translate a hierarchical machine, the hierarchy is represented as function calls of nested states with the only complication being the initialization and termination of nested states. The limitation of the tool is that only part of the action language is translatable. The main difference with our work is that the models generated are intended to be verified only one by one for safety properties (*e.g.,* confluence of parallel states and confluence of event broadcasting), while our models are intended to be verified several at a time to check lack of feature interactions.

Banphawatthanarak and Krogh developed a Matlab program that generates an SMV program from a Stateflow model so that properties of the Stateflow system can be investigated, although in the paper there is no description of what are the properties to be verified [5]. The tool is called sf2smv and it does not run in current versions of Matlab. In the translation process, this tool creates an SMV module per OR- and AND-state, plus a module to coordinate the status of AND-states (*i.e.,* 'not-active', 'active-active' or 'active-wait'). The conditions for making transitions to each of these states is passed as variables to the coordinator AND-state module. Since the tool allows event broadcasting (only maximum one event), two additional variables are used for each AND-group: working-state (state currently being processed) and storing-state (state that was being processed when the local event occurred). Transitions are realized by an SMV variable transition in the main module, which represents the transition that is currently being processed. Only one event is processed at a time, and input or local data, if used, should be Boolean. The limitations of the tool are that input data must be Boolean, there is no support of transition_actions, output signals are not computed, and no more than one level of nested event broadcasting can exist. Only one module is verified for properties at a time. The main differences with our work are: our translator tool is standalone and does not depend of Matlab (sf2smv does not currently work since it was written in an older version of Matlab), we support input data other than Booleans as well as transition_actions, which are necessary for the modelling of automotive features, and our generated models are better suited to be used for verification of integration of features.

Pingree and Mikk, part of the NASA Jet Propulsion Laboratory, developed the "HiVy Toolset" as an automatic translation tool set from Stateflow models to the input language of Spin for the validation of mission-specific components [15]. An abstract syntax of hierarchical sequential automata (HSA) is provided as an intermediate format for the tool set. The HiVy toolset programs include *sfParse* (creation of parse tree), *sf2hsa* (Stateflow to HSA), *hsa2pr* (HSA to Promela) and the HSA merge facility. For the translation process, each model is associated with an equivalent HSA that consists of a finite set of cooperating sequential automata that can be implemented as parallel processes in Promela. Stateflow states, events and variables are encoded as Promela variables and Promela processes change the values of these variables to simulate state changes, event generation and variable changes according to the semantics of STATEFLOW. In the translated model, there is one Promela process per OR-state, but there is no indication of support for AND-state translation. Safety properties are checked in an individual generated model. The limitations of the tool are that it does not allow implicit event generation, inner transitions with the same source and destination, transition actions in transitions segments that end on a junction, and history junctions. The main differences with our work are: our translator tool allows inner transitions, which are important part of our automotive features, our tool also supports AND-states, and our generated models will allow us to integrate features using parallel composition semantics, whereas Promela would only allow interleaving composition of features.

Agrawal *et al.* developed a translation algorithm that converts a subset of the MATLAB's SIMULINK and STATEFLOW (MSS) modelling language into an equivalent Hybrid Automata (HA) [4]. The translator process transforms models expressed in the MSS language into the Hybrid System Interchange Format (HSIF), which is based on XML. The HSIF has been developed to represent dynamic networks of hybrid automata, so the generated models can be verified using HSIF verification tools, although in the paper there is no description of what are the properties to be verified. The translation algorithm was specified and implemented using a graph transformation tool called Graph Rewriting and Transformation (GReAT) [3]. The GReAT graph transformation language is used to specify transformations on subgraphs following rules. A rule contains a pattern graph that consist of vertices and edges and it is the basic unit of transformation. GReAT also has a high-level control flow language, built on top of the transformation language, that defines constructs such as: sequencing, hierarchy, recursion and branching. The sequencing construct is used to specify the order in which a set of transformation rules should execute. The sequence of rules followed to translate the STATEFLOW part is: *CreateHierarchicalStateChart* (STATEFLOW to internal representation), *HSM2FSM* (hierarchical concurrent state machine to "flat" finite state machine), *CreateVarAs* (associate corresponding input/output signals), *StateSplitting* (sequence of rules) and *Reachability* (eliminate unreachable states). The hierarchy construct is used in the *StateSplitting* part of the mapping, which is composed of several rules: a rule that infers the signal values that are not explicitly defined at each state; if a signal's values cannot be inferred, a rule splits states based on the different values that the signals can take at each state; and finally, a rule that transfers the transitions to the newly created state machine. For example, consider a state *Low* that has two signals associated to it, say *v1* and *v2*, where *v1* is explicitly defined as 1 and *v2* is undefined. If after inferring the signal's values *v2* is still undefined, but can take values 0 and 1, state *Low* would be split into two: *Low10* (with label based on {*v1*=1, *v2*=0}) and *Low11* (with label based on {*v1*=1, *v2*=1}). One of the limitations of the tool is that it only translates a subset of the MSS modelling language since some MSS components are not possible to be expressed in HSIF. However, the paper does not specify which components are the ones that cannot be expressed. Another limitation is that the mapping between HSIF and MSS constructs were determined by hand instead of using a simulation algorithm because HSIF is defined using mathematical definitions in English rather than an operational definition, so it is hard to verify the correctness of the translation. The main differences with our work are: our tool takes as an input a textual representation instead of a graphical one, there is no need for us to infer signal values, and our translated model can be verified using a simulation relation with the original model.

Kalita and Khargonekar developed a tool, called SF2STeP, to translate a timed STATEFLOW model into a fair transition system, which can be used as an input to STeP [12]. STeP (Stanford's Temporal Prover) is a formal verification tool, which combines theorem proving with model-checking to verify temporal specifications of reactive systems [6]. In the paper, timed STATEFLOW models are considered since they have the same execution semantics as TTMcharts. TTMcharts were introduced by Ostroff and are similar to Statecharts, but with simpler execution semantics [14]. A STATEFLOW model can be augmented to a timed STATEFLOW model based on the execution semantics of TTMcharts by associating a lower bound and an upper bound in time to each of the transitions of a STATEFLOW model. Also, for every state of the STATEFLOW model which has outgoing transitions, a superstate containing two substates "Idle" and "TransitionTimer" is created. The substate "TransitionTimer" ensures that the timed STATEFLOW model follows the semantics of a TTMChart by using a counter that is started as soon as the enabling condition of a transition T is satisfied. If the enabling condition of the transition T remains satisfied until the counter reaches *ub* (where *ub* is a random number between the lower and upper time bounds), the transition action is executed and the destination state of transition T becomes active; otherwise, the source of transition T remains active. A fair transition system, which STeP needs as an input, can be translated from a timed Stateflow model because of its simple execution semantics. The tool SF2STeP uses built-in Matlab's commands to obtain information about Stateflow's object hierarchy and retrieve parameters associated to the Stateflow objects, so that the output file in the syntax of a fair transition system can be produced. However, only the syntax of a fair transition system is given, but no explanation of what is the mapping between the objects in the timed Stateflow model and the elements of a fair transition system, nor details of how the mapping is performed. The paper indicates that the properties which have been verified in STeP on the output from SF2STeP are safety, liveness, timing and fail-safe. The main limitation of this tool is that it uses a timed Stateflow model instead of a generic Stateflow. The differences with our work are: our tool differentiates events and conditions in transitions, which are relevant for the automotive models we designed, while the timed Stateflow models treat them as the same entity; and while our translation approach preserves the semantics of Stateflow (particularly sequential execution for AND-states), SF2STeP follows Statechart semantics.

Camera developed a tool called SF2VHD to translate Stateflow to VHDL for his master's project [7]. Since Stateflow is a rich graphical and textual language that does not map natively into hardware, it is the goal of SF2VHD to find suitable hardware implementations for the Stateflow constructs. SF2VHD is a program written in C and C++, which parses the .mdl file containing the textual description of Stateflow objects, and constructs an internal representation that is used to generate VHDL code. The generation of VHDL code is composed of two processes: (1) Stateflow datatypes are converted into VHDL datatypes, both at declaration and within every expression operation; (2) Stateflow expressions (including operators) must be converted into their VHDL equivalents line by line. The translator intends to preserve Stateflow semantics in the Stateflow objects that it supports. The limitation of the tool is that the following Stateflow constructs are not supported by the translator because they would lead to inefficient or redundant hardware implementations: events, AND-states, history junctions, junctions on default transitions, exit actions, transition actions, and the use of mixed arithmetic and logical operations. The main difference with our work is that our tool supports Stateflow constructs that are needed in the modelling of automotive features whereas SF2VHD disallows them, such as events and AND-states.

7 Conclusions

Feature interaction in the automotive domain is a relatively new area of study and will remain of interest in the future as vehicles continue to increase in complexity. This report helps in this research effort by producing the base for future feature interaction analysis, creating a set of non-proprietary automotive feature design models in Stateflow, called "University of Waterloo Feature Model Set" (UWFMS), and developing of a tool to translate such design models into SMV for analysis.

The translator described in this report, mdl2smv, captures all the functionality necessary to translate our UWFMS modelled in Stateflow to the input notation of the model checker SMV. The semantics of Stateflow are maintained by the SMV generated representation. The translated SMV models will allow us to verify properties of the automotive features at the same level of description as the design, so that the findings of our analysis can be applied to the design of the final automotive components. Some of the properties that we are interested in verifying are the absence of feature interactions in the integration of the automotive features and the lack of errors in the design. During our analysis, we must refer to more than one feature while performing analysis since in the automotive domain, when several features are integrated in a vehicle, they work concurrently, *i.e.,* they all receive the same inputs simultaneously. In contrast, other related efforts to translate Stateflow design models for property verification only consider one model at a time. Also, most of the previous work assumed that Stateflows semantics and that of Statecharts were the same, but it is not the case, so our work follows closely the semantics of Stateflow.

The translation tool mdl2smv has the capability of translating all the Stateflow syntax that we used to model our UWFMS. However, there is still some room for improvement, by adding, if necessary, the ability to translate notation such as actions in state labels, connective junctions, graphical functions, Matlab functions, the In(state_name) condition function, temporal conditions, as well as history states.

As future work, we plan to use SMV on our translated models to detect feature interactions in the automotive domain. The automobile industry will continue to advance, making feature interactions an ongoing problem since features often evolve as a product line. Translators, such as the one we developed, which allow feature interaction analysis, are a step in the right direction to prevent undesirable interactions from ever occurring, or at least decrease the severity of their results.

References

[1] Stateflow documentation, http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/.

[2] TRW automotive - the global leader in automotive safety systems, `http://www.trw.com/`.

[3] A. Agrawal. Graph rewriting and transformation (great): A solution for the model integrated computing (mic) bottleneck. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 03)*, pages 364–368. IEEE Computer Society, 2003.

[4] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. In *Electronic Notes in Theoretical Computer Science of the International Workshop on Graph Transformation and Visual Modeling Techniques*, 2004.

[5] C. Banphawatthanarak and B. H. Krogh. Verification of stateflow diagrams using smv: sf2smv 2.0. Technical Report CMU-ECE-2000-020, Carnegie Mellon University, 2000.

[6] N. Bjorner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *International Conference on Computer Aided Verification (CAV 96)*, volume LNCS 1102, pages 415–418. Springer-Verlag, 1996.

[7] K. Camera. SF2VHD: A stateflow to VHDL translator. Master's thesis, University of California, Berkeley, 2001.

[8] J. Dabney and T. L. Harman. *Mastering Simulink*. Pearson/Prentice Hall, 2004.

[9] D. Grune and C. J. Jacobs. *Parsing Techniques - A Practical Guide*. Ellis Horwood Limited, first edition, 1998.

[10] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. In *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, volume 18, pages 785–793, 1992.

[11] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[12] D. Kalita and P. P. Khargonekar. SF2STeP: A CAD tool for formal verification of timed stateflow diagrams. In *Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design*, pages 156–162, 2000.

[13] K. L. McMillan. *Symbolic model checking*. Kluwer Academic, 1993.

[14] J. Ostroff. A visual toolset for the design of real-time discrete-event systems. In *IEEE Transactions on Control Systems Technology*, volume 5, pages 320–337, 1997.

[15] P. J. Pingree and E. Mikk. The hivy tool set. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV) 2004*, volume 3114, pages 466–469. Springer, 2004.

[16] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a “safe” subset of simulink/stateflow into lustre. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268. ACM Press, 2004.